django-filer Documentation

Release 1.3.0.dev

Stefan Foulis

Oct 06, 2017

Contents

Warning: This documentation refers to the development version of `django-filer`.

As this version has not been released yet, any part of the API maybe subject to modifications without notice, and this documentation may be outdated and not in sync with the code.

`django-filer` is a file management application for django. It handles uploading and organizing files and images in contrib.admin.

Filer detail view:

Filer picker widget:

Custom model fields are provided for use in 3rd party apps as a replacement for the default `FileField` from django. Behind the scenes a `ForeignKey` to the File model is used.

CHAPTER 1

Getting help

- google group: http://groups.google.com/group/django-filer
- IRC: #django-filer on freenode.net

CHAPTER 2

Contributing

The code is hosted on github at http://github.com/divio/django-filer/ and is fully open source. We hope you choose to help us on the project! More information on how to contribute can be found in contributing.

CHAPTER 3

Contents

Installation and Configuration

Getting the latest release

The easiest way to get `django-filer` is simply install it with pip:

```
$ pip install django-filer
```

Dependencies

- Django >= 1.8
- django-mptt >=0.6
- easy_thumbnails >= 2.0
- django-polymorphic >= 0.7
- Pillow >=2.3.0 (with JPEG and ZLIB support, PIL may work but is not supported)

`django.contrib.staticfiles` is required.

Please make sure you install Pillow with JPEG and ZLIB support installed; for further information on Pillow installation and its binary dependencies, check Pillow doc.

Configuration

Add `"filer"` and related apps to your project's `INSTALLED_APPS` setting and run `manage.py migrate`:

```
INSTALLED_APPS = [
    ...
    'easy_thumbnails',
```

```
    'filer',
    'mptt',
    ...
]
```

Note that easy_thumbnails also has database tables and needs a `python manage.py migrate`.

For easy_thumbnails to support retina displays (recent MacBooks, iOS) add to settings.py:

```
THUMBNAIL_HIGH_RESOLUTION = True
```

If you forget this, you may not see thumbnails for your uploaded files. Adding this line and refreshing the admin page will create the missing thumbnails.

Static media

django-filer javascript and css files are managed by `django.contrib.staticfiles`; please see staticfiles documentation to know how to deploy filer static files in your environment.

Subject location aware cropping

It is possible to define the *important* part of an image (the *subject location*) in the admin interface for django-filer images. This is very useful when later resizing and cropping images with easy_thumbnails. The image can then be cropped automatically in a way, that the important part of the image is always visible.

To enable automatic subject location aware cropping of images replace `easy_thumbnails.processors.scale_and_crop` with `filer.thumbnail_processors.scale_and_crop_with_subject_location` in the `THUMBNAIL_PROCESSORS` setting:

```
THUMBNAIL_PROCESSORS = (
    'easy_thumbnails.processors.colorspace',
    'easy_thumbnails.processors.autocrop',
    #'easy_thumbnails.processors.scale_and_crop',
    'filer.thumbnail_processors.scale_and_crop_with_subject_location',
    'easy_thumbnails.processors.filters',
)
```

To crop an image and respect the subject location:

```
{% load thumbnail %}
{% thumbnail obj.img 200x300 crop upscale subject_location=obj.img.subject_location %}
```

Permissions

> **Warning:** File permissions are an experimental feature. The api may change at any time.

See *Permissions* section.

Secure downloads

Warning: File download permissions are an experimental feature. The api may change at any time.

See *Secure Downloads* section.

Canonical URLs

You can configure your project to generate canonical URLs for your public files. Just include django-filer's URLConf in your project's `urls.py`:

```
urlpatterns = [
    ...
    url(r'^filer/', include('filer.urls')),
    ...
]
```

Contrary to the file's actual URL, the canonical URL does not change if you upload a new version of the file. Thus, you can safely share the canonical URL. As long as the file exists, people will be redirected to its latest version.

The canonical URL is displayed in the "advanced" panel on the file's admin page. It has the form:

```
/filer/canonical/1442488644/12/
```

The "filer" part of the URL is configured in the project's URLconf as described above. The "canonical" part can be changed with the setting `FILER_CANONICAL_URL`, which defaults to `'canonical/'`. Example:

```
# settings.py

FILER_CANONICAL_URL = 'sharing/'
```

Debugging and logging

While by default `django-filer` usually silently skips icon/thumbnail generation errors, two options are provided to help when working with `django-filer`:

- `FILER_DEBUG`: Boolean, controls whether bubbling up any `easy-thumbnails` exception (typically if an image file doesn't exists); is `False` by default;
- `FILER_ENABLE_LOGGING`: Boolean, controls whether logging the above exceptions. It requires proper django logging configuration for default logger or `filer` logger. Please see https://docs.djangoproject.com/en/dev/topics/logging/ for further information about Django's logging configuration.

Upgrading

Usually upgrade procedure is straightforward: update the package and run migrations. Some versions require special attention from the developer and here we provide upgrade instructions for such cases.

from 0.9.1 to 0.9.2

From 0.9.2 `File.name` field is `null=False`.

Warning: Data migration in 0.9.2 changes existing null values to empty string.

from 0.8.7 to 0.9

0.9 introduces real separation of private and public files through multiple storage backends. Public files are placed inside `MEDIA_ROOT`, using Djangos default file storage. Private files are now placed in their own location. Unfortunatly the default settings in django-filer 0.8.x made all new uploads "private", but still placed them inside `MEDIA_ROOT` in a subfolder called `filer_private`. In most cases these files are actually meant to be public, so they should be moved.

Note: **Quick and Dirty**: set `FILER_0_8_COMPATIBILITY_MODE=True`. It will pick up the old style settings and configure storage backends the way they were in 0.8. This setting is only meant to easy migration and is not intended to be used long-term.

Manually (SQL)

faster for many large files

Fire up the sql-console and change `is_public` to `True` on all files in the `filer_file` table (`UPDATE filer_file SET is_public=1 WHERE is_public=0;`). The files will still be in `MEDIA_ROOT/filer_private/`, but serving them should already work. Then you can move the files into `filer_private` in the filesystem and update the corresponding paths in the database.

Automatic (Django)

Have filer move all files between storages. This might take a while, since django will read each file into memory and write it to the new location. Especially if you are using an external storage backend such as *Amazon S3*, this might not be an option. Set `FILER_0_8_COMPATIBILITY_MODE=True` and make sure you can access public and private files. Then run this snippet in the django shell:

```
from filer.models import File
import sys
for f in File.objects.filter(is_public=False):
    sys.stdout.write(u'moving %s to public storage... ' % f.id)
    f.is_public = True
    f.save()
    sys.stdout.write(u'done\n')
```

After running the script you can delete the `FILER_0_8_COMPATIBILITY_MODE` setting. If you want to use secure downloads see *Secure Downloads*.

from pre-0.9a3 develop to 0.9

In develop pre-0.9a3 file path was written in the database as relative path inside *filer* directory; since 0.9a3 this is no longer the case so file must be migrate to the new paths. Same disclaimer as 0.8x migration applies: SQL migration is

much faster for large datasets.

Manually (SQL)

Use whatever tool to access you database console and insert the correct directory name at the start of the *file* field. Example:

```
UPDATE filer_file SET file= 'filer_public/' || file WHERE file LIKE '20%' AND is_
↪public=True;
UPDATE filer_file SET file= 'filer_private/' || file WHERE file LIKE '20%' AND is_
↪public=False;
```

Then you will have to move by hand the files from the *MEDIA_ROOT/filer* directory to the new public and private storage directories

Automatic (Django)

Make sure the console user can access/write public and private files. Please note that the *"filer/"* string below should be modified if your files are not saved in *MEDIA_ROOT/filer* Then run this snippet in the django shell:

```
from filer.models import File
import sys
for f in File.objects.filter(is_public=True):
    sys.stdout.write(u'moving %s to public storage... ' % f.id)
    f.is_public = False
    f.file.name = "filer/%s" % f.file.name
    f.save()
    f.is_public = True
    f.save()
    sys.stdout.write(u'done\n')
for f in File.objects.filter(is_public=False):
    sys.stdout.write(u'moving %s to private storage... ' % f.id)
    f.is_public = True
    f.file.name = "filer/%s" % f.file.name
    f.save()
    f.is_public = False
    f.save()
    sys.stdout.write(u'done\n')
```

Double access modification is needed to enabled automatic file move.

Usage

`django-filer` provides model fields to replace *djangos* own django.db.models.FileField and django.db.models.ImageField. The *django-filer* versions provide the added benefit of being able to manage the files independently of where they are actually used in your content. As such the same file can be used in multiple places without re-uploading it multiple times and wasting bandwidth, time and storage.

It also comes with additional tools to detect file duplicates based on SHA1 checksums.

Note: behind the scenes this field is actually just a ForeignKey to the File model in `django-filer`. So you can easily access the extra metadata like this:

```
company.disclaimer.sha1
company.disclaimer.size
company.logo.width
company.logo.height
company.logo.icons['64'] # or {{ company.logo.icons.64 }} in a template
company.logo.url         # prints path to original image
```

FilerFileField and FilerImageField

They are subclasses of django.db.models.ForeignKey, so the same rules apply. The only difference is, that there is no need to declare what model we are referencing (it is always `filer.models.File` for the `FilerFileField` and `filer.models.Image` for the `FilerImageField`).

Simple example `models.py`:

```
from django.db import models
from filer.fields.image import FilerImageField
from filer.fields.file import FilerFileField

class Company(models.Model):
    name = models.CharField(max_length=255)
    logo = FilerImageField(null=True, blank=True,
                           related_name="logo_company")
    disclaimer = FilerFileField(null=True, blank=True,
                                related_name="disclaimer_company")
```

multiple file fields on the same model:

```
from django.db import models
from filer.fields.image import FilerImageField

class Book(models.Model):
    title = models.CharField(max_length=255)
    cover = FilerImageField(related_name="book_covers")
    back = FilerImageField(related_name="book_backs")
```

As with django.db.models.ForeignKey in general, you have to define a non-clashing `related_name` if there are multiple `ForeignKey` s to the same model.

templates

`django-filer` plays well with easy_thumbnails . At the template level a `FilerImageField` can be used the same as a regular django.db.models.ImageField:

```
{% load thumbnail %}
{% thumbnail company.logo 250x250 crop %}
```

admin

The default widget provides a popup file selector that also directly supports uploading new images.

- Clicking on the magnifying glass will display the file selction popup.

- The red X will de-select the currently selected file (usefull if the field can be `null`).

Warning: Don't place a `FilerFileField` as the first field in admin. Django admin will try to set the focus to the first field in the form. But since the form field of `FilerFileField` is hidden that will cause in a javascript error.

Permissions

Warning: File download permissions are an experimental feature. The api may change at any time.

By default files can be uploaded and managed by all staff members based on the standard django model permissions.

Activating permission checking with the `FILER_ENABLE_PERMISSIONS` setting enables fine grained permissions based on individual folders. Permissions can be set in the "Folder permissions" section in Django admin.

Note: These permissions only concern editing files and folders in Django admin. All the files are still world downloadable by anyone who guesses the url. For real permission checks on downloads see the *Secure Downloads* section.

Secure Downloads

Warning: Secure downloads are experimental and the API may change at any time.

Warning: Server Backends currently only work with files in the local filesystem.

Note: For the impatient:

- set `FILER_ENABLE_PERMISSIONS` to `True`
- include `filer.server.urls` in the root `urls.py` without a prefix

To be able to check permissions on the file downloads, a special view is used. The files are saved in a separate location outside of `MEDIA_ROOT` to prevent accidental serving. By default this is a directory called `smedia` that is located in the parent directory of `MEDIA_ROOT`. The smedia directory must **NOT** be served by the webserver directly, because that would bypass the permission checks.

To hook up the view `filer.server.urls` needs to be included in the root `urls.py`:

```
urlpatterns += [
    url(r'^', include('filer.server.urls')),
]
```

Files with restricted permissions need to be placed in a secure storage backend. Configure a secure storage backend in *FILER_STORAGES* or use the default.

Warning: The "Permissions disabled" checkbox in the file detail view in Django admin controls in which storage backend the file is saved. In order for it to be protected, this field must not be checked.

For images the permissions also extend to all generated thumbnails.

By default files with permissions are served directly by the Django process (using the `filer.server.backends.default.DefaultServer` backend). That is acceptable in a development environment, but is very bad for performance and security in production.

The private file view will serve the permission-checked media files by delegating to one of its server backends. The ones bundled with django-filer live in `filer.server.backends` and it is easy to create new ones.

The default is `filer.server.backends.default.DefaultServer`. It is suitable for development and serves the file directly from django.

More suitiable for production are server backends that delegate the actual file serving to an upstream webserver.

NginxXAccelRedirectServer

location: `filer.server.backends.nginx.NginxXAccelRedirectServer`

nginx docs about this stuff: http://wiki.nginx.org/XSendfile

in `settings.py`:

```
FILER_SERVERS = {
    'private': {
        'main': {
            'ENGINE': 'filer.server.backends.nginx.NginxXAccelRedirectServer',
            'OPTIONS': {
                'location': '/path/to/smedia/filer',
                'nginx_location': '/nginx_filer_private',
            },
        },
        'thumbnails': {
            'ENGINE': 'filer.server.backends.nginx.NginxXAccelRedirectServer',
            'OPTIONS': {
                'location': '/path/to/smedia/filer_thumbnails',
                'nginx_location': '/nginx_filer_private_thumbnails',
            },
        },
    },
}
```

`nginx_location` is the location directive where nginx "hides" permission-checked files from general access. A fitting nginx configuration might look something like this:

```
location /nginx_filer_private/ {
  internal;
```

```
  alias /path/to/smedia/filer_private/;
}
location /nginx_filer_private_thumbnails/ {
  internal;
  alias /path/to/smedia/filer_private_thumbnails/;
}
```

Note: make sure you follow the example exactly. Missing trailing slashes and `alias` vs. `root` have subtle differences that can make your config fail.

`NginxXAccelRedirectServer` will add the a `X-Accel-Redirect` header to the response instead of actually loading and delivering the file itself. The value in the header will be something like `/nginx_filer_private/2011/03/04/myfile.pdf`. Nginx picks this up and does the actual file delivery while the django backend is free to do other stuff again.

ApacheXSendfileServer

location: `filer.server.backends.xsendfile.ApacheXSendfileServer`

Warning: I have not tested this myself. Any feedback and example configurations are very welcome :-)

Once you have `mod_xsendfile` installed on your apache server you can configure the settings.

in `settings.py`:

```
FILER_SERVERS = {
    'private': {
        'main': {
            'ENGINE': 'filer.server.backends.xsendfile.ApacheXSendfileServer',
            },
        'thumbnails': {
            'ENGINE': 'filer.server.backends.xsendfile.ApacheXSendfileServer',
            },
        },
    }
```

in your apache configuration:

```
XSendFile On
XSendFilePath /path/to/smedia/
```

`XSendFilePath` is a whitelist for directories where apache will serve files from.

Settings

FILER_ENABLE_PERMISSIONS

Activate the or not the Permission check on the files and folders before displaying them in the admin. When set to `False` it gives all the authorization to staff members based on standard Django model permissions.

Defaults to `False`

FILER_IS_PUBLIC_DEFAULT

Should newly uploaded files have permission checking disabled (be public) by default.

Defaults to `True` (new files have permission checking disable, are public)

FILER_STORAGES

A dictionary to configure storage backends used for file storage.

e.g:

```
FILER_STORAGES = {
    'public': {
        'main': {
            'ENGINE': 'filer.storage.PublicFileSystemStorage',
            'OPTIONS': {
                'location': '/path/to/media/filer',
                'base_url': '/media/filer/',
            },
            'UPLOAD_TO': 'filer.utils.generate_filename.randomized',
            'UPLOAD_TO_PREFIX': 'filer_public',
        },
        'thumbnails': {
            'ENGINE': 'filer.storage.PublicFileSystemStorage',
            'OPTIONS': {
                'location': '/path/to/media/filer_thumbnails',
                'base_url': '/media/filer_thumbnails/',
            },
        },
    },
    'private': {
        'main': {
            'ENGINE': 'filer.storage.PrivateFileSystemStorage',
            'OPTIONS': {
                'location': '/path/to/smedia/filer',
                'base_url': '/smedia/filer/',
            },
            'UPLOAD_TO': 'filer.utils.generate_filename.randomized',
            'UPLOAD_TO_PREFIX': 'filer_public',
        },
        'thumbnails': {
            'ENGINE': 'filer.storage.PrivateFileSystemStorage',
            'OPTIONS': {
                'location': '/path/to/smedia/filer_thumbnails',
                'base_url': '/smedia/filer_thumbnails/',
            },
        },
    },
}
```

Defaults to FileSystemStorage in `<MEDIA_ROOT>/filer_public/` and `<MEDIA_ROOT>/filer_public_thumbnails/` for public files and `<MEDIA_ROOT>/../smedia/filer_private/` and `<MEDIA_ROOT>/../smedia/filer_private_thumbnails/` for private files. Public storage uses `DEFAULT_FILE_STORAGE` as default storage backend.

`UPLOAD_TO` is the function to generate the path relative to the storage root. The default generates a random path like `1d/a5/1da50fee-5003-46a1-a191-b547125053a8/filename.jpg`. This will be applied whenever a

file is uploaded or moved between public (without permission checks) and private (with permission checks) storages. Defaults to `'filer.utils.generate_filename.randomized'`.

FILER_SERVERS

> **Warning:** Server Backends are experimental and the API may change at any time.

A dictionary to configure server backends to serve files with permissions.

e.g:

```
DEFAULT_FILER_SERVERS = {
    'private': {
        'main': {
            'ENGINE': 'filer.server.backends.default.DefaultServer',
        },
        'thumbnails': {
            'ENGINE': 'filer.server.backends.default.DefaultServer',
        }
    }
}
```

Defaults to using the DefaultServer (doh)! This will serve the files with the django app.

FILER_PAGINATE_BY

The number of items (Folders, Files) that should be displayed per page in admin.

Defaults to `20`

FILER_SUBJECT_LOCATION_IMAGE_DEBUG

Draws a red circle around to point in the image that was used to do the subject location aware image cropping.

Defaults to `False`

FILER_ALLOW_REGULAR_USERS_TO_ADD_ROOT_FOLDERS

Regular users are not allowed to create new folders at the root level, only subfolders of already existing folders, unless this setting is set to `True`.

Defaults to `False`

FILER_IMAGE_MODEL

Defines the dotted path to a custom Image model; please include the model name. Example: 'my.app.models.CustomImage'

Defaults to `False`

FILER_CANONICAL_URL

Defines the path element common to all canonical file URLs.

Defaults to `'canonical/'`

FILER_UPLOADER_CONNECTIONS

Number of simultaneous AJAX uploads. Defaults to 3.

If your database backend is SQLite it would be set to 1 by default. This allows to avoid `database is locked` errors on SQLite during multiple simultaneous file uploads.

Development

Working on front-end

To started development fron-end part of `django-filer` simply install all the packages over npm:

```
npm install
```

To compile and watch scss, run javascript unit-tests, jshint and jscs watchers:

```
gulp
```

To compile scss to css:

```
gulp sass
```

To run sass watcher:

```
gulp sass:watch
```

To run javascript linting and code styling analysis:

```
gulp lint
```

To run javascript linting and code styling analysis watcher:

```
gulp lint:watch
```

To run javascript linting:

```
gulp jshint
```

To run javascript code style analysis:

```
gulp jscs
```

To fix javascript code style errors:

```
gulp jscs:fix
```

To run javascript unit-tests:

```
gulp tests:unit
```

Contributing

Claiming Issues

Since github issues does not support assigning an issue to a non collaborator (yet), please just add a comment on the issue to claim it.

Code Guidelines

The code should be PEP8 compliant. With the exception that the line width is not limited to 80, but to 120 characters.

The flake8 command can be very helpful (we run it as a separate env through *Tox* on *Travis*). If you want to check your changes for code style:

```
$ flake8
```

This runs the checks without line widths and other minor checks, it also ignores source files in the *migrations* and *tests* and some other folders.

This is the last command to run before submitting a PR (that will run tests in all tox environments):

```
$ tox
```

Another useful tool is reindent. It fixes whitespace and indentation stuff:

```
$ reindent -n filer/models/filemodels.py
```

Workflow

Fork -> Code -> Pull request

django-filer uses the excellent branching model from nvie. It is highly recommended to use the git flow extension that makes working with this branching model very easy.

- fork django-filer on github
- clone your fork `git clone git@github.com:username/django-filer.git`
- `cd django-filer`
- initialize git flow: `git flow init` (choose all the defaults)
- `git flow feature start my_feature_name` creates a new branch called `feature/my_feature_name` based on `develop`
- ...code... ...code... ..commit.. ..commit..
- `git flow feature publish` creates a new branch remotely and pushes your changes
- navigate to the feature branch on github and create a pull request to the `develop` branch on `divio/django-filer`
- after reviewing the changes may be merged into `develop` and then eventually into `master` for the release.

If the feature branch is long running, it is good practice to merge in the current state of the `develop` branch into the feature branch sometimes. This keeps the feature branch up to date and reduces the likeliness of merge conflicts once it is merged back into develop.

Extending Django Filer

Django Filer ships with support for image files, and generic files (everything that's not an image).

So what if you wanted to add support for a particular kind of file thats's not already included? It's easy to extend it to do this, without needing to touch the Filer's code at all (and no-one wants to have to maintain fork if they can avoid it).

So for example, you might want to be able to manage video files. You could of course simply store and file them as generic file types, but that might not be enough - perhaps your own application needs to know that certain files are in fact video files, so that it can treat them appropriately (process them, allow only them to be selected in certain widgets, and so on).

In this example we will create support for video files.

The model

The very basics

In your own application, you need to create a Video model. This model has to inherit from `filer.models.filemodels.File`.

```
# import the File class to inherit from
from filer.models.filemodels import File

# we'll need to refer to filer settings
from filer import settings as filer_settings

class Video(File):
    pass # for now...
```

When a file is uploaded, `filer.admin.clipboardadmin.ClipboardAdmin.ajax_upload()` loops over the different models in `filer.settings.FILER_FILE_MODELS` and calls its `matches_file_type()` to see if the file matches a known filename extension.

When a match is found, the filer will create an instance of that class for the file.

So let's add a `matches_file_type()` method to the `Video` model:

```
@classmethod
def matches_file_type(cls, iname, ifile, request):
    # the extensions we'll recognise for this file type
    filename_extensions = ['.dv', '.mov', '.mp4', '.avi', '.wmv',]
    ext = os.path.splitext(iname)[1].lower()
    return ext in filename_extensions
```

Now you can upload files of those types into the Filer.

For each one you upload an instance of your `Video` class will be created.

Icons

At the moment, the files you upload will have the Filer's generic file icon - not very appropriate or helpful for video. What you need to do is add a suitable `_icon` attribute to the class.

The `filer.models.filemodels.File` class we've inherited from has an `icons()` property, from `filer.models.mixins.IconsMixin`.

This checks for the `_icon` attribute; if it finds one, it uses it to build URLs for the icons in various different sizes. If `_icons` is `video`, a typical result might be `/static/filer/icons/video_48x48.png`.

Of course, you can also create an `icons()` property specific to your new model. For example, `filer.models.imagemodels.Image` does that, so that it can create thumbnail icons for each file rather than a single icon for all of that type.

In our `Video` model the simple case will do:

```
# the icon it will use
_icon = "video"
```

And in fact, the Filer *already* has an icon that matches this - if there were not already a set of video icons in the Filer's static assets, we'd have to provide them - see `filer/static/icons` for examples.

The admin

Now we need to register our new model with the admin. Again, the very simplest case:

```
from django.contrib import admin
from filer.admin.fileadmin import FileAdmin
from models import Video

admin.site.register(Video, FileAdmin) # use the standard FileAdmin
```

... but of course if your model had particular fields of its own (as for example the `Image` model has a `subject_location` field) you would create your own ModelAdmin class for it, along with a form, special widgets and whatever else you needed.

Using your new file type

You've now done enough to be able to get hold of files of your new kind in the admin (wherever the admin uses a `FilerFileField`) but to make it really useful we need to to a little more.

For example, it might be useful to have:

- its own field type to get hold of it in some other model
- a special form for the field
- a widget for selecting it in the admin
- ... and so on

How you use it will be up to you, but a fairly typical use case would be in a django CMS plugin, and that is the example that will be followed here.

Create a custom field for your file type

```
from filer.fields.file import FilerFileField

class FilerVideoField(FilerFileField):
    default_model_class = Video
```

Of course you could also create an admin widget and admin form, but it's not necessary at this stage - the ones generic files use will do just fine.

Create some other model that uses it

Here, it's going to be a django CMS plugin:

```
from cms.models import CMSPlugin

class VideoPluginEditor(CMSPlugin):
    video = FilerVideoField()
    # you'd probably want some other fields in practice...
```

You'll have to provide an admin class for your model; in this case, the admin will be provided as part of the django CMS plugin architecture.

Note: If you are not already familiar with the django CMS plugin architecture, http://docs.django-cms.org/en/latest/extending_cms/custom_plugins.html#overview will provide an explanation.

```
from cms.plugin_base import CMSPluginBase
from models import VideoPluginEditor

class VideoPluginPublisher(CMSPluginBase):
    model = VideoPluginEditor
    render_template = "video/video.html"
    text_enabled = True
    admin_preview = False

    def icon_src(self, instance):
        return "/static/plugin_icons/video.png"

    def render(self, context, instance, placeholder):
        context.update({
            'video':instance,
            'placeholder':placeholder,
        })
        return context

plugin_pool.register_plugin(VideoPluginPublisher)
```

... and now, assuming you have created a suitable `video/video.html`, you've got a working plugin that will make use of your new Filer file type.

Other things you could add

Admin templating

`filer/templates/templates/admin/filer/folder` lists the individual items in each folder. It checks `item.file_type` to determine how to display those items and what to display for them.

You might want to extend this, so that the list includes the appropriate information for your new file type. In that case you will need to override the template, and in the `Video` model:

```
# declare the file_type for the list template
file_type = 'Video'
```

Note that if you do this, you *will* need to override the template - otherwise your items will fail to display in the folder lists.

Overriding the Directory Listing Search

By default, filer will search against `name` for `Folders` and `name`, `description`, and `original_filename` for `Files`, in addition to searching against the owner. If you are using `auth.User` as your User model, filer will search against the `username`, `first_name`, `last_name`, `email` fields. If you are using a custom User model, filer will search against all fields that are CharFields except for the password field. You can override this behavior by subclassing the `filer.admin.folderadmin.FolderAdmin` class and overriding the `owner_search_fields` property.

```
# in an admin.py file
from django.contrib import admin
from filer.admin import FolderAdmin
from filer.models import Folder

class MyFolderAdmin(FolderAdmin):
    owner_search_fields = ['field1', 'field2']

admin.site.unregister(Folder)
admin.site.register(Folder, FolderAdmin)
```

You can also override the search behavior for `Folders`. Just override `search_fields` by subclassing the `filer.admin.folderadmin.FolderAdmin`. It works as described in Django's docs. E.g.:

```
# in an admin.py file
class MyFolderAdmin(FolderAdmin):
    search_fields = ['=field1', '^field2']

admin.site.unregister(Folder)
admin.site.register(Folder, MyFolderAdmin)
```

Providing custom Image model

As the `Image` model is special, a different way to implement custom Image model is required.

Defining the model

First a custom model must be defined; it should inherit from BaseImage, the basic abstract class:

```
from filer.models.abstract.BaseImage

class CustomImage(BaseImage):
    my_field = models.CharField(max_length=10)

    class Meta:
        # You must define a meta with en explicit app_label
        app_label = 'myapp'
```

The model can be defined in any installed application declared **after** `django-filer`.

`BaseImage` defines the following fields (plus the basic fields defined in `File`):

- default_alt_text
- default_caption
- subject_location

you may add whatever fields you need, just like any other model.

..warning: `app_label` in `Meta` must be explicitly defined.

Customize the admin

If you added fields in your custom Image model, you have to customize the admin too:

```
from django.contrib import admin
from filer.admin.imageadmin import ImageAdmin
from filer.models.imagemodels import Image

class CustomImageAdmin(ImageAdmin):
    # your custom code
    pass

# Using build_fieldsets allows to easily integrate common field in the admin
# Don't define fieldsets in the ModelAdmin above and add the custom fields
# to the ``extra_main_fields`` or ``extra_fieldsets`` as shown below
CustomImageAdmin.fieldsets = CustomImageAdmin.build_fieldsets(
    extra_main_fields=('default_alt_text', 'default_caption', 'my_field'...),
    extra_fieldsets=(
        ('Subject Location', {
            'fields': ('subject_location',),
            'classes': ('collapse',),
        }),
    )
)

# Unregister the default admin
admin.site.unregister(ImageAdmin)
# Register your own
admin.site.register(Image, CustomImageAdmin)
```

Swap the Image model

Set `FILER_IMAGE_MODEL` to the dotted path of your custom model:

```
FILER_IMAGE_MODEL = 'my.app.models.CustomImage'
```

Running tests

django-filer is continuously being tested on travis-ci.

There is no easy way to run test suite on any python version you have installed without using `tox`.

The recommended way to test locally is with tox. Once `tox` is installed, simply running the `tox` command inside the package root. You don't need to bother with any virtualenvs, it will be done for you. Tox will setup multiple virtual environments with different python and django versions to test against:

```
# run all tests in all default environments
tox
# run tests on particular versions
tox -e py27-dj18,py34-dj_master
```

```
# run a test class in specific environment
tox -e py27-dj18 -- test filer.tests.models.FilerApiTests
# run a specific testcase in specific environment
tox -e py27-dj18 -- test filer.tests.models.FilerApiTests.test_create_folder_structure
```

Other test runner options are also supported, see djangocms-helper documentation for details.

To speed things up a bit use detox. `detox` runs each testsuite in a separate process in parallel. Detox also supports using `pyenv` to install multiple python versions.

Dump payload

django-filer stores the meta-data of each file in the database, while the files payload is stored on disk. This is fine, since large binary data shall only exceptionally be stored in a relational database. The consequence however is, that when invoking `manage dumpdata` only the meta-data is dumped, while the payload remains on disk. During backups this can be a problem, since the payload must be handled though other means, for example `tar` or `zip`.

django-filer has a feature, which allows to dump the files payload together with their meta-data. This is achieved by converting the payload into a BASE64 string which in consequence is added to the dumped data. The advantage is, that the dumped file can be imported without having to fiddle with zip, tar and file pathes.

In order to activate this feature, add:

```
FILER_DUMP_PAYLOAD = True
```

to the projects `settings.py` file.

If the content has been dumped together with to payload, the files are restored when using `manage.py loaddata`. If the payload is missing, only the meta-data is restored. This is the default behavior.

Other benefits

- It simplifies backups and migrations, since the data entered into the content management system is dumped into one single file.
- If the directory `filer_public` is missing, **django-filer** rebuilds the file tree from scratch. This can be used to get rid of zombie files, such as generated thumbnails which are not used any more.
- When dumping the filers content, you get warned about missing files.
- When dumping the filers content, the checksum of the dumped file is compared to that generated during the primary file upload. In case the checksum diverges, you will be warned.
- Only the uploaded file is dumped. Thumbnails derived from the uploaded files will be regenerated by **django-filer** when required. This saves some space during backups.

Reloading a dump back to the database

If you dumped a whole database, and not only a partial application, then you may encounter problems with primary key conflicts during an import:

```
Could not load contenttypes.ContentType(...)
```

To circumvent this, first you must flush the whole database's content. Using the management command `./manage.py flush`, does not truncate all tables: This is because the content in the table `django_content_type` is reset to the state after initializing the database using `./manage.py syncdb` and thus is not empty.

Therefore, to flush the database's content, use:

```
./manage.py sqlflush | ./manage.py dbshell
```

Now the dump can be loaded using:

```
./manage.py loaddata <dumpfile>
```

Management commands

Generating thumbnails

django-filer generates preview thumbnails of the images which are displayed in the admin. Usually these are generated on the fly when images are uploaded. If however you have imported the images programmatically you may want to generate the thumbnails eagerly utilizing a management command.

To generate them, use:

```
./manage.py generate_thumbnails
```