PREMIS Event Service Documentation

Release 1.0

UNT Libraries

February 09, 2016

Contents

Contents:

CHAPTER 1

Overview

For a general overview of the PREMIS Event Service, please see the project README.rst file (included below for convenience).

1.1 PREMIS Event Service

PREMIS Event Service is a Django application for managing PREMIS Events in a structured, centralized, and searchable manner.

1.1.1 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.

1.1.2 Dependencies

- Python 2.6+ (not Python 3)
- Django (tested on 1.6.1; at least 1.3 or higher required)
- lxml (requires libxml2-dev to be installed on your system)

1.1.3 Documentation

Documentation, including installation instructions, can be viewed online at:

http://premis-event-service.readthedocs.org/

The documentation is also browsable locally from within the `docs` directory of this repository. You can read the source files in plain text from the `docs/source` directory, or generate your own local copy of the HTML files by doing the following:

1. Make sure Sphinx is installed (`pip install sphinx`)
2. `cd docs`

3. `make html`
4. Open `index.html` (generated in `docs/build/html`)

1.1.4 License

See LICENSE.

1.1.5 Acknowledgements

The Premis Event Service was developed at the UNT Libraries and has been worked on by a number of developers over the years including

- Kurt Nordstrom
- Joey Liechty
- Lauren Ko
- Stephen Eisenhauer
- Mark Phillips
- Damon Kelley

If you have questions about the project feel free to contact Mark Phillips at mark.phillips@unt.edu

1.1.6 Developing

To take advantage of the dev environment that is already configured, you need to have Docker and Docker Compose installed.

Install Docker

Install Docker Compose

```
$ pip install docker-compose
```

Clone the repository.

```
$ git clone https://github.com/unt-libraries/django-premis-event-service.git
$ cd django-premis-event-service
```

Start the app and run the migrations.

```
# start the app
$ docker-compose up -d

# optional: add a superuser in order to login to the admin interface
$ docker-compose run --rm app ./manage.py createsuperuser
```

The code is in a volume that is shared between your workstation and the app container, which means any edits you make on your workstation will also be reflected in the Docker container. No need to rebuild the container to pick up changes in the code.

However, if the requirements files change, it is important that you rebuild the app container for those packages to be installed. This is something that could happen when switching between feature branches, or when pulling updates from the remote.

```
# stop the app
$ docker-compose stop

# remove the app container
$ docker-compose rm app

# rebuild the app container
$ docker-compose build app

# start the app
$ docker-compose up -d
```

1.1.7 Running the Tests

To run the tests via Tox, use this command.

```
$ docker-compose run --rm app tox
```

To run the tests only with the development environment.

```
$ docker-compose run --rm app py.test
```

CHAPTER 2

Technical Overview

2.1 Events

A standard PREMIS event encoded as XML looks something like the following:

```
<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 service 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 service 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.

2.2 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://institution.edu/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; see the Administration section to learn how.

CHAPTER 3

Installation

The project's README.rst file contains some basic installation instructions. We'll elaborate a bit in this section.

3.1 Dependencies

- Python 2.7.x
- Django 1.8
- libxml2-dev libxslt-dev
- Django Admin - `django.contrib.admin`

3.2 Important security warning

This application **does not** attempt to authenticate requests or differentiate between clients in any way – even for write and edit operations via the API. Do not simply expose the application to the public in your server configuration. Instead, use a network firewall to whitelist the server to authorized clients, or use a web server configuration directive (such as Apache's `<LimitExcept GET>`) to set up who is allowed to POST/PUT/DELETE events.

3.3 Install

1. Install the package.

```
$ pip install django-premis-event-service
```

2. Add `premis_event_service` to your `INSTALLED_APPS`. Be sure to add `django.contrib.admin` if it is not already present.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    # ...
```

```
        'premis_event_service',
    )
```

4. Include the URLs.

```
    urlpatterns = [
        url(r'', include('premis_event_service.urls'))
        # ...
        url(r'^admin/', include(admin.site.urls)),
    ]
```

5. Migrate the database.

```
    $ python manage.py migrate
```

6. Continue to Administration to begin setting up Agents.

CHAPTER 4

Configuration

All configuration related to the PREMIS Event Service takes place inside your project's `settings.py` file.

Note: Make sure you only make changes in your project's `settings.py`, not the `settings.py` file inside the `premis_event_service` app directory.

4.1 Mandatory Configuration

1. Update your INSTALLED_APPS setting as follows:

```
INSTALLED_APPS = (
    ...
    'django.contrib.humanize',
    'premis_event_service',
)
```

2. Make sure you have a `TEMPLATE_CONTEXT_PROCESSORS` setting defined containing at least the entries shown below:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
    'django.core.context_processors.request',
)
```

3. In your `MIDDLEWARE_CLASSES` setting, remove or comment out the `CsrfViewMiddleware` entry:

```
MIDDLEWARE_CLASSES = (
    ...
    #'django.middleware.csrf.CsrfViewMiddleware',
    ...
)
```

4. Add a `MAINTENANCE_MSG` setting at the bottom of the file:

```
MAINTENANCE_MSG = ''  # Message to show during maintenance
```

4.2 Customizing the Controlled Vocabulary

4.2.1 Deciding on Controlled Vocabulary Design

The Premis Event Service was designed to us a wide variety of or identifiers for values within PREMIS Event Objects. That being said there are some best practices that can be suggested to new a implementer.

It is advantageous for someone implementing the Premis Event Service to make use of existing controlled vocabularies whenever possible for some of the concepts that are used throughout the application. For example the Library of Congress has added a number of Preservation Vocabulary entries to its Authorities and Vocabularies Service. Starting with these identifiers for concepts such as "Fixity Check", "Replication", "Ingestion", or "Migration" is a suggestion unless there is a reason to deviate from these in a local implementation.

Additional concepts that are not covered by the Library of Congress Authorities and Vocabularies Service are those for the outcome of an event, for example "Success" and "Failure". The Premis Event Service has placeholders set aside for these values that utilize the controlled vocabularies at the University of North Texas: http://purl.org/NET/untl/vocabularies/

The Premis Event Service will work without fully fleshed out controlled vocabularies, and the authors have worked to give examples with reasonable values which can be added to or modified to meet local needs.

4.2.2 Configuring a Custom Controlled Vocabulary

The Event Service makes no attempt to validate values given to it against any set of allowed values; it is up to your policies and integrations to enforce consistency across the events you store.

However, you can change the choices that are shown in the "Search" interface by adding some statements like these to your `settings.py` file:

```
EVENT_OUTCOME_CHOICES = (
    ('', 'None'),
    ('http://purl.org/net/untl/vocabularies/eventOutcomes/#success', 'Success'),
    ('http://purl.org/net/untl/vocabularies/eventOutcomes/#failure', 'Failure'),
)
EVENT_TYPE_CHOICES = (
    ('', 'None'),
    ('http://id.loc.gov/vocabulary/preservation/eventType/fix', 'Fixity Check'),
    ('http://id.loc.gov/vocabulary/preservation/eventType/rep', 'Replication'),
    ('http://id.loc.gov/vocabulary/preservation/eventType/ing', 'Ingestion'),
    ('http://id.loc.gov/vocabulary/preservation/eventType/mig', 'Migration'),
)
```

CHAPTER 5

Administration

This section outlines the initial work needed after installation in order to prepare your Event Service for use.

5.1 Manage User Accounts

At this point, you might only have one user account in your system (which is the superuser account created when you ran `python manage.py syncdb` during installation).

To manage or create other user accounts, do the following:

1. Visit the Django admin interface (`http://[host]/admin/`) in a web browser.
2. Log in using your superuser account (if you haven't already).
3. Click **Users**. This takes you to the list of Users.
4. Click the **Add user** button near the top-right corner of the page.
5. Fill and submit the form.

Keep in mind that any account needing the ability to also administer user accounts using the admin interface will need to be given "superuser" status.

5.2 Create an Agent

Every event stored in the Event Service must be associated with an Agent. Agents merely represent entities that produce events. In many cases these are software processes (e.g. a web application or a script), but an agent can also be a person, an institution, or anything else.

To create a new agent (or to manage existing ones), do the following:

1. Visit the Django admin interface (`http://[host]/admin/`) in a web browser.
2. Log in using your superuser account (if you haven't already).
3. Click **Agents**. This takes you to the list of Agents, which will be empty at first.
4. Click the **Add agent** button near the top-right corner of the page.

5. Fill and submit the form.

Create as many agents as you have a need for.

CHAPTER 6

Using the Event Service

There are two ways of using the PREMIS Event Service:

- using the web interface to view and manage events by hand
- using the APIs to create or query events from other software workflows

This document will cover how to use the web interface and admin site. For information about the APIs, refer to the next section (API).

6.1 Events

6.1.1 Browse all Events

URL: `http://[host]/event/`

Human readable HTML listing of events.

6.1.2 View a single Event

URL: `http://[host]/event/[id]/`

Human readable HTML listing of a single event. Contains links to other formats/representations of the event, such as PREMIS XML.

6.1.3 Search for Events

URL: `http://[host]/event/search/`

Web interface for searching events. Events can be filtered by outcome, type, start/end dates, or Linked Object ID.

6.2 Agents

6.2.1 Browse all Agents

URL: `http://[host]/agent/`

Human readable HTML listing of agents.

6.2.2 View a single Agent

URL: `http://[host]/agent/[id]/`

Human readable HTML listing of a single agent. Contains links to other formats/representations of the agent, such as PREMIS XML.

CHAPTER 7

API

The bulk of event creation using the Event Service will probably take place via software as opposed to by hand. This section explains the AtomPub API (Application Programming Interface) used for interacting with the Event Service from your custom applications and scripts.

7.1 Introduction

The PREMIS Event Service 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.

7.2 API URL Structure

APIs for communicating with the Event Service programmatically are located under the `/APP/` URL tree:

7.2.1 /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.

7.2.2 /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 serves as a 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 thatallow 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.

7.2.3 /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.

7.2.4 /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.

7.2.5 /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.

7.3 Example

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 localidentifier (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. To begin with, we run the `validate_object` script on our directory and wait for it to run.

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 Event Service manages the event identifiers, and assigns new ones upon ingest.

Now, in order to send the event to the Event Service, it must be wrapped in 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 previously-generated PREMIS XML going inside of the "content" tag.)

Now that the entry is generated and wrapped in a valid Atom document, it is ready for upload. In order to do this, we POST the Atom XML to the `/APP/event/` URL.

When the Event Service 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 have been 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 `/APP/event/bfa2cf2c2a4f11e089b3-005056935974`, which would return an Atom entry containing the final event record, which we could then analyze and use for whatever purposes desired.

CHAPTER 8

Development

Here, you will find some information helpful if you plan on developing upon or making changes to the Event Service source code itself.

8.1 Project Structure

The PREMIS Event Service is structured as a common Python project, providing a Python package named *premis_event_service* which is a Django app:

```
premis_event_service/ # The Django app itself. Technically a Python package.
    admin.py          ## Customizes the Django admin interface
    coda/             ## Some supporting modules we need for some tasks
        anvl.py
        bagatom.py
        __init__.py   ### Marks this directory as a valid Python package
        util.py
    forms.py          ## Form processing code
    helpy.py
    __init__.py       ## Marks this directory as a valid Python package
    models.py         ## Database model definitions
    populator.py
    presentation.py
    settings.py       ## Wrapper for project settings file with custom defaults
    templates/        ## HTML templates
    urls.py           ## URL routing patterns
    views.py          ## View generation code
```

If you're not sure where to look for something, *urls.py* is usually the best place to start. There you'll find a list of every URL pattern handled by the application, along with its corresponding view (found in *views.py*) and arguments.

8.2 Models

Models define the data objects Django keeps in its database. The PREMIS Event Service defines these three:

- Event - Represents an event.
- Agent - Represents an agent you've defined using the Django admin interface.
- LinkObject - Contains an identifier for an object in your preservation workflow. Exists for the purpose of relating multiple events that pertain to the same object.

See `premis_event_service/models.py` for the full definitions to these models.

8.3 Views

View are functions (or sometimes classes) that Django calls upon to generate the result of a request. Usually this just means rendering some HTML from a template and serving it, but sometimes this involves form processing and API interactions as well. Django decides which view to run based on what's defined in `urls.py`.

See `premis_event_service/views.py` for the full source code to all the views provided by the Event Service.