EULfedora Documentation

Release 0.23.0

Emory University Libraries

February 16, 2016

Contents

EULfedora is an extensible library for creating and managing digital objects in a Fedora Commons repository. It eases mapping Fedora digital object types to Python classes along with ingesting, managing, and searching reposited content. Its builtin datastream abstractions include idiomatic Python access to XML and RDF datastreams. They're also extensible, allowing applications to define other datastream types as needed.

The library contains extra integration for Django apps, though the core repository functionality is framework-agnostic.

CHAPTER 1

Contents

1.1 Creating a simple Django app for Fedora Commons repository content

This is a tutorial to walk you through using EULfedora with Django to build a simple interface to the Fedora-Commons repository for uploading files, viewing uploaded files in the repository, editing Dublin Core metadata, and searching content in the repository.

This tutorial assumes that you have an installation of the Fedora Commons repository available to interact with. You should have some familiarity with Python and Django (at the very least, you should have worked through the Django Tutorial). You should also have some familiarity with the Fedora Commons Repository and a basic understanding of objects and content models in Fedora.

We will use pip to install EULfedora and its dependencies; on some platforms (most notably, in Windows), you may need to install some of the python dependencies manually.

1.1.1 Create a new Django project and setup `eulfedora`

Use `pip` to install the *`eulfedora`* library and its dependencies. For this tutorial, we'll use the latest released version:

```
$ pip install eulfedora
```

This command should install EULfedora and its Python dependencies.

We're going to make use of a few items in `eulcommon`, so let's install that now too:

```
$ pip install eulcommon
```

We'll use Django, a popular web framework, for the web components of this tutorial:

```
$ pip install django
```

Now, let's go ahead and create a new Django project. We'll call it *simplerepo*:

```
$ django-admin.py startproject simplerepo
```

Go ahead and do some minimal configuration in your django settings. For simplicity, you can use a sqlite database for this tutorial (in fact, we won't make much use of this database).

In addition to the standard Django settings, add *`eulfedora`* to your `INSTALLED_APPS` and add Fedora connection configurations to your `settings.py` so that the *`eulfedora Repository`* object can automatically connect to your configured Fedora repository:

```
# Fedora Repository settings
FEDORA_ROOT = 'https://localhost:8543/fedora/'
FEDORA_USER = 'fedoraAdmin'
FEDORA_PASSWORD = 'fedoraAdmin'
FEDORA_PIDSPACE = 'simplerepo'
```

Since we're planning to upload content into Fedora, make sure you are using a fedora user account that has permission to upload, ingest, and modify content.

1.1.2 Create a model for your Fedora object

Before we can upload any content, we need to create an object to represent how we want to store that data in Fedora. Let's create a new Django app where we will create this model and associated views:

```
$ python manage.py startapp repo
```

In `repo/models.py`, create a class that extends *DigitalObject*:

```
from eulfedora.models import DigitalObject, FileDatastream

class FileObject(DigitalObject):
    FILE_CONTENT_MODEL = 'info:fedora/genrepo:File-1.0'
    CONTENT_MODELS = [ FILE_CONTENT_MODEL ]
    file = FileDatastream("FILE", "Binary datastream", defaults={
            'versionable': True,
    })
```

What we're doing here extending the default *DigitalObject*, which gives us Dublin Core and RELS-EXT datastream mappings for free, since those are part of every Fedora object. In addition, we're defining a custom datastream that we will use to store the binary files that we're going to upload for ingest into Fedora. This configures a versionable *FileDatastream* with a datastream id of `FILE` and a default datastream label of `Binary datastream`. We could also set a default mimetype here, if we wanted.

Let's inspect our new model object in the Django console for a moment:

```
$ python manage.py shell
```

The easiest way to initialize a new object is to use the Repository object `get_object` method, which can also be used to access existing Fedora objects. Using the Repository object allows us to seamlessly pass along the Fedora connection configuration that the Repository object picks up from your django `settings.py`:

```
>>> from eulfedora.server import Repository
>>> from simplerepo.repo.models import FileObject

# initialize a connection to the configured Fedora repository instance
>>> repo = Repository()

# create a new FileObject instance
>>> obj = repo.get_object(type=FileObject)
# this is an uningested object; it will get the default type of generated pid when we save it
>>> obj
<FileObject (generated pid; uningested)>

# every DigitalObject has Dublin Core
>>> obj.dc
<eulfedora.models.XmlDatastreamObject object at 0xa56f4ec>
# dc.content is where you access and update the actual content of the datastream
>>> obj.dc.content
```

```
<eulxml.xmlmap.dc.DublinCore object at 0xa5681ec>
# print out the content of the DC datastream - nothing there (yet)
>>> print obj.dc.content.serialize(pretty=True)
<oai_dc:dc xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/" xmlns:dc="http://purl.org/dc/el

# every DigitalObject also gets rels_ext for free
>>> obj.rels_ext
<eulfedora.models.RdfDatastreamObject object at 0xa56866c>
# this is an RDF datastream, so the content uses rdflib instead of :mod:`eulxml.xmlmap`
>>> obj.rels_ext.content
<Graph identifier=omYiNhtw0 (<class 'rdflib.graph.Graph'>)>
# print out the content of the rels_ext datastream
# notice that it has a content-model relation defined based on our class definition
>>> print obj.rels_ext.content.serialize(pretty=True)
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:fedora-model="info:fedora/fedora-system:def/model#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="info:fedora/TEMP:DUMMY_PID">
    <fedora-model:hasModel rdf:resource="info:fedora/genrepo:File-1.0"/>
  </rdf:Description>
</rdf:RDF>

# our FileObject also has a custom file datastream, but there's no content yet
>>> obj.file
<eulfedora.models.FileDatastreamObject object at 0xa56ffac>

# save the object to Fedora
>>> obj.save()

# our object now has a pid that was automatically generated by Fedora
>>> obj.pid
'simplerepo:1'
# the object also has information about when it was created, modified, etc
>>> obj.created
datetime.datetime(2011, 3, 16, 19, 22, 46, 317000, tzinfo=tzutc())
>>> print obj.created
2011-03-16 19:22:46.317000+00:00
# datastreams have this kind of information as well
>>> print obj.dc.mimetype
text/xml
>>> print obj.dc.created
2011-03-16 19:22:46.384000+00:00

# we can modify the content and save the changes
>>> obj.dc.content.title = 'My SimpleRepo test object'
>>> obj.save()
```

We've defined a FileObject with a custom content model, but we haven't created the content model object in Fedora yet. For simple content models, we can do this with a custom django manage.py command. Run it in verbose mode so you can more details about what it is doing:

```
$ python manage.py syncrepo -v 2
```

You should see some output indicating that content models were generated for the class you just defined.

This command was is analogous to the Django `syncdb` command. It looks through your models for classes that extend DigitalObject, and when it finds content models defined that it can generate, which don't already exist in the

configured repository, it will generate them and ingest them into Fedora. It can also be used to load initial objects by way of simple XML filters.

1.1.3 Create a view to upload content

So, we have a custom *DigitalObject* defined. Let's do something with it now.

Display an upload form

We haven't defined any url patterns yet, so let's create a `urls.py` for our repo app and hook that into the main project urls. Create `repo/urls.py` with this content:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('simplerepo.repo.views',
    url(r'^upload/$', 'upload', name='upload'),
)
```

Then include that in your project `urls.py`:

```
(r'^', include('simplerepo.repo.urls')),
```

Now, let's define a simple upload form and a view method to correspond to that url. First, for the form, create a file named `repo/forms.py` and add the following:

```
from django import forms

class UploadForm(forms.Form):
    label = forms.CharField(max_length=255, # fedora label maxes out at 255 characters
                help_text='Preliminary title for the new object. 255 characters max.')
    file = forms.FileField()
```

The minimum we need to create a new FileObject in Fedora is a file to ingest and a label for the object in Fedora. We're could actually make the label optional here, because we could use the file name as a preliminary label, but for simplicity let's require it.

Now, define an upload view to use this form. For now, we're just going to display the form on GET; we'll add the form processing in a moment. Edit `repo/views.py` and add:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from simplerepo.repo.forms import UploadForm

def upload(request):
    if request.method == 'GET':
            form = UploadForm()

    return render_to_response('repo/upload.html',
            {'form': form}, context_instance=RequestContext(request))
```

But we still need a template to display our form. Create a template directory and add it to your `TEMPLATE_DIRS` configuration in `settings.py`. Create a `repo` directory inside your template directory, and then create `upload.html` inside that directory and give it this content:

```
<form method="post" enctype="multipart/form-data">{% csrf_token %}
    {{ form.as_p }}
```

```
    <input type="submit" value="Submit"/>
</form>
```

Let's start the django server and make sure everything is working so far. Start the server:

```
$ python manage.py runserver
```

Then load http://localhost:8000/upload/ in your Web browser. You should see a simple upload form with the two fields defined.

Process the upload

Ok, but our view doesn't do anything yet when you submit the web form. Let's add some logic to process the form. We need to import the Repository and FileObject classes and use the posted form data to initialize and save a new object, rather like what we did earlier when we were investigating FileObject in the console. Modify your `repo/views.py` so it looks like this:

```
from django.shortcuts import render_to_response
from django.template import RequestContext

from eulfedora.server import Repository

from simplerepo.repo.forms import UploadForm
from simplerepo.repo.models import FileObject

def upload(request):
    obj = None
    if request.method == 'POST':
        form = UploadForm(request.POST, request.FILES)
        if form.is_valid():
            # initialize a connection to the repository and create a new FileObject
            repo = Repository()
            obj = repo.get_object(type=FileObject)
            # set the file datastream content to use the django UploadedFile object
            obj.file.content = request.FILES['file']
            # use the browser-supplied mimetype for now, even though we know this is unreliable
            obj.file.mimetype = request.FILES['file'].content_type
            # let's store the original file name as the datastream label
            obj.file.label = request.FILES['file'].name
            # set the initial object label from the form as the object label and the dc:title
            obj.label = form.cleaned_data['label']
            obj.dc.content.title = form.cleaned_data['label']
            obj.save()

            # re-init an empty upload form for additional uploads
            form = UploadForm()

    elif request.method == 'GET':
           form = UploadForm()

    return render_to_response('repo/upload.html', {'form': form, 'obj': obj},
        context_instance=RequestContext(request))
```

When content is posted to this view, we're binding our form to the request data and, when the form is valid, creating a new FileObject and initializing it with the label and file that were posted, and saving it. The view is now passing that object to the template, so if it is defined that should mean we've successfully ingested content into Fedora. Let's update our template to show something if that is defined. Add this to `repo/upload.html` before the form is displayed:

```
{% if obj %}
    <p>Successfully ingested <b>{{ obj.label }}</b> as {{ obj.pid }}.</p>
    <hr/>
    {# re-display the form to allow additional uploads #}
    <p>Upload another file?</p>
{% endif %}
```

Go back to the upload page in your web browser. Go ahead and enter a label, select a file, and submit the form. If all goes well, you should see a the message we added to the template for successful ingest, along with the pid of the object you just created.

1.1.4 Display uploaded content

Now we have a way to get content in Fedora, but we don't have any way to get it back out. Let's build a display method that will allow us to view the object and its metadata.

Object display view

Add a new url for a single-object view to your urlpatterns in `repo/urls.py`:

```
url(r'^objects/(?P<pid>[^/]+)/$', 'display', name='display'),
```

Then define a simple view method that takes a pid in `repo/views.py`:

```
def display(request, pid):
    repo = Repository()
    obj = repo.get_object(pid, type=FileObject)
    return render_to_response('repo/display.html', {'obj': obj})
```

For now, we're going to assume the object is the type of object we expect and that we have permission to access it in Fedora; we can add error handling for those cases a bit later.

We still need a template to display something. Create a new file called `repo/display.html` in your templates directory, and then add some code to output some information from the object:

```
<h1>{{ obj.label }}</h1>
<table>
    <tr><th>pid:</th><td> {{ obj.pid }}</td></tr>
    {% with obj.dc.content as dc %}
        <tr><th>title:</th><td>{{ dc.title }}</td></tr>
        <tr><th>creator:</th><td>{{ dc.creator }}</td></tr>
        <tr><th>date:</th><td>{{ dc.date }}</td></tr>
 {% endwith %}
</table>
```

We're just using a simple table layout for now, but of course you can display this object information anyway you like. We're just starting with a few of the Dublin Core fields for now, since most of them don't have any content yet.

Go ahead and take a look at the object you created before using the upload form. If you used the `simplerepo` PIDSPACE configured above, then the the first item you uploaded should now be viewable at http://localhost:8000/objects/simplerepo:1/.

You might notice that we're displaying the text 'None' for creator and date. This is because those fields aren't present at all yet in our object Dublin Core, and `eulxml.xmlmap` fields distinguish between an empty XML field and one that is not-present at all by using the empty string and None respectively. Still, that doesn't look great, so let's adjust our template a little bit:

```
<tr><th>creator:</th><td>{{ dc.creator|default:'' }}</td></tr>
<tr><th>date:</th><td>{{ dc.date|default:'' }}</td></tr>
```

We actually have more information about this object than we're currently displaying, so let's add a few more things to our object display template. The object has information about when it was created and when it was last modified, so let's add a line after the object label:

```
<p>Uploaded at {{ obj.created }}; last modified {{ obj.modified }}.</p>
```

These fields are actually Python datetime objects, so we can use Django template filters to display then a bit more nicely. Try modifying the line we just added:

```
<p>Uploaded at {{ obj.created }}; last modified {{ obj.modified }}
   ({{  obj.modified|timesince }} ago).</p>
```

It's pretty easy to display the Dublin Core datastream content as XML too. This may not be something you'd want to expose to regular users, but it may be helpful as we develop the site. Add a few more lines at the end of your `repo/display.html` template:

```
<hr/>
<pre>{{ obj.dc.content.serialize }}</pre>
```

You could do this with the RELS-EXT just as easily (or basically any XML or RDF datastream), although it may not be as valuable for now, since we're not going to be modifying the RELS-EXST just yet.

So far, we've got information about the object and the Dublin Core displaying, but nothing about the file that we uploaded to create this object. Let's add a bit more to our template:

```
<p>{{ obj.file.label }} ({{ obj.file.info.size|filesizeformat }},
                         {{ obj.file.mimetype }})</p>
```

Remember that in our `upload` view method we set the file datastream label and mimetype based on the file that was uploaded from the web form. Those are stored in Fedora as part of the datastream information, along with some other things that Fedora calculates for us, like the size of the content.

Download File datastream

Now we're displaying information about the file, but we don't actually have a way to get the file back out of Fedora yet. Let's add another view.

Add another line to your url patterns in `repo/urls.py`:

```
url(r'^objects/(?P<pid>[^/]+)/file/$', 'file', name='download'),
```

And then update `repo/views.py` to define the new view method. First, we need to add a new import:

```
from eulfedora.views import raw_datastream
```

`eulfedora.views.raw_datastream()` is a generic view method that can be used for displaying datastream content from fedora objects. In some cases you may be able to use `raw_datastream()` directly (e.g., it might be useful for displaying XML datastreams), but in this case we want to add an extra header to indicate that the content should be downloaded. Add this method to `repo/views.py`:

```
def file(request, pid):
    dsid = 'FILE'
    extra_headers = {
        'Content-Disposition': "attachment; filename=%s.pdf" % pid,
    }
    return raw_datastream(request, pid, dsid, type=FileObject, headers=extra_headers)
```

We've defined a content disposition header so the user will be prompted to save the response with a filename based on the pid do the object in fedora. The `raw_datastream()` method will add a few additional response headers based on the datastream information from Fedora. Let's link this in from our object display page so we can try it out. Edit your `repo/display.html` template and turn the original filename into a link:

```
<a href="{% url download obj.pid %}">{{ obj.file.label }}</a>
```

Now, try it out! You should be able to download the file you originally uploaded.

But, hang on– you may have noticed, there are a couple of details hard-coded in our download view that really shouldn't be. What if the file you uploaded wasn't a PDF? What if we decide we want to use a different datastream ID? Let's revise our view method a bit:

```
def file(request, pid):
    dsid = FileObject.file.id
    repo = Repository()
    obj = repo.get_object(pid, type=FileObject)
    extra_headers = {
        'Content-Disposition': "attachment; filename=%s" % obj.file.label,
    }
    return raw_datastream(request, pid, dsid, type=FileObject, headers=extra_headers)
```

We can get the ID for the file datastream directly from the *FileDatastream* object on our FileObject class. And in our upload view we set the original file name as our datastream label, so we'll go ahead and use that as the download name.

1.1.5 Edit Fedora content

So far, we can get content into Fedora and we can get it back out. Now, how do we modify it? Let's build an edit form & a view that we can use to update the Dublin Core metadata.

XmlObjectForm for Dublin Core

We're going to create an `eulxml.forms.XmlObjectForm` instance for editing `eulxml.xmlmap.dc.DublinCore`. `XmlObjectForm` is roughly analogous to Django's `ModelForm`, except in place of a Django Model we have an `XmlObject` that we want to make editable.

First, add some new imports to `repo/forms.py`:

```
from eulxml.xmlmap.dc import DublinCore
from eulxml.forms import XmlObjectForm
```

Then we can define our new edit form:

```
class DublinCoreEditForm(XmlObjectForm):
    class Meta:
        model = DublinCore
        fields = ['title', 'creator', 'date']
```

We'll start simple, with just the three fields we're currently displaying on our object display page. This code creates a custom `XmlObjectForm` with a *model* of (which for us is an instance of `XmlObject`) `DublinCore`. `XmlObjectForm` knows how to look at the model object and figure out how to generate form fields that correspond to the xml fields. By adding a list of fields, we tell XmlObjectForm to only build form fields for these attributes of our model.

Now we need a view and a template to display our new form. Add another url to `repo/urls.py`:

```
url(r'^objects/(?P<pid>[^/]+)/edit/$', 'edit', name='edit'),
```

And then define the corresponding method in `repo/views.py`. We need to import our new form:

```
from simplerepo.repo.forms import DublinCoreEditForm
```

Then, use it in a view method. For now, we'll just instantiate the form, bind it to our content, and pass it to a template:

```
def edit(request, pid):
    repo = Repository()
    obj = repo.get_object(pid, type=FileObject)
    form = DublinCoreEditForm(instance=obj.dc.content)
    return render_to_response('repo/edit.html', {'form': form, 'obj': obj},
            context_instance=RequestContext(request))
```

We have to instantiate our object, and then pass in the *content* of the DC datastream as the instance to our model. Our XmlObjectForm is using `DublinCore` as its model, and `obj.dc.content` is an instance of DublinCore with data loaded from Fedora.

Create a new file called `repo/edit.html` in your templates directory and add a little bit of code to display the form:

```
<h1>Edit {{ obj.label }}</h1>
<form method="post">{% csrf_token %}
    <table>{{ form.as_table }}</table>
    <input type="submit" value="Save"/>
</form>
```

Load the edit page for that first item you uploaded: http://localhost:8000/objects/simplerepo:1/edit/. You should see a form with the three fields that we listed. Let's modify our view method so it will do something when we submit the form:

```
def edit(request, pid):
    repo = Repository()
    obj = repo.get_object(pid, type=FileObject)
    if request.method == 'POST':
        form = DublinCoreEditForm(request.POST, instance=obj.dc.content)
        if form.is_valid():
            form.update_instance()
            obj.save()
    elif request.method == 'GET':
        form = DublinCoreEditForm(instance=obj.dc.content)
    return render_to_response('repo/edit.html', {'form': form, 'obj': obj},
            context_instance=RequestContext(request))
```

When the data is posted to this view, we're binding our form to the posted data and the XmlObject instance. If it's valid, then we can call the `update_instance()` method, which actually updates the `XmlObject` that is attached to our DC datastream object based on the form data that was posted to the view. When we save the object, the *`DigitalObject`* class detects that the `dc.content` has been modified and will make the necessary API calls to update that content in Fedora.

Note: It may not matter too much in this case, since we are working with simple Dublin Core XML, but it's probably worth noting that the form `is_valid()` check actually includes XML Schema validation on `XmlObject` instances that have a schema defined. In most cases, it should be difficult (if not impossible) to generate invalid XML via an `XmlObjectForm`; but if you edit the XML manually and introduce something that is not schema-valid, you'll see the validation error when you attempt to update that content with `XmlObjectForm`.

Try entering some text in your form and submitting the data. It should update your object in Fedora with the changes you made. However, our interface isn't very user friendly right now. Let's adjust the edit view to redirect the user to the object display after changes are saved.

We'll need some additional imports:

```
from django.core.urlresolvers import reverse
from eulcommon.djangoextras.http import HttpResponseSeeOtherRedirect
```

Note: `HttpResponseSeeOtherRedirect` is a custom subclass of `django.http.HttpResponse` analogous to `HttpResponseRedirect` or `HttpResponsePermanentRedirect`, but it returns a See Other redirect (HTTP status code 303).

After the `object.save()` call in the edit view method, add this:

```
return HttpResponseSeeOtherRedirect(reverse('display', args=[obj.pid]))
```

Now when you make changes to the Dublin Core fields and submit the form, it should redirect you to the object display page and show the changes you just made.

Right now our edit form only has three fields. Let's customize it a bit more. First, let's add all of the Dublin Core fields. Replace the original list of fields in DublinCoreEditForm with this:

```
fields = ['title', 'creator', 'contributor', 'date', 'subject',
    'description', 'relation', 'coverage', 'source', 'publisher',
    'rights', 'language', 'type', 'format', 'identifier']
```

Right now all of those are getting displayed as text inputs, but we might want to treat some of them a bit differently. Let's customize some of the widgets:

```
widgets = {
    'description': forms.Textarea,
    'date': SelectDateWidget,
}
```

You'll also need to add another import line so you can use `SelectDateWidget`:

```
from django.forms.extras.widgets import SelectDateWidget
```

Reload the object edit page in your browser. You should see all of the Dublin Core fields we added, and the custom widgets for description and date. Go ahead and fill in some more fields and save your changes.

While we're adding fields, let's change our display template so that we can see any Dublin Core fields that are present, not just those first three we started with. Replace the title, creator, and date lines in your `repo/display.html` template with this:

```
{% for el in dc.elements %}
    <tr><th>{{ el.name }}:</th><td>{{el}}</td></tr>
{% endfor %}
```

Now when you load the object page in your browser, you should see all of the fields that you entered data for on the edit page.

1.1.6 Search Fedora content

So far, we've just been working with the objects we uploaded, where we know the PID of the object we want to view or edit. But how do we come back and find that again later? Or find other content that someone else created? Let's build a simple search to find objects in Fedora.

Note: For this tutorial, we'll us the Fedora **findObjects** API method. This search is quite limited, and for a production system, you'll probably want to use something more powerful, such as GSearch or Solr, but findObjects is enough to get you started.

The built-in fedora search can either do a keyword search across all indexed fields *or* a fielded search. For the purposes of this tutorial, a simple keyword search will accomplish what we need. Let's create a simple form with one input for keyword search terms. Add the following to `repo/forms.py`:

```
class SearchForm(forms.Form):
    keyword = forms.CharField()
```

Add a search url to `repo/urls.py`:

```
url(r'^search/$', 'search', name='search'),
```

Then import the new form into `repo/views.py` and define the view that will actually do the searching:

```
from simplerepo.repo.forms import SearchForm

def search(request):
    objects = None
    if request.method == 'POST':
        form = SearchForm(request.POST)
        if form.is_valid():
            repo = Repository()
            objects = list(repo.find_objects(form.cleaned_data['keyword'], type=FileObject))

    elif request.method == 'GET':
        form = SearchForm()
    return render_to_response('repo/search.html', {'form': form, 'objects': objects},
            context_instance=RequestContext(request))
```

As before, on a GET request we simple pass the form to the template for display. When the request is a POST with valid search data, we're going to instantiate our *Repository* object and call the *find_objects()* method. Since we're just doing a term search, we can just pass in the keywords from the form. If you wanted to do a fielded search, you could build a keyword-argument style list of fields and search terms instead. We're telling *find_objects()* to return everything it finds as an instance of our `FileObject` class for now, even though that is an over-simplification and in searching across all content in the Fedora repository we may well find other kinds of content.

Let's create a search template to display the search form and search results. Create `repo/search.html` in your templates directory and add this:

```
<h1>Search for objects</h1>
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Submit"/>
</form>
{% if objects %}
    <hr/>
    {% for obj in objects %}
        <p><a href="{% url display obj.pid %}">{{ obj.label }}</a></p>
    {% endfor %}
{% endif %}
```

This template will always display the search form, and if any objects were found, it will list them. Let's take it for a whirl! Go to http://localhost:8000/search/ and enter a search term. Try searching for the object labels, any of the values you entered into the Dublin Core fields that you edited, or if you're using `simplerepo` for your configured `PIDSPACE`, search on `simplerepo:*` to find the objects you've uploaded.

When you are searching across disparate content in the Fedora repository, depending on how you have access configured for that repository, there is a possibility that the search could return an object that the current user doesn't actually have permission to view. For efficiency reasons, the *DigitalObject* postpones any Fedora API calls until the last possibly moment– which means that in our search results, any connection errors will happen in the template instead of in the view method. Fortunately, *eulfedora.templatetags* has a template tag to help with that! Let's rewrite the search template to use it:

```
{% load fedora %}
<h1>Search for objects</h1>
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Submit"/>
</form>
{% if objects %}
    <hr/>
    {% for obj in objects %}
      {% fedora_access %}
        <p><a href="{% url display obj.pid %}">{{ obj.label }}</a></p>
      {% permission_denied %}
        <p>You don't have permission to view this object.</p>
      {% fedora_failed %}
        <p>There was an error accessing fedora.</p>
      {% end_fedora_access %}
    {% endfor %}
{% endif %}
```

What we're doing here is loading the `fedora` template tag library, and then using fedora_access for each object that we want to display. That way we can catch any permission or connection errors and display some kind of message to the user, and still display all the content they have permission to view.

For this template tag to work correctly, you're also going to have disable template debugging (otherwise, the Django template debugging will catch the error first). Edit your `settings.py` and change `TEMPLATE_DEBUG` to False.

1.2 Example Uses

1.2.1 Bulk purging test objects via console

The combination of *eulfedora* and interactive Python or Django's shell provides a simple but powerful console interface to Fedora. For example, if you loaded a bunch of test or demo objects to a test or development Fedora instance and you wanted to remove them, you could purge them with *eulfedora* as follows. This example assumes a django project with Fedora settings configured and *eulfedora* already installed (see *eulfedora.server* for documentation on supported Django settings). First, start up the Django console:

```
$ python manage.py shell
```

Inside the Django shell, import *Repository* and your Django settings to easily initialize a Repository connection to your configured Fedora (in this example, we're accessing the repository that is configured for testing):

```
>>> from eulfedora.server import Repository
>>> from django.conf import settings
>>> repo = Repository(settings.FEDORA_TEST_ROOT, \
...     settings.FEDORA_TEST_USER, settings.FEDORA_TEST_PASSWORD)
>>> for o in repo.find_objects(pid__contains='test-obj:*'):
...     repo.purge_object(o.pid)
...
```

This example will find and purge all objects in the `test-obj` pidspace. Of course, you could easily find objects by ownerId, title text, or any of the other fields supported by *`find_objects()`*.

1.3 `eulfedora` – Python objects to interact with the Fedora Commons repository

1.3.1 `eulfedora.models` - Fedora models

DigitalObject

class `eulfedora.models.`**`DigitalObject`**(*api*, *pid=None*, *create=False*, *default_pidspace=None*)

A single digital object in a Fedora respository, with methods and properties to easy creating, accessing, and updating a Fedora object or any of its component parts, with pre-defined datastream mappings for the standard Fedora Dublin Core (*`dc`*) and RELS-EXT (*`rels_ext`*) datastreams.

Note: If you want idiomatic access to other datastreams, consider extending *`DigitalObject`* and defining your own datastreams using *`XmlDatastream`*, *`RdfDatastream`*, or *`FileDatastream`* as appropriate.

`OWNER_ID_SEPARATOR = ','`

Owner ID separator for multiple owners. Should match the OWNER-ID-SEPARATOR configured in Fedora. For more detail, see https://jira.duraspace.org/browse/FCREPO-82

`add_relationship`(*rel_uri*, *object*)

Add a new relationship to the RELS-EXT for this object. Calls `API_M.addRelationship()`.

Example usage:

```
isMemberOfCollection = 'info:fedora/fedora-system:def/relations-external#isMemberOfCollectic
collection_uri = 'info:fedora/foo:456'
object.add_relationship(isMemberOfCollection, collection_uri)
```

Parameters

- **`rel_uri`** – URI for the new relationship
- **`object`** – related object; can be *`DigitalObject`* or string; if string begins with info:fedora/ it will be treated as a resource, otherwise it will be treated as a literal

Return type boolean

`audit_trail`

Fedora audit trail as an instance of *`eulfedora.xml.AuditTrail`*

Note: Since Fedora (as of 3.5) does not make the audit trail available via an API call or as a datastream, accessing the audit trail requires loading the foxml for the object. If an object has large, versioned XML datastreams this may be slow.

`audit_trail_users`

A set of all usernames recorded in the *`audit_trail`*, if available.

`dc`

`XmlDatastream` for the required Fedora **DC** datastream; datastream content will be automatically loaded as an instance of `eulxml.xmlmap.dc.DublinCore`

default_pidspace = None
Default namespace to use when generating new PIDs in *get_default_pid()* (by default, calls Fedora getNextPid, which will use Fedora-configured namespace if default_pidspace is not set).

ds_list
Dictionary of all datastreams that belong to this object in Fedora. Key is datastream id, value is an `ObjectDatastream` for that datastream.

Only retrieved when requested; cached after first retrieval.

exists

Type bool

True when the object actually exists (and can be accessed by the current user) in Fedora

getDatastreamObject(*dsid*, *dsobj_type=None*)
Get any datastream on this object as a *DatastreamObject* **or** add a new datastream. If the datastream id corresponds to a predefined datastream, the configured object will be returned and the datastream object will be returned. If type is not specified for an existing datastream, attempts to infer the appropriate subclass of datastream object to return based on the mimetype (for XML and RELS-EXT).

Note that if you use this method to add new datastreams you should be sure to set all datastream metadata appropriately for your content (i.e., label, mimetype, control group, etc).

Parameters

- **dsid** – datastream id
- **dsobj_type** – optional *DatastreamObject* type to be returned

getDatastreamProfile(*dsid*)
Get information about a particular datastream belonging to this object.

Parameters **dsid** – datastream id

Return type `DatastreamProfile`

getProfile()
Get information about this object (label, owner, date created, etc.).

Return type `ObjectProfile`

get_default_pid()
Get the next default pid when creating and ingesting a new DigitalObject instance without specifying a pid. By default, calls `ApiFacade.getNextPID()` with the configured class default_pidspace (if specified) as the pid namespace.

If your project requires custom pid logic (e.g., object pids are based on an external pid generator), you should extend DigitalObject and override this method.

get_models()
Get a list of content models the object subscribes to.

get_object(*pid*, *type=None*)
Initialize and return a new *DigitalObject* instance from the same repository, passing along the connection credentials in use by the current object. If type is not specified, the current DigitalObject class will be used.

Parameters

- **pid** – pid of the object to return
- **type** – (optional) *DigitalObject* type to initialize and return

has_model(*model*)
Check if this object subscribes to the specified content model.

Parameters **model** – URI for the content model, as a string (currently only accepted in `info:fedora/foo:###` format)

Return type boolean

has_requisite_content_models

Type bool

True when the current object has the expected content models for whatever subclass of *DigitalObject* it was initialized as.

index_data()
Generate and return a dictionary of default fields to be indexed for searching (e.g., in Solr). Includes top-level object properties, Content Model URIs, and Dublin Core fields.

This method is intended to be customized and extended in order to easily modify the fields that should be indexed for any particular type of object in any project; data returned from this method should be serializable as JSON (the current implementation uses `django.utils.simplejson`).

This method was designed for use with *eulfedora.indexdata*.

index_data_descriptive()
Descriptive data to be included in *index_data()* output. This implementation includes all Dublin Core fields, but should be extended or overridden as appropriate for custom *DigitalObject* classes.

index_data_relations()
Standard Fedora relations to be included in *index_data()* output. This implementation includes all standard relations included in the Fedora relations namespace, but should be extended or overridden as appropriate for custom *DigitalObject* classes.

ingest_user
Username responsible for ingesting this object into the repository, as recorded in the *audit_trail*, if available.

label
object label

label_max_size = 255
maximum label size allowed by fedora

modify_relationship(*rel_uri*, *old_object*, *new_object*)
Modify a relationship from RELS-EXT for this object. As the Fedora API-M does not contain a native "modifyRelationship", this method purges an existing one, then adds a new one, pivoting on the predicate. Calls `API_M.purgeRelationship()`, `API_M.addRelationship()`

Example usage:

```
predicate = 'info:fedora/fedora-system:def/relations-external#isMemberOfCollection'
old_object = 'info:fedora/foo:456'
new_object = 'info:fedora/foo:789'

object.modify_relationship(predicate, old_object, new_object)
```

Parameters

- **rel_uri** – URI for the existing relationship

- **old_object** – previous target object for relationship; can be *DigitalObject* or string; if string begins with info:fedora/ it will be treated as a resource, otherwise it will be treated as a literal
- **new_object** – new target object for relationship; can be *DigitalObject* or string; if string begins with info:fedora/ it will be treated as a resource, otherwise it will be treated as a literal

Return type boolean

object_xml
Fedora object XML as an instance of `FoxmlDigitalObject`. (via `REST_API.getObjectXML()`).

owner
object owner

owner_max_size = 64
maximum owner size allowed by fedora

owners
Read-only list of object owners, separated by the configured *OWNER_ID_SEPARATOR*, with whitespace stripped.

pidspace
Fedora pidspace of this object

purge_relationship(*rel_uri*, *object*)
Purge a relationship from RELS-EXT for this object. Calls `API_M.purgeRelationship()`.

Example usage:

```
isMemberOfCollection = 'info:fedora/fedora-system:def/relations-external#isMemberOfCollectio
collection_uri = 'info:fedora/foo:789'
object.purge_relationship(isMemberOfCollection, collection_uri)
```

Parameters

- **rel_uri** – URI for the existing relationship
- **object** – related object; can be *DigitalObject* or string; if string begins with info:fedora/ it will be treated as a resource, otherwise it will be treated as a literal

Return type boolean

rels_ext
RdfDatastream for the standard Fedora **RELS-EXT** datastream

risearch
Instance of *eulfedora.api.ResourceIndex*, with the same root url and credentials

save(*logMessage=None*)
Save to Fedora any parts of this object that have been modified (including object profile attributes such as *label*, *owner*, or *state*, and any changes to datastream content or datastream properties). If a failure occurs at any point on saving any of the parts of the object, will back out any changes that have been made and raise a *DigitalObjectSaveFailure* with information about where the failure occurred and whether or not it was recoverable.

If the object is new, ingest it. If object profile information has been modified before saving, this data is used in the ingest. Datastreams are initialized to sensible defaults: XML objects are created using their default constructor, and RDF graphs start empty. If they're updated before saving then those updates are

included in the initial version. Datastream profile information is initialized from defaults specified in the *Datastream* declaration, though it too can be overridden prior to the initial save.

state
object state (Active/Inactive/Deleted)

uri
Fedora URI for this object (`info:fedora/foo:###` form of object pid)

uriref
Fedora URI for this object, as an `rdflib.URIRef` URI object

Custom Exception

class `eulfedora.models.`**DigitalObjectSaveFailure**(*pid*, *failure*, *to_be_saved*, *saved*, *cleaned*)
Custom exception class for when a save error occurs part-way through saving an instance of *DigitalObject*. This exception should contain enough information to determine where the save failed, and whether or not any changes saved before the failure were successfully rolled back.

These properties are available:

- obj_pid - pid of the *DigitalObject* instance that failed to save
- failure - string indicating where the failure occurred (either a datastream ID or 'object profile')
- to_be_saved - list of datastreams that were modified and should have been saved
- saved - list of datastreams that were successfully saved before failure occurred
- cleaned - list of saved datastreams that were successfully rolled back
- not_cleaned - saved datastreams that were not rolled back
- recovered - boolean, True indicates all saved datastreams were rolled back

Datastream

Datastream Descriptors

class `eulfedora.models.`**Datastream**(*id*, *label*, *defaults={}*)
Datastream descriptor to simplify configuration and access to datastreams that belong to a particular *DigitalObject*.

When accessed, will initialize a *DatastreamObject* and cache it on the *DigitalObject* that it belongs to.

Example usage:

```
class MyDigitalObject(DigitalObject):
    text = Datastream("TEXT", "Text content", defaults={'mimetype': 'text/plain'})
```

All other configuration defaults are passed on to the *DatastreamObject*.

class `eulfedora.models.`**XmlDatastream**(*id*, *label*, *objtype=None*, *defaults={}*)
XML-specific version of *Datastream*. Datastreams are initialized as instances of *XmlDatastreamObject*. An additional, optional parameter `objtype` is passed to the Datastream object to configure the type of `eulxml.xmlmap.XmlObject` that should be used for datastream content.

Example usage:

```
    from eulxml.xmlmap.dc import DublinCore

    class MyDigitalObject(DigitalObject):
        extra_dc = XmlDatastream("EXTRA_DC", "Dublin Core", DublinCore)

    my_obj = repo.get_object("example:1234", type=MyDigitalObject)
    my_obj.extra_dc.content.title = "Example object"
    my_obj.save(logMessage="automatically setting dc title")
```

class eulfedora.models.**RdfDatastream**(*id*, *label*, *defaults={}*)

RDF-specific version of *Datastream* for accessing datastream content as an rdflib RDF graph. Datastreams are initialized as instances of *RdfDatastreamObject*.

Example usage:

```
    from rdflib import RDFS, Literal

    class MyDigitalObject(DigitalObject):
        extra_rdf = RdfDatastream("EXTRA_RDF", "an RDF graph of stuff")

    my_obj = repo.get_object("example:4321", type=MyDigitalObject)
    my_obj.extra_rdf.content.add((my_obj.uriref, RDFS.comment,
                                  Literal("This is an example object.")))
    my_obj.save(logMessage="automatically setting rdf comment")
```

class eulfedora.models.**FileDatastream**(*id*, *label*, *defaults={}*)

File-based content version of *Datastream*. Datastreams are initialized as instances of *FileDatastreamObject*.

Datastream Objects

class eulfedora.models.**DatastreamObject**(*obj*, *id*, *label*, *mimetype=None*, *versionable=False*, *state='A'*, *format=None*, *control_group='M'*, *checksum=None*, *checksum_type='MD5'*)

Object to ease accessing and updating a datastream belonging to a Fedora object. Handles datastream content as well as datastream profile information. Content and datastream info are only pulled from Fedora when content and info fields are accessed.

Intended to be used with *DigitalObject* and intialized via *Datastream*.

Initialization parameters:

param obj the *DigitalObject* that this datastream belongs to.

param id datastream id

param label default datastream label

param mimetype default datastream mimetype

param versionable default configuration for datastream versioning

param state default configuration for datastream state (default: A [active])

param format default configuration for datastream format URI

param control_group default configuration for datastream control group (default: M [managed])

param checksum default configuration for datastream checksum

param checksum_type default configuration for datastream checksum type (default: MD5)

content
contents of the datastream; for existing datastreams, content is only pulled from Fedora when first requested, and cached after first access; can be used to set or update datastream contents by means of text content or a file-like object. For example, if you have a *DigitalObject* subclass with a *FileDatastream* defined as `image`:

```
with open(filename) as imgfile:
    myobj.image.content = imgfile
```

For an alternate method to set datastream content, see *ds_location*.

ds_location = None
Datastream content location: set this attribute to a URI that Fedora can resolve (e.g., http:// or file://) in order to add or update datastream content from a known, accessible location, rather than posting via *content*. If *ds_location* is set, it takes precedence over *content*.

get_chunked_content(*chunksize=4096*)
Generator that returns the datastream content in chunks, so larger datastreams can be used without reading the entire contents into memory.

history()
Get history/version information for this datastream and return as an instance of *DatastreamHistory*.

isModified()
Check if either the datastream content or profile fields have changed and should be saved to Fedora.

Return type boolean

label
datastream label

mimetype
datastream mimetype

save(*logmessage=None*)
Save datastream content and any changed datastream profile information to Fedora.

Return type boolean for success

size
Size of the datastream content

state
datastream state (Active/Inactive/Deleted)

undo_last_save(*logMessage=None*)
Undo the last change made to the datastream content and profile, effectively reverting to the object state in Fedora as of the specified timestamp.

For a versioned datastream, this will purge the most recent datastream. For an unversioned datastream, this will overwrite the last changes with a cached version of any content and/or info pulled from Fedora.

validate_checksum(*date=None*)
Check if this datastream has a valid checksum in Fedora, by running the `REST_API.compareDatastreamChecksum()` API call. Returns a boolean based on the checksum valid response from Fedora.

Parameters **date** – (optional) check the datastream validity at a particular date/time (e.g., for versionable datastreams)

versionable
boolean; indicates if Fedora is configured to version the datastream

class eulfedora.models.**XmlDatastreamObject**(*obj*, *id*, *label*, *objtype=<class 'eulxml.xmlmap.core.XmlObject'>*, ***kwargs*)
Extends *DatastreamObject* in order to initialize datastream content as an instance of a specified XmlObject.

See *DatastreamObject* for more details. Has one additional parameter:

Parameters **objtype** – xml object type to use for datastream content; if not specified, defaults to XmlObject

class eulfedora.models.**RdfDatastreamObject**(*obj*, *id*, *label*, *mimetype=None*, *versionable=False*, *state='A'*, *format=None*, *control_group='M'*, *checksum=None*, *checksum_type='MD5'*)
Extends *DatastreamObject* in order to initialize datastream content as an rdflib RDF graph.

replace_uri(*src*, *dest*)
Replace a uri reference everywhere it appears in the graph with another one. It could appear as the subject, predicate, or object of a statement, so for each position loop through each statement that uses the reference in that position, remove the old statement, and add the replacement.

class eulfedora.models.**FileDatastreamObject**(*obj*, *id*, *label*, *mimetype=None*, *versionable=False*, *state='A'*, *format=None*, *control_group='M'*, *checksum=None*, *checksum_type='MD5'*)
Extends *DatastreamObject* in order to allow setting and reading datastream content as a file. To update contents, set datastream content property to a new file object. For example:

```
class ImageObject(DigitalObject):
    image = FileDatastream('IMAGE', 'image datastream', defaults={
        'mimetype': 'image/png'
    })
```

Then, with an instance of ImageObject:

```
obj.image.content = open('/path/to/my/file')
obj.save()
```

content
contents of the datastream; only pulled from Fedora when accessed, cached after first access

Relations

class eulfedora.models.**Relation**(*relation*, *type=None*, *ns_prefix={}*, *rdf_type=None*, *related_name=None*)
This descriptor is intended for use with *DigitalObject* RELS-EXT relations, and provides get, set, and delete functionality for a single related *DigitalObject* instance or literal value in the RELS-EXT of an individual object.

Example use for a related object: a *Relation* should be initialized with a predicate URI and optionally a subclass of *DigitalObject* that should be returned:

```
class Page(DigitalObject):
    volume = Relation(relsext.isConstituentOf, type=Volume)
```

When a *Relation* is created with a type that references a *DigitalObject* subclass, a corresponding *ReverseRelation* will automatically be added to the related subclass. For the example above, the fictional `Volume` class would automatically get a `page_set` attribute configured with the same URI and a class of `Page`. Reverse property names can be customized using the `related_name` parameter, which is documented below and follows the basic conventions of Django's `ForeignKey` model field (to which *Relation* is roughly analogous).

Note: Currently, auto-generated *ReverseRelation* properties will always be initialized with `multiple=True`, since that is the most common pattern for Fedora object relations (one to many). Other variants may be added later, if and when use cases arise.

Relation also supports configuring the RDF type and namespace prefixes that should be used for serialization; for example:

```
from rdflib import XSD, URIRef
from rdflib.namespace import Namespace

MYNS = Namespace(URIRef("http://example.com/ns/2011/my-test-namespace/#"))

class MyObj(DigitalObject):
    total = Relation(MYNS.count, ns_prefix={"my": MYNS}, rdf_type=XSD.int)
```

This would allow us to access `total` as an integer on a MyObj object, e.g.:

```
myobj.total = 3
```

and when the RELS-EXT is serialized it will use the configured namespace prefix, e.g.:

```
```

<rdf:RDF xmlns:my="xmlns:fedora-model="info:fedora/fedora-system:def/model#"

xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:about="info:fedora/myobj:1"> <my:count rdf:datatype="http://www.w3.org/2001/XMLSchema#i

</rdf:Description>

</rdf:RDF>

Note: If a namespace prefix is not specified, `rfdlib` will automatically generate a namespace to produce valid output, but it may be less readable than a custom namespace.

Initialization options:

Parameters

- **`relation`** – the RDF predicate URI as a `rdflib.URIRef`
- **`type`** – optional *DigitalObject* subclass to initialize (for object relations); use `type="self"` to specify that the current DigitalObject class should be used (currently no reverse relation will be created for recursive relations).
- **`ns_prefix`** – optional dictionary to configure namespace prefixes to be used for serialization; key should be the desired prefix, value should be an instance of `rdflib.namespace.Namespace`

- **rdf_type** – optional rdf type for literal values (passed to `rdflib.Literal` as the datatype option)
- **related_name** – optional name for the auto-generated *ReverseRelation* property, when the relation is to a subclass of *DigitalObject*; if not specified, the related name will be `classname_set`; a value of + indicates no *ReverseRelation* should be created

class `eulfedora.models.`**ReverseRelation**(*relation*, *type=None*, *multiple=False*)

Descriptor for use with *DigitalObject* RELS-EXT reverse relations, where the owning object is the RDF **object** of the predicate and the related object is the RDF **subject**. This descriptor will query the Fedora *ResourceIndex* for the requested subjects, based on the configured predicate, and return resulting items.

This descriptor *only* provides read access; there is no functionality for setting or deleting reverse-related objects.

It is highly recommended to use *Relation* and let the corresponding *ReverseRelation* be automatically generated for you.

Note: There is currently no support for sorting when multiple items are returned; items are currently returned in whatever order the *ResourceIndex* returns them.

Example use:

```
class Volume(DigitalObject):
    pages = ReverseRelation(relsext.isConstituentOf, type=Page, multiple=True)
```

1.3.2 `eulfedora.xml` - Fedora XML objects (for REST API returns)

Currently, this module consists of `XmlObject` wrappers for the XML returned by the REST API, to simplify dealing with results.

class `eulfedora.xml.`**AuditTrail**(*node=None*, *context=None*, ***kwargs*)

`XmlObject` for the Fedora built-in audit trail that is automatically populated from any modifications made to an object.

records = <eulxml.xmlmap.fields.NodeListField>
list of *AuditTrailRecord* entries

class `eulfedora.xml.`**AuditTrailRecord**(*node=None*, *context=None*, ***kwargs*)

`XmlObject` for a single audit entry in an *AuditTrail*.

action = <eulxml.xmlmap.fields.StringField>
the particular action taken, e.g. *addDatastream*

component = <eulxml.xmlmap.fields.StringField>
the component that was modified, e.g. a datastream ID such as *DC* or *RELS-EXT*

date = <eulfedora.xml.FedoraDateField>
date the change was made, as `datetime.datetime`

id = <eulxml.xmlmap.fields.StringField>
id for this audit trail record

message = <eulxml.xmlmap.fields.StringField>
justification for the change, if any (i.e., log message passed to save method)

process_type = <eulxml.xmlmap.fields.StringField>
type of modification, e.g. *Fedora API-M*

user = <eulxml.xmlmap.fields.StringField>
the user or account responsible for the change (e.g., *fedoraAdmin*)

class eulfedora.xml.**DatastreamHistory**(*node=None*, *context=None*, ***kwargs*)
XmlObject for datastream history information returned by REST_API.getDatastreamHistory().

dsid = <eulxml.xmlmap.fields.StringField>
datastream id

pid = <eulxml.xmlmap.fields.StringField>
pid

versions = <eulxml.xmlmap.fields.NodeListField>
list of *DatastreamProfile* objects for each version

class eulfedora.xml.**DatastreamProfile**(*node=None*, *context=None*, ***kwargs*)
XmlObject for datastream profile information returned by REST_API.getDatastream().

checksum = <eulxml.xmlmap.fields.StringField>
checksum for current datastream contents

checksum_type = <eulxml.xmlmap.fields.StringField>
type of checksum

checksum_valid = <eulxml.xmlmap.fields.SimpleBooleanField>
Boolean flag indicating if the current checksum is valid. Only present when profile is accessed via REST_API.compareDatastreamChecksum()

control_group = <eulxml.xmlmap.fields.StringField>
datastream control group (inline XML, Managed, etc)

created = <eulfedora.xml.FedoraDateField>
date the datastream was created

format = <eulxml.xmlmap.fields.StringField>
format URI for the datastream, if any

label = <eulxml.xmlmap.fields.StringField>
datastream label

mimetype = <eulxml.xmlmap.fields.StringField>
datastream mimetype

size = <eulxml.xmlmap.fields.IntegerField>
integer; size of the datastream content

state = <eulxml.xmlmap.fields.StringField>
datastream state (A/I/D - Active, Inactive, Deleted)

version_id = <eulxml.xmlmap.fields.StringField>
current datastream version id

versionable = <eulxml.xmlmap.fields.SimpleBooleanField>
boolean; indicates whether or not the datastream is currently being versioned

class eulfedora.xml.**DsCompositeModel**(*node=None*, *context=None*, ***kwargs*)
XmlObject for a ContentModel's DS-COMPOSITE-MODEL datastream

class eulfedora.xml.**FedoraDateField**(*xpath*)
Map an XPath expression to a single Python *datetime.datetime*. Assumes date-time format in use by Fedora, e.g. 2010-05-20T18:42:52.766Z

class eulfedora.xml.**FedoraDateListField**(*xpath*)
Map an XPath expression to a list of Python *datetime.datetime*. Assumes date-time format in use by Fedora, e.g. 2010-05-20T18:42:52.766Z. If the XPath expression evaluates to an empty NodeList, evaluates to an empty list.

class eulfedora.xml.**FoxmlDigitalObject**(*node=None*, *context=None*, ***kwargs*)
Minimal XmlObject for Foxml DigitalObject as returned by REST_API.getObjectXML(), to provide access to the Fedora audit trail.

audit_trail = <eulxml.xmlmap.fields.NodeField>
Fedora audit trail, as instance of *AuditTrail*

class eulfedora.xml.**NewPids**(*node=None*, *context=None*, ***kwargs*)
XmlObject for a list of pids as returned by REST_API.getNextPID().

class eulfedora.xml.**ObjectDatastream**(*node=None*, *context=None*, ***kwargs*)
XmlObject for a single datastream as returned by REST_API.listDatastreams()

dsid = <eulxml.xmlmap.fields.StringField>
datastream id - *@dsid*

label = <eulxml.xmlmap.fields.StringField>
datastream label - *@label*

mimeType = <eulxml.xmlmap.fields.StringField>
datastream mime type - *@mimeType*

class eulfedora.xml.**ObjectDatastreams**(*node=None*, *context=None*, ***kwargs*)
XmlObject for the list of a single object's datastreams, as returned by REST_API.listDatastreams()

datastreams = <eulxml.xmlmap.fields.NodeListField>
list of *ObjectDatastream*

pid = <eulxml.xmlmap.fields.StringField>
object pid - *@pid*

class eulfedora.xml.**ObjectHistory**(*node=None*, *context=None*, ***kwargs*)
XmlObject for object history information returned by REST_API.getObjectHistory().

class eulfedora.xml.**ObjectMethodService**(*node=None*, *context=None*, ***kwargs*)
XmlObject for object method services; included in *ObjectMethods* for data returned by REST_API.listMethods().

class eulfedora.xml.**ObjectMethods**(*node=None*, *context=None*, ***kwargs*)
XmlObject for object method information returned by REST_API.listMethods().

class eulfedora.xml.**ObjectProfile**(*node=None*, *context=None*, ***kwargs*)
XmlObject for object profile information returned by REST_API.getObjectProfile().

created = <eulfedora.xml.FedoraDateField>
date the object was created

label = <eulxml.xmlmap.fields.StringField>
object label

modified = <eulfedora.xml.FedoraDateField>
date the object was last modified

owner = <eulxml.xmlmap.fields.StringField>
object owner

state = <eulxml.xmlmap.fields.StringField>
object state (A/I/D - Active, Inactive, Deleted)

class eulfedora.xml.**RepositoryDescription**(*node=None*, *context=None*, ***kwargs*)
XmlObject for a repository description as returned by API_A_LITE.describeRepository()

access_url = <eulxml.xmlmap.fields.StringField>
sample access url

admin_email = <eulxml.xmlmap.fields.StringListField>
administrator emails

base_url = <eulxml.xmlmap.fields.StringField>
base url

name = <eulxml.xmlmap.fields.StringField>
repository name

oai_info = <eulxml.xmlmap.fields.NodeField>
RepositoryDescriptionOAI - configuration info for OAI

oai_url = <eulxml.xmlmap.fields.StringField>
sample OAI url

pid_info = <eulxml.xmlmap.fields.NodeField>
RepositoryDescriptionPid - configuration info for pids

search_url = <eulxml.xmlmap.fields.StringField>
sample search url

version = <eulxml.xmlmap.fields.StringField>
version of Fedora being run

class eulfedora.xml.**RepositoryDescriptionOAI**(*node=None*, *context=None*, ***kwargs*)
XmlObject for OAI section of *RepositoryDescription*

delimiter = <eulxml.xmlmap.fields.StringField>
OAI delimiter

namespace = <eulxml.xmlmap.fields.StringField>
OAI namespace

sample = <eulxml.xmlmap.fields.StringField>
sample OAI id

class eulfedora.xml.**RepositoryDescriptionPid**(*node=None*, *context=None*, ***kwargs*)
XmlObject for PID section of *RepositoryDescription*

delimiter = <eulxml.xmlmap.fields.StringField>
PID delimiter

namespace = <eulxml.xmlmap.fields.StringField>
PID namespace

retain_pids = <eulxml.xmlmap.fields.StringField>
list of pid namespaces configured to be retained

sample = <eulxml.xmlmap.fields.StringField>
sample PID

class eulfedora.xml.**SearchResult**(*node=None*, *context=None*, ***kwargs*)
XmlObject for a single entry in the results returned by REST_API.findObjects()

pid = <eulxml.xmlmap.fields.StringField>
pid

class eulfedora.xml.**SearchResults**(*node=None*, *context=None*, ***kwargs*)
XmlObject for the results returned by REST_API.findObjects()

cursor = <eulxml.xmlmap.fields.IntegerField>
session cursor

expiration_date = <eulxml.xmlmap.fields.DateTimeField>
session experation date

results = <eulxml.xmlmap.fields.NodeListField>
search results - list of *SearchResult*

session_token = <eulxml.xmlmap.fields.StringField>
session token

1.3.3 Server objects

Repository

eulfedora.server.Repository has the capability to automatically use connection configuration parameters pulled from Django settings, when available, but it can also be used without Django.

When you create an instance of *Repository*, if you do not specify connection parameters, it will attempt to initialize the repository connection based on Django settings, using the configuration names documented below.

If you are writing unit tests that use *eulfedora*, you may want to take advantage of eulfedora.testutil.FedoraTestSuiteRunner, which has logic to set up and switch configurations between a development fedora repository and a test repository.

Projects that use this module should include the following settings in their settings.py:

```
# Fedora Repository settings
FEDORA_ROOT = 'http://fedora.host.name:8080/fedora/'
FEDORA_USER = 'user'
FEDORA_PASSWORD = 'password'
FEDORA_PIDSPACE = 'changeme'
FEDORA_TEST_ROOT = 'http://fedora.host.name:8180/fedora/'
FEDORA_TEST_PIDSPACE = 'testme'
```

If username and password are not specified, the Repository instance will be initialized without credentials and access Fedora as an anonymous user. If pidspace is not specified, the Repository will use the default pidspace for the configured Fedora instance.

Projects that need unit test setup and clean-up tasks (syncrepo and test object removal) to access Fedora with different credentials than the configured Fedora credentials should use the following settings:

```
FEDORA_TEST_USER = 'testuser'
FEDORA_TEST_PASSWORD = 'testpassword'
```

---

class eulfedora.server.**Repository**(*root=None*, *username=None*, *password=None*, *request=None*)
Pythonic interface to a single Fedora Commons repository instance.

best_subtype_for_object(*obj*, *content_models=None*)
Given a *DigitalObject*, examine the object to select the most appropriate subclass to instantiate. This generic implementation examines the object's content models and compares them against the defined subclasses of *DigitalObject* to pick the best match. Projects that have a more nuanced understanding of their particular objects should override this method in a *Repository* subclass. This method is intended primarily for use by *infer_object_subtype()*.

Parameters

- **obj** – a *DigitalObject* to inspect
- **content_models** – optional list of content models, if they are known ahead of time (e.g., from a Solr search result), to avoid an additional Fedora look-up

Return type a subclass of *DigitalObject*

default_object_type

Default type to use for methods that return fedora objects - DigitalObject

alias of DigitalObject

find_objects(*terms=None*, *type=None*, *chunksize=None*, ***kwargs*)

Find objects in Fedora. Find query should be generated via keyword args, based on the fields in Fedora documentation. By default, the query uses a contains (~) search for all search terms. Calls ApiFacade.findObjects(). Results seem to return consistently in ascending PID order.

Example usage - search for all objects where the owner contains 'jdoe':

```
repository.find_objects(ownerId='jdoe')
```

Supports all search operators provided by Fedora findObjects query (exact, gt, gte, lt, lte, and contains). To specify the type of query for a particular search term, call find_objects like this:

```
repository.find_objects(ownerId__exact='lskywalker')
repository.find_objects(date__gt='20010302')
```

Parameters

- **type** – type of objects to return; defaults to DigitalObject
- **chunksize** – number of objects to return at a time

Return type generator for list of objects

get_next_pid(*namespace=None*, *count=None*)

Request next available pid or pids from Fedora, optionally in a specified namespace. Calls ApiFacade.getNextPID().

Deprecated since version 0.14: Mint pids for new objects with *eulfedora.models.DigitalObject.get_default_pid()* instead, or call ApiFacade.getNextPID() directly.

Parameters

- **namespace** – (optional) get the next pid in the specified pid namespace; otherwise, Fedora will return the next pid in the configured default namespace.
- **count** – (optional) get the specified number of pids; by default, returns 1 pid

Return type string or list of strings

get_object(*pid=None*, *type=None*, *create=None*)

Initialize a single object from Fedora, or create a new one, with the same Fedora configuration and credentials.

Parameters

- **pid** – pid of the object to request, or a function that can be called to get one. if not specified, *get_next_pid()* will be called if a pid is needed
- **type** – type of object to return; defaults to DigitalObject

Return type single object of the type specified

Create boolean: create a new object? (if not specified, defaults to False when pid is specified, and True when it is not)

get_objects_with_cmodel(*cmodel_uri*, *type=None*)
Find objects in Fedora with the specified content model.

Parameters

- **cmodel_uri** – content model URI (should be full URI in info:fedora/pid:### format)
- **type** – type of object to return (e.g., class:*DigitalObject*)

Return type list of objects

infer_object_subtype(*api*, *pid=None*, *create=False*, *default_pidspace=None*)
Construct a DigitalObject or appropriate subclass, inferring the appropriate subtype using *best_subtype_for_object()*. Note that this method signature has been selected to match the *DigitalObject* constructor so that this method might be passed directly to *get_object()* as a *type*:

```
>>> obj = repo.get_object(pid, type=repo.infer_object_subtype)
```

See also: *TypeInferringRepository*

ingest(*text*, *log_message=None*)
Ingest a new object into Fedora. Returns the pid of the new object on success. Calls `ApiFacade.ingest()`.

Parameters

- **text** – full text content of the object to be ingested
- **log_message** – optional log message

Return type string

purge_object(*pid*, *log_message=None*)
Purge an object from Fedora. Calls `ApiFacade.purgeObject()`.

Parameters

- **pid** – pid of the object to be purged
- **log_message** – optional log message

Return type boolean

risearch
instance of *eulfedora.api.ResourceIndex*, with the same root url and credentials

search_fields = ['pid', 'label', 'state', 'ownerId', 'cDate', 'mDate', 'dcmDate', 'title', 'creator', 'subject', 'description
fields that can be searched against in *find_objects()*

search_fields_aliases = {'owner': 'ownerId', 'dc_modified': 'dcmDate', 'modified': 'mDate', 'created': 'cDate'}
human-readable aliases for oddly-named fedora search fields

class eulfedora.server.**TypeInferringRepository**(*root=None*, *username=None*, *password=None*, *request=None*)
A simple *Repository* subclass whose default object type for *get_object()* is *infer_object_subtype()*. Thus, each call to *get_object()* on a repository such as this will automatically use *best_subtype_for_object()* (or a subclass override) to infer the object's proper type.

default_object_type(*api*, *pid=None*, *create=False*, *default_pidspace=None*)
Construct a DigitalObject or appropriate subclass, inferring the appropriate subtype using `best_subtype_for_object()`. Note that this method signature has been selected to match the *DigitalObject* constructor so that this method might be passed directly to `get_object()` as a *type*:

```
>>> obj = repo.get_object(pid, type=repo.infer_object_subtype)
```

See also: *TypeInferringRepository*

Resource Index

class eulfedora.api.**ResourceIndex**(*opener*)
Python object for accessing Fedora's Resource Index.

RISEARCH_FLUSH_ON_QUERY = False
Specify whether or not RI search queries should specify flush=true to obtain the most recent results. If flush is specified to the query method, that takes precedence.

Irrelevant if Fedora RIsearch is configured with syncUpdates = True.

count_statements(*query*, *language='spo'*, *type='triples'*, *flush=None*)
Run a query in a format supported by the Fedora Resource Index (e.g., SPO or Sparql) and return the count of the results.

Parameters

- **query** – query as a string
- **language** – query language to use; defaults to 'spo'
- **flush** – flush results to get recent changes; defaults to False

Return type integer

find_statements(*query*, *language='spo'*, *type='triples'*, *flush=None*)
Run a query in a format supported by the Fedora Resource Index (e.g., SPO or Sparql) and return the results.

Parameters

- **query** – query as a string
- **language** – query language to use; defaults to 'spo'
- **type** – type of query - tuples or triples; defaults to 'triples'
- **flush** – flush results to get recent changes; defaults to False

Return type `rdflib.ConjunctiveGraph` when type is `triples`; list of dictionaries (keys based on return fields) when type is `tuples`

get_objects(*subject*, *predicate*)
Search for all subjects related to the specified subject and predicate.

Parameters

- **subject** –
- **object** –

Return type generator of RDF statements

get_predicates (*subject*, *object*)
Search for all subjects related to the specified subject and object.

Parameters

- **subject** –
- **object** –

Return type generator of RDF statements

get_subjects (*predicate*, *object*)
Search for all subjects related to the specified predicate and object.

Parameters

- **predicate** –
- **object** –

Return type generator of RDF statements

sparql_query (*query*, *flush=None*)
Run a Sparql query.

Parameters **query** – sparql query string

Return type list of dictionary

spo_search (*subject=None*, *predicate=None*, *object=None*)
Create and run a subject-predicate-object (SPO) search. Any search terms that are not specified will be replaced as a wildcard in the query.

Parameters

- **subject** – optional subject to search
- **predicate** – optional predicate to search
- **object** – optional object to search

Return type `rdflib.ConjunctiveGraph`

spoencode (*val*)
Encode search terms for an SPO query.

Parameters **val** – string to be encoded

Return type string

1.3.4 RDF Namespaces

Predefined RDF namespaces for convenience, for use with *RdfDatastream* objects, in *ResourceIndex* queries, for defining a *eulfedora.models.Relation*, for adding relationships via *eulfedora.models.DigitalObject.add_relationship()*, or anywhere else that Fedora-related `rdflib.term.URIRef` objects might come in handy.

Example usage:

```
from eulfedora.models import DigitalObject, Relation
from eulfedora.rdfns import relsext as relsextns

class Item(DigitalObject):
  collection = Relation(relsextns.isMemberOfCollection)
```

eulfedora.rdfns.**model** = rdf.namespace.ClosedNamespace('info:fedora/fedora-system:def/model#')
rdflib.namespace.ClosedNamespace for the Fedora model namespace (currently only includes hasModel).

eulfedora.rdfns.**oai** = rdf.namespace.ClosedNamespace('http://www.openarchives.org/OAI/2.0/')
rdflib.namespace.ClosedNamespace for the OAI relations commonly used with Fedora and the PROAI OAI provider. Available URIs are: itemID, setSpec, and setName.

eulfedora.rdfns.**relsext** = rdf.namespace.ClosedNamespace('info:fedora/fedora-system:def/relations-external#')
rdflib.namespace.ClosedNamespace for the Fedora external relations ontology.

1.3.5 Django integration

views Fedora views

indexdata Fedora Indexing

Management commands

The following management commands will be available when you include *eulfedora* in your django INSTALLED_APPS and rely on the existdb settings described above.

For more details on these commands, use manage.py <command> help

- **syncrepo** - load simple content models and fixture object to the configured fedora repository

eulfedora Template tags

eulfedora adds custom template tags for use in templates.

fedora_access

Catch fedora failures and permission errors encountered during template rendering:

```
{% load fedora %}

{% fedora_access %}
   <p>Try to access data on fedora objects which could be
     <span class='{{ obj.inaccessible_ds.content.field }}'>inaccessible</span>
     or when fedora is
     <span class='{{ obj.regular_ds.content.other_field }}'>down</span>.</p>
{% permission_denied %}
   <p>Fall back to this content if the main body results in a permission
     error while trying to access fedora data.</p>
{% fedora_failed %}
   <p>Fall back to this content if the main body runs into another error
     while trying to access fedora data.</p>
{% end_fedora_access %}
```

The permission_denied and fedora_failed sections are optional. If only permission_denied is present then non-permission errors will result in the entire block rendering empty. If only fedora_failed is present then that section will be used for all errors whether permission-related or not. If neither is present then all errors will result in the entire block rendering empty.

Note that when Django's `TEMPLATE_DEBUG` setting is on, it precludes all error handling and displays the Django exception screen for all errors, including fedora errors, even if you use this template tag. To disable this Django internal functionality and see the effects of the `fedora_access` tag, add the following to your Django settings:

```
TEMPLATE_DEBUG = False
```

`testutil` Unittest utilities

1.4 Scripts

1.4.1 fedora-checksums

fedora-checksums is a command line utility script to validate or repair datastream checksums for content stored in a Fedora Commons repository.

The script has two basic modes: **validate** and **repair**.

In **validate** mode, the script will iterate through all objects and validate the checksum for each datastream (or optionally each version of any versioned datastream), reporting on invalid or missing checksums.

In **repair** mode, the script will iterate through all objects looking for datastreams with a checksum type of `DISABLED` and a checksum value of `none`; any such datastreams will be updated in Fedora with a new checksum type (either as specified via script argument `--checksum-type` or using the Fedora configured default), prompting Fedora to calculate and save a new checksum.

Running this script in either mode requires passing Fedora connection information and credentials, for example:

```
$ fedora-checksums validate --fedora-root=http://localhost:8080/fedora/
      --fedora-user=fedoraAdmin --fedora-password=fedoraAdmin
```

If you prefer not to specify your fedora password on the command line, specify the `--fedora-password` option with an empty value and you will be prompted:

```
$ fedora-checksums validate --fedora-root=http://localhost:8080/fedora/
      --fedora-user=fedoraAdmin --fedora-password=
```

Note: The fedora user you specify must have permission to find objects, access datastream profiles and history, have permission to run the compareDatastreamChecksum API method (when validating), and permission to modify datastreams (when repairing).

If you have specific objects you wish to check or repair, you can run the script with a list of pids. When validating, there is also an option to output details to a CSV file for further investigation. For more details, see the script usage for the appropriate mode:

```
$ fedora-checksums validate --help
$ fedora-checksums repair --help
```

Note: If the python package `progressbar` is available, progress will be displayed as objects are processed; however, `progressbar` is not required to run this script.

1.4.2 validate-checksums

validate-checksums is a command line utility script intended for regularly, periodically checking that datastream checksums are valid for content stored in a Fedora Commons repository.

When a fixity check is completed, the objects will be updated with a RELS-EXT property indicating the date of the last fixity check, so that objects can be checked again after a specified period.

The default logic is to find and process all objects without any fixity check date in the RELS-EXT (prioritizing objects with the oldest modification dates first, since these are likely to be most at risk), and then to find any objects whose last fixity check was before a specified window (e.g., 30 days).

Because the script needs to run as a privileged fedora user (in order to access and make minor updates to all content), if you are configuring it to run as a cron job or similar, it is recommended to use the options to generate a config file and then load options from that config file when running under cron.

For example, to generate a config file:

```
validate-checksums --generate-config /path/to/config.txt --fedora-password=#####
```

Any arguments passed via the command line will be set in the generated config file; you must pass the password so it can be encrypted in the config file and decrypted for use.

To update a config file from an earlier version of the script:

```
validate-checksums --config /old/config.txt --generate-config /new/config.txt
```

This will preserve all settings in the old config file and generate a new config file with all new settings that are available in the script.

To configure the script to send an email report when invalid or missing checksums are found or when there are any errors saving objects, you can specify email addresses, a from email address, and an smtp server via the command line or a config file.

1.5 Change & Version Information

The following is a summary of changes and improvements to *eulfedora*. New features in each version should be listed, with any necessary information about installation or upgrade notes.

1.5.1 0.23

- Related objects accessed via *Relation* are now cached for efficiency, similar to the way datastreams are cached on *DigitalObject*.
- Methods *purge_relationship()* and *modify_relationship()* added to *DigitalObject*. Contributed by Graham Hukill @ghukill.

1.5.2 0.22.2

- bugfix: correction in detailed output for validate-checksum script when all versions are checked and at least one checksum is invalid

1.5.3 0.22.1

- bugfix: support HTTP Range requests in `eulfedora.views.raw_datastream()` only when explicitly enabled

1.5.4 0.22

- A repository administrator can configure a script to periodically check content checksums in order to identify integrity issues so that they can be dealt with.
- A repository administrator will receive an email notification if the system encounters bad or missing checksums so that they can then resolve any integrity issues.
- A repository admin can view fixity check results for individual objects in the premis data stream (for objects where premis exists) in order to view a more detailed result and the history.
- Support for *basic* HTTP Range requests in `eulfedora.views.raw_datastream()` (e.g., to allow audio/video seek in HTML5 media players)

1.5.5 0.21

- It is now possible to add new datastreams using *`eulfedora.models.DigitalObject.getDatastreamObject()`* (in contrast to predefined datastreams on a subclass of *`DigitalObject`*). Adding new datastreams is supported when ingesting a new object as well as when saving an existing object. This method can also be used to update existing datastreams that are not predefined on a DigitalObject subclass.

1.5.6 0.20

- Development requirements can now be installed as an optional requirement of the eulfedora package (`pip install "eulfedora[dev]"`).
- Unit tests have been updated to use `nose`
- Provides a nose plugin to set up and tear down for a test Fedora Commons repository instance for tests, as an alternative to the custom test runners.

1.5.7 0.19.2

- Bugfix: don't auto-create an XML datastream at ingest when the xml content is empty (i.e., content consists of bootstrapped `xmlmap.XmlObject` only)

1.5.8 0.19.1

- Bugfix: handle Fedora restriction of ownerId field length to 64 characters. When setting *`owner`*, will now warn and truncate the value to allow the object to be saved.

1.5.9 0.19.0

- New command-line script `fedora-checksums` for datastream checksums validation and repair. See Scripts for more details.
- *`DigitalObject`* now provides access to the Fedora built-in audit trail; see *`audit_trail`*. Also provides:

  - `eulfedora.views.raw_audit_trail()`: Django view to serve out audit trail XML, comparable to `eulfedora.views.raw_datastream()`.
  - *`DigitalObject`* attribute *`audit_trail_users`*: set of all usernames listed in the audit trail (i.e., any users who have modified the object)
  - *`DigitalObject`* attribute *`ingest_user`*: username responsible for ingesting the object into Fedora if ingest is listed in the audit trail

- *`Relation`* now supports recursive relations via the option `type="self"`.
- API wrappers have been updated to take advantage of all methods available in the REST API as of Fedora 3.4 which were unavailable in 3.2. This removes the need for any SOAP-based APIs and the dependency on `soaplib`.
- Minor API / unit test updates to support Fedora 3.5 in addition to 3.4.x.

1.5.10 0.18.1

- Bugfix: Default checksum type for *`DatastreamObject`* was previously ignored when creating a new datastream from scratch (e.g., when ingesting a new object). In certain versions of Fedora, this could result in datastreams with missing checksums (checksum type of 'DISABLED', checksum value of 'none').

1.5.11 0.18.0

- Exposed RIsearch `count` return option via *`eulfedora.api.ResourceIndex.count_statements()`*
- *`DatastreamObject`* now supports setting datastream content by URI through the new *`ds_location`* attribute (this is in addition to the previously-available *`content`* attribute).

1.5.12 0.17.0

- Previously, several of the REST API calls in `eulfedora.api.REST_API` suppressed errors and only returned True or False for success or failure; this made it difficult to determine what went wrong when an API call fails. This version of *`eulfedora`* revises that logic so that all methods in `eulfedora.api.REST_API` will raise exceptions when an exception-worthy error occurs (e.g., permission denied, object not found, etc. - anything that returns a 40x or 500 HTTP error response from Fedora). The affected REST methods are:
  - `addDatastream()`
  - `modifyDatastream()`
  - `purgeDatastream()`
  - `modifyObject()`
  - `purgeObject()`
  - `setDatastreamState()`
  - `setDatastreamVersionable()`
- New custom Exception `eulfedora.util.ChecksumMismatch`, which is a subclass of `eulfedora.util.RequestFailed`. This exception will be raised if `addDatastream()` or `modifyDatastream()` is called with a checksum value that Fedora determines to be invalid.

Note: If `addDatastream()` is called with a checksum value but no checksum type, current versions of Fedora ignore the checksum value entirely; in particular, an invalid checksum with no type does not result in a `ChecksumMismatch` exception being raised. You should see a warning if your code attempts to do this.

- Added read-only access to *`DigitalObject`* owners as a list; changed default *`eulfedora.models.DigitalObject.index_data()`* to make owner field a list.
- Modified default *`eulfedora.models.DigitalObject.index_data()`* and sample Solr schema to include a new field (dsids) with a list of datastream IDs available on the indexed object.

1.5.13 0.16.0 - Indexing Support

- Addition of *`eulfedora.indexdata`* to act as a generic webservice that can be used for the creation and updating of indexes such as SOLR; intended to be used with `eulindexer`.

1.5.14 0.15.0 - Initial Release

- Split out fedora-specific components from `eulcore`; now depends on `eulxml`.

1.6 README

EULfedora is a Python module that provides utilities, API wrappers, and classes for interacting with the Fedora-Commons Repository (versions 3.4.x and 3.5) in a pythonic, object-oriented way, with optional Django integration.

eulfedora.api provides complete access to the Fedora API, primarily making use of Fedora's REST API. This low-level interface is wrapped by **eulfedora.server.Repository** and **eulfedora.models.DigitalObject**, which provide a more abstract, object-oriented, and Pythonic way of interacting with a Fedora Repository or with individual objects and datastreams.

eulfedora.indexdata provides a webservice that returns data for fedora objects in JSON form, which can be used in conjunction with a service for updating an index, such as *eulindexer*.

When used with Django, **eulfedora** can pull the Repository connection configuration from Django settings, and provides a custom management command for loading simple content models and fixture objects to the configured repository.

1.6.1 Dependencies

eulfedora currently depends on eulxml, rdflib, python-dateutil, pycrypto, soaplib.

eulfedora can be used without Django, but additional functionality is available when used with Django.

1.6.2 Contact Information

eulfedora was created by the Digital Programs and Systems Software Team of Emory University Libraries.

libsysdev-l@listserv.cc.emory.edu

1.6.3 License

eulfedora is distributed under the Apache 2.0 License.

1.6.4 Development History

For instructions on how to see and interact with the full development history of **eulfedora**, see eulcore-history.

1.6.5 Developer Notes

To install dependencies for your local check out of the code, run `pip install` in the `eulfedora` directory (the use of virtualenv is recommended):

```
pip install -e .
```

If you want to run unit tests or build sphinx documentation, you will also need to install development dependencies:

```
pip install -e . "eulfedora[dev]"
```

Running the unit tests requires a Fedora Commons repository instance. Before running tests, you will need to copy `test/localsettings.py.dist` to `test/localsettings.py` and edit the configuration for your test repository.

To run all unit tests:

```
nosetests test # for normal development
nosetests test --with-coverage --cover-package=eulfedora --cover-xml --with-xunit   # for continuous
```

To run unit tests for a specific module or class, use syntax like this:

```
nosetests test.test_fedora.test_api
nosetests test.test_fedora:TestDigitalObject
```

To generate sphinx documentation:

```
cd doc
make html
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

e

i

m

r

s

t

x

A

B

C

D

E