fragapy Documentation

Release 1.0

November 09, 2011

CONTENTS

1 Adminhelp **3**

2 Amazon **5**
- 2.1 AWS branded scripts . . . 5
- 2.2 SES SMTP relay . . . 5
- 2.3 SMTP relay daemon . . . 6

3 Authentication/Authorization utilities **7**
- 3.1 PermissionDeniedMiddleware . . . 7

4 Common utitlities **9**
- 4.1 Current site, STATIC_URL context processors . . . 9
- 4.2 Middleware for limiting access, shortening HTML responses . . . 9
- 4.3 {% macro %} templatetag . . . 9

5 Ella utilities **11**
- 5.1 Multiple article contents . . . 11
- 5.2 Send email about Publishable . . . 13
- 5.3 Utilities for markup customization . . . 13
- 5.4 JavaScript HitCounts . . . 14

6 Printable output to ODT files (OpenOffice format) **15**
- 6.1 Installation . . . 15
- 6.2 OdtPrintable class . . . 15

7 Soft deletable models **17**

8 Static sitemaps **19**
- 8.1 Installation . . . 19
- 8.2 CRON job to generate sitemaps . . . 20

9 Unix tools **21**

10 Versioning **23**

11 Indices and tables **25**

Fragapy is shared library made and sustained by people from Fragaria, s.r.o. to aviod reinventing the wheel for things that have been written before.

The library has no precise aim so you should think about it as heap of things that proven to be useful and so someone has put them here.

Contents:

CHAPTER

ONE

ADMINHELP

Adminhelp is an application that adds help support for Django admin.

CHAPTER

TWO

AMAZON

Amazon is package of utilities to work with Amazon Web Services (AWS). It is composed of serveral separate tools:

- **AWS branded scripts** - bundled only for easier access as their documentation is somehow confusing
- **SES SMTP relay** - implementation of SMTP server that forwards incoming mail messages to Amazon SES (simple e-mail service) via *Boto* library
- **SMTP relay daemon** - Python executable which will daemonize when executed. Works as support for SMTP relay to use it as service on servers.

2.1 AWS branded scripts

Scripts are written in Perl and require following .deb packages to be installed:

- libxml-libxml-perl
- libssl-dev

Most of the scripts also require you to have valid Amazon AWS credentials. It's easier to put them in the file in your home directory. Example content of the file goes like this:

```
AWSAccessKeyId=YOUR_KEY_ID
AWSSecretKey=YOUR_SECRET_KEY
```

These consist of following:

`ses-verify-email-address.pl`

Use to verify e-mail addresses with Amazon SES. Example usage:

```
./ses-verify-email-address.pl -k ~/.amazon_aws.cred -v example@email.com
```

2.2 SES SMTP relay

class **SESRelaySMTPServer** (*smtpd.SMTPServer*)

This is very simple Python implementation of SMTP server. The only thing it does is to forward the messages to *Boto* library, which is a Python interface for Amazon Web Services.

__init__ (*localaddr*, *remoteaddr*, *aws_key*, *aws_secret*)

Parameters

- **localaddr** – same as in `smtp.SMTPServer`

- **remoteaddr** – same as in `smtp.SMTPServer`
- **aws_key** – ID key to your AWS account
- **aws_secret** – Secret key to your AWS account

2.3 SMTP relay daemon

Daemon can be used to run SMTP relay server as a service in unix-like environments. It expects Debian distribution though.

Steps to run the service are following:

1. Install fragapy library from sources repository. Install boto library.
2. Make symlink of smtpreldaemon.py to /usr/sbin/smtpreldaemon
3. Copy smtp_relay.sh to /etc/init.d
4. Create user amazon.
5. Create /home/amazon/.smtpreldaemon.cfg and edit configuration. You can take smtpreldaemon.cfg as template.
6. Set permissions to chosen config files to amazon user.
7. Install daemon: update-rc.d smtp_relay.sh defaults

Last step ensures that the `smtp_relay.sh` will be run on startup of the server automatically so you don't need to care about it.

CHAPTER

THREE

AUTHENTICATION/AUTHORIZATION UTILITIES

3.1 PermissionDeniedMiddleware

Django doesn't have build-in support for handling `PermissionDenied` exceptions.

Here is one.

To use it, optionally create `403.html` in your templates and add `fragapy.auth.middlware.PermissionDeniedMiddlewa`
to your `MIDDLEWARE_CLASSES`.

CHAPTER

FOUR

COMMON UTITLITIES

4.1 Current site, STATIC_URL context processors

4.2 Middleware for limiting access, shortening HTML responses

4.3 {% macro %} templatetag

CHAPTER

FIVE

PRINTABLE OUTPUT TO ODT FILES (OPENOFFICE FORMAT)

There are many times one wants to provide nice printable output to user while trying to keep it as simple as possible to the programmer.

We have found one easy way to do this - **ODT templates** for OpenOffice!

These templates have strong advantage of being nothing else than **zipped XML file** which allows use to work with them quite easily. In fact, you can do almost everything just by opening them in OpenOffice or LibreOffice and edit them **like they were normal Django templates**.

Sometimes, you would want to do more complicated things and in those cases, you will have to edit XML file directly but still, it's much more simple than create directives to export the files as PDFs.

To make this even simpler, we have created some tools to work with ODT files.

5.1 Installation

Only thing you have to do is add `fragapy.odt` to your `INSTALLED_APPS` and set `ODT_DIR` to hold path where to look for your ODT files.

5.2 OdtPrintable class

The easiest way to add print support to your model is to subclass `OdtPrintable`. Just subclassing this method and creating one of following templates:

- `[app_label]/[object_name]/object_detail.odt`
- `[app_label]/object_detail.odt`
- `object_detail.odt`

does the trick and you can now have your view like this:

```
def example_view(request, object_id):
    object = MyOdtPrintable.objects.get(pk=object_id)
    return object.print_to_response()
```

Which results in ODT being returned as response.

class `OdtPrintable`

Allows for printing models as ODTs.

`get_file_name`(*self*)

Returns file name of resulting ODT. Defaults to slugified `unicode` called on `self`.

`get_template`(*self*)

Returns iterable of relative paths to look for template to render.

`get_template_path`(*self*)

Returns full path to the first template returned by `get_template()` or *None* if no template has been found.

`get_context`(*self*)

Returns context that will be available in ODT template when rendering.

Defaults to `{'object': self}`

`print_to_response`(*self*, ***kwargs*)

Returns `HttpResponse` containing the ODT file.

CHAPTER

SIX

SOFT DELETABLE MODELS

There is a lot of occassions when you want some record not permanently removed when user click "Delete" but just made inactive. This can be for a lot of reasons (to keep history of old orders in e-shop etc.).

Soft deletable models package gives you Django model subclass implementing this repeting task. This package is very simple and consist only of two classes.

class **SoftDeletableModel** (*models.Model*)

`SoftDeletableModel` is an abstract Django model subclass.

The `SoftDeletableModel` suclasses receive one extra attribute, `active` that signals if the record has been deleted or not. It also defines default model manager class: `SoftDeletableModelManager`. If you need to use custom manager for your `SoftDeletableModel` subclass, be sure that your manager subclasses `SoftDeletableModelManager` too.

When delete is called on subclass, nothing bad really happens, only the `active` flag is updated to False.

class **SoftDeletableModelManager** (*models.Manager*)

This is Django model manager subclass that filters out deleted records by default.

If you want your deleted records in your queryset, use `with_deleted()`.

with_deleted (*self*)

Adds deleted records to the queryset.

CHAPTER

SEVEN

STATIC SITEMAPS

Static sitemaps is small pluggable Django application that wraps around the `django.contrib.sitemaps` framework. It's basic purpose is to enable serving of sitemap files through your webserver and not Django application itself.

This has reasonable performance advantages.

This app also gives you **Django management command** which will generate sitemap files along with sitemap index file to your `MEDIA_URL` base path.

7.1 Installation

First, it is necessary to add the application to your `INSTALLED_APPS` as usual:

```
INSTALLED_APPS = (
    ...
    'fragapy.static_sitemaps',
    ...
)
```

Second, you need to add one URL that will serve the Sitemap index file (sitemap.xml). It goes like this:

```
urlpatterns = patterns('',
    url(r'^sitemap.xml', include('fragapy.static_sitemaps.urls')),
)
```

This will ensure that /sitemap.xml will be handled by static_sitemaps view. This step can be avoided if you serve your media files from the same domain as your Django aplication lives on. In that case, you already have /sitemap.xml available.

If that's not the case (e.g. you media files live on different domain than the application itself, like a subdomain or so) you should have this URL set up so that /sitemap.xml is still available.

Next step is some small configuration. Put these into your settings file. There are several configuration variables available:

STATICSITEMAPS_ROOT_SITEMAP This needs to point to the sitemap info dict which is used in common sitemaps framework, e.g.:

```
from iw.charts.sitemaps import sitemaps as charts_sitemaps
from iw.iwapp.sitemaps import sitemaps as iwapp_sitemaps

sitemaps = {}
```

```
sitemaps.update(iwapp_sitemaps)
sitemaps.update(charts_sitemaps)
```

In this example, we want to point our `STATICSITEMAPS_ROOT_SITEMAP` to the `sitemaps` variable, e.g.:

```
STATICSITEMAPS_ROOT_SITEMAP = 'iw.sitemaps.sitemaps'
```

The infodict contents are supposed to be same as for common sitemaps framework. We simply use the same classes.

`STATICSITEMAPS_SITEMAP_DOMAIN` This variable needs to be set to the domain on which the sitemaps files will be held. If you serve your media files from `http://media.web.com`, this should be set to this URL.

`STATICSITEMAPS_USE_GZIP` This defaults to `True`. Keep this unchanged if you want your sitemap files to be gzipped which is definitely beneficial and it is allowed in docs on sitemaps.org.

`STATICSITEMAPS_FILENAME_TEMPLATE` There is usually no need to change this.

7.2 CRON job to generate sitemaps

To generate sitemap files, we recommend using CRON job. There is a management command that will refresh the sitemap files, usage is very simple:

```
django-admin.py refresh_sitemap
```

After it finishes, it also **pings google** so that it will know that sitemap has been updated.

CHAPTER

EIGHT

UNIX TOOLS

Unix tools package contains handy scripts to make your Unix Python programs fancier.

CHAPTER

NINE

VERSIONING

The `fragapy.versioning` is simple package with tools to provide versioning to you Django applications.

Most of utils which follows depend on one simple configuration variable expected in your Django settings file. Example follows:

```
VERSION = (1, 0, 0)
```

`fragapy.versioning.context_processors.version` This is a Django context processor that will add `VERSION` variable in your template context. It can be feasible when you need to show application version in your web page footer or so. Configuration is simple, just add *fragarpy.versioning.context_processors.version* to your *TEMPLATE_CONTEXT_PROCESSORS* tuple in Django settings.

`fragapy.versioning.utils.get_version` This function returns the version tuple (e.g. *(1, 0, 0)*).

`fragapy.versioning.format.format_version` Simple function that formats version tuple in human sensible way.

Results might be `1.0.0.` or `1.0` depending on the lenght of version tuple given.

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*