# Django FileBrowser Documentation

***Release 3.11.3***

**Patrick Kranzlmueller**

**Nov 14, 2019**

# Contents

**Media-Management with Grappelli**.

**Note:** FileBrowser 3.11.3 requires Django 2.1 and Grappelli 2.12.

CHAPTER 1

# Installation and Setup

## 1.1 Quick start guide

For using the FileBrowser, Django needs to be installed and an Admin Site has to be activated.

### 1.1.1 Requirements

- Django 1.11, http://www.djangoproject.com
- Grappelli 2.9, https://github.com/sehmaschine/django-grappelli
- Pillow, https://github.com/python-imaging/Pillow

### 1.1.2 Installation

Install the FileBrowser:

```
pip install django-filebrowser
```

Add the filebrowser to your `INSTALLED_APPS` (before django.contrib.admin):

```
INSTALLED_APPS = (
    'grappelli',
    'filebrowser',
    'django.contrib.admin',
)
```

Add the FileBrowser site to your url-patterns (before any admin-urls):

```
from filebrowser.sites import site

urlpatterns = [
```

(continues on next page)

(continued from previous page)

```
    path('admin/filebrowser/', site.urls),
    path('grappelli/', include('grappelli.urls')),
    path('admin/', admin.site.urls),
]
```

Collect the static files (please refer to the Staticfiles Documentation for more information):

```
python manage.py collectstatic
```

### 1.1.3 Settings

Check the *Settings*.

**Note:** You need to add a folder "uploads" within `site.storage.location` when using the default settings.

### 1.1.4 Testing

Start the devserver and login to your admin site:

```
python manage.py runserver <IP-address>:8000
```

Goto /admin/filebrowser/browse/ and check if everything looks/works as expected. If you're having problems, see *Troubleshooting*.

## 1.2 Settings

There are some settings in order to customize the FileBrowser. Nonetheless, you should be able to start with the default settings.

All settings can be defined in your projects settings-file. In that case, you have to use the prefix `FILEBROWSER_` for every setting (e.g. `FILEBROWSER_EXTENSIONS` instead of `EXTENSIONS`).

### 1.2.1 Main URL/Paths Settings

#### DIRECTORY (relative to storage location)

Main FileBrowser Directory. Leave empty in order to browse all files and folders within a storage location:

```
DIRECTORY = getattr(settings, "FILEBROWSER_DIRECTORY", 'uploads/')
```

You can override this setting on a per–site basis:

```
from filebrowser.sites import site
site.directory = "uploads/"
```

**Warning:** If you define site.directory, make sure to use a trailing slash.

### 1.2.2 Extensions and Formats

#### EXTENSIONS

Allowed extensions for file upload:

```
EXTENSIONS = getattr(settings, "FILEBROWSER_EXTENSIONS", {
    'Image': ['.jpg','.jpeg','.gif','.png','.tif','.tiff'],
    'Document': ['.pdf','.doc','.rtf','.txt','.xls','.csv'],
    'Video': ['.mov','.wmv','.mpeg','.mpg','.avi','.rm'],
    'Audio': ['.mp3','.mp4','.wav','.aiff','.midi','.m4p']
})
```

#### SELECT_FORMATS

Set different Options for selecting elements from the FileBrowser:

```
SELECT_FORMATS = getattr(settings, "FILEBROWSER_SELECT_FORMATS", {
    'file': ['Image','Document','Video','Audio'],
    'image': ['Image'],
    'document': ['Document'],
    'media': ['Video','Audio'],
})
```

When using the browse-function for selecting Files/Folders, you can use an additional query-attribute `type` in order to restrict the choices.

### 1.2.3 Versions

#### VERSIONS_BASEDIR (relative to storage location)

Directory to save image versions (and thumbnails). If no directory is given, versions are stored at the same location as the original image:

```
VERSIONS_BASEDIR = getattr(settings, 'FILEBROWSER_VERSIONS_BASEDIR', '_versions')
```

We do recommend the following structure for media files:

```
└── media  # site.storage.location (e.g. MEDIA_ROOT)
    ├── _versions  # VERSIONS_BASEDIR (outside of site.directory)
    └── uploads  # site.directory
```

**Warning:** From version 3.7, defining a VERSIONS_BASEDIR outside of site.directory is mandatory.

#### VERSIONS

Each key in the `VERSIONS` dict contains an *options* dict with params that will be forwarded to the image processors chain.

If an option is not recognized by the processors in use, it will be ignored. This allows you to define custom options to be used with *Custom processors*.

Define the versions according to your websites grid:

```
VERSIONS = getattr(settings, "FILEBROWSER_VERSIONS", {
    'admin_thumbnail': {'verbose_name': 'Admin Thumbnail', 'width': 60, 'height': 60,
↪'opts': 'crop'},
    'thumbnail': {'verbose_name': 'Thumbnail (1 col)', 'width': 60, 'height': 60,
↪'opts': 'crop'},
    'small': {'verbose_name': 'Small (2 col)', 'width': 140, 'height': '', 'opts': ''}
↪,
    'medium': {'verbose_name': 'Medium (4col )', 'width': 300, 'height': '', 'opts': '
↪'},
    'big': {'verbose_name': 'Big (6 col)', 'width': 460, 'height': '', 'opts': ''},
    'large': {'verbose_name': 'Large (8 col)', 'width': 680, 'height': '', 'opts': ''}
↪,
})
```

### VERSION_QUALITY

Quality of saved versions:

```
VERSION_QUALITY = getattr(settings, 'FILEBROWSER_VERSION_QUALITY', 90)
```

### ADMIN_VERSIONS

The versions you want to show with the admin interface:

```
ADMIN_VERSIONS = getattr(settings, 'FILEBROWSER_ADMIN_VERSIONS', ['thumbnail', 'small
↪', 'medium', 'big', 'large'])
```

### ADMIN_THUMBNAIL

The version being used as the admin thumbnail:

```
ADMIN_THUMBNAIL = getattr(settings, 'FILEBROWSER_ADMIN_THUMBNAIL', 'admin_thumbnail')
```

### VERSION_PROCESSORS

New in version 3.7.2.

An image version is generated by passing the source image through a series of image processors. Each processor may alter the image, often dependent on the options it receives.

You can define the image processors through which the source image is run when you create a version by overriding:

```
VERSION_PROCESSORS = getattr(settings, 'FILEBROWSER_VERSION_PROCESSORS', [
    'filebrowser.utils.scale_and_crop',
])
```

The order of the processors is the order in which they will be sequentially called to process the image. The image received by a processor is the output of the previous processor.

**See also:**

*Custom processors*.

#### VERSION_NAMER

New in version 3.7.2.

The class used to generate the filename for versions:

```
VERSION_NAMER = getattr(settings, 'FILEBROWSER_VERSION_NAMER', 'filebrowser.namers.
↪VersionNamer')
```

Namers bult-in:

**filebrowser.namers.VersionNamer** Default. Generates a name based on the `version_suffix`.

**filebrowser.namers.OptionsNamer** Generates a name using the options provided to the *FileObject.version_generate* and the options in *VERSIONS* if an `version_suffix` is provided. Restores the original file name wipping out the last ‘‘_version_suffix–plus-any-configs‘ block entirely.

### 1.2.4 Placeholder

With your locale environment, you don’t necessarily have access to all media files (e.g. images uploaded by your client). Therefore, you can use a PLACEHOLDER.

#### PLACEHOLDER

Path to placeholder image (relative to storage location):

```
PLACEHOLDER = getattr(settings, "FILEBROWSER_PLACEHOLDER", "")
```

#### SHOW_PLACEHOLDER

Show placeholder (instead of a version) if the original image does not exist:

```
SHOW_PLACEHOLDER = getattr(settings, "FILEBROWSER_SHOW_PLACEHOLDER", False)
```

#### FORCE_PLACEHOLDER

Always show placeholder (even if the original image exists):

```
FORCE_PLACEHOLDER = getattr(settings, "FILEBROWSER_FORCE_PLACEHOLDER", False)
```

### 1.2.5 Extra Settings

#### STRICT_PIL

If set to `True`, the FileBrowser will not try to import a mis-installed PIL:

```
STRICT_PIL = getattr(settings, 'FILEBROWSER_STRICT_PIL', False)
```

### IMAGE_MAXBLOCK

see http://mail.python.org/pipermail/image-sig/1999-August/000816.html:

```
IMAGE_MAXBLOCK = getattr(settings, 'FILEBROWSER_IMAGE_MAXBLOCK', 1024*1024)
```

### EXCLUDE

Exclude-patterns for files you don't want to show:

```
EXTENSION_LIST = []
for exts in EXTENSIONS.values():
    EXTENSION_LIST += exts
EXCLUDE = getattr(settings, 'FILEBROWSER_EXCLUDE', (r'_(%(exts)s)_.*_q\d{1,3}\.(
↪%(exts)s)' % {'exts': ('|'.join(EXTENSION_LIST))},))
```

### MAX_UPLOAD_SIZE

Max. Upload Size in Bytes:

```
MAX_UPLOAD_SIZE = getattr(settings, "FILEBROWSER_MAX_UPLOAD_SIZE", 10485760)
```

### NORMALIZE_FILENAME

`True` if you want to normalize filename on upload and remove all non-alphanumeric characters (except for underscores, spaces & dashes):

```
NORMALIZE_FILENAME = getattr(settings, "FILEBROWSER_NORMALIZE_FILENAME", False)
```

### CONVERT_FILENAME

`True` if you want to convert the filename on upload (replace spaces and convert to lowercase):

```
CONVERT_FILENAME = getattr(settings, "FILEBROWSER_CONVERT_FILENAME", True)
```

### LIST_PER_PAGE

How many items appear on each paginated list:

```
LIST_PER_PAGE = getattr(settings, "FILEBROWSER_LIST_PER_PAGE", 50)
```

### DEFAULT_SORTING_BY

Default sorting attribute:

```
DEFAULT_SORTING_BY = getattr(settings, "FILEBROWSER_DEFAULT_SORTING_BY", "date")
```

Options are: `date`, `filesize`, `filename_lower`, `filetype_checked`, `mimetype`. You can also combine attributes, e.g. `('mimetype', 'filename_lower')`.

### DEFAULT_SORTING_ORDER

Default sorting order:

```
DEFAULT_SORTING_ORDER = getattr(settings, "FILEBROWSER_DEFAULT_SORTING_ORDER", "desc")
```

Options are: `asc` or `desc`

### FOLDER_REGEX

regex to clean directory names before creation:

```
FOLDER_REGEX = getattr(settings, "FILEBROWSER_FOLDER_REGEX", r'^[\w._\ /-]+$')
```

### SEARCH_TRAVERSE

`True` if you want to traverse all subdirectories when searching. Please note that with thousands of files/directories, this might take a while:

```
SEARCH_TRAVERSE = getattr(settings, "FILEBROWSER_SEARCH_TRAVERSE", False)
```

### DEFAULT_PERMISSIONS

Default upload and version permissions:

```
DEFAULT_PERMISSIONS = getattr(settings, "FILEBROWSER_DEFAULT_PERMISSIONS", 0o755)
```

### OVERWRITE_EXISTING

`True` in order to overwrite existing files. `False` to use the behaviour of the storage engine:

```
OVERWRITE_EXISTING = getattr(settings, "FILEBROWSER_OVERWRITE_EXISTING", True)
```

CHAPTER 2

# API

## 2.1 API

### 2.1.1 FileListing

**class FileListing**(*path*, *filter_func=None*, *sorting_by=None*, *sorting_order=None*)

Returns a list of FileObjects for a server path, see *FileObject*.

**Parameters**

- **path** – Relative path to a location within *site.storage.location*.
- **filter_func** – Filter function, see example below.
- **sorting_by** – Sort the files by any attribute of FileObject.
- **sorting_order** – Sorting order, either "asc" or "desc".

If you want to list all files within a storage location you do:

```
from filebrowser.sites import site
from filebrowser.base import FileListing
filelisting = FileListing(site.storage.location, sorting_by='date', sorting_order=
↪'desc')
```

Use a custom filter function to limit the list of files:

```
def filter_filelisting(item):
    # item is a FileObject
    return item.filetype != "Folder"

filelisting = FileListing(site.storage.location, filter_func=filter_listing, sorting_
↪by='date', sorting_order='desc')
```

### Methods

For the below examples, we're using this folder-structure.:

```
/media/uploads/testfolder/testimage.jpg
/media/uploads/blog/1/images/blogimage.jpg
```

**Note:** We defined `filter_browse` as `filter_func` (see sites.py). And we did not define a `VERSIONS_BASEDIR` for this demonstration, though it is highly recommended to use one.

**`listing`**()
Returns all items for the given path with `os.listdir(path)`:

```
>>> for item in filelisting.listing():
...     print item
blog
testfolder
```

**`walk`**()
Returns all items for the given path with `os.walk(path)`:

```
>>> for item in filelisting.walk():
...     print item
blog
blog/1
blog/1/images
blog/1/images/blogimage.jpg
blog/1/images/blogimage_admin_thumbnail.jpg
blog/1/images/blogimage_medium.jpg
blog/1/images/blogimage_small.jpg
blog/1/images/blogimage_thumbnail.jpg
testfolder
testfolder/testimage.jpg
```

**`files_listing_total`**()
Returns a sorted list of `FileObjects` for *`listing()`*:

```
>>> for item in filelisting.files_listing_total():
...     print item
uploads/blog/
uploads/testfolder/
```

**`files_walk_total`**()
Returns a sorted list of `FileObjects` for *`walk()`*:

```
>>> for item in filelisting.files_walk_total():
...     print item
uploads/blog/
uploads/blog/1/
uploads/blog/1/images/
uploads/blog/1/images/blogimage.jpg
uploads/blog/1/images/blogimage_admin_thumbnail.jpg
uploads/blog/1/images/blogimage_medium.jpg
uploads/blog/1/images/blogimage_small.jpg
uploads/blog/1/images/blogimage_thumbnail.jpg
```

(continues on next page)

(continued from previous page)

```
uploads/testfolder/
uploads/testfolder/testimage.jpg
```

**files_listing_filtered**()
Returns a sorted and filtered list of `FileObjects` for *listing()*:

```
>>> for item in filelisting.files_listing_filtered():
...     print item
uploads/blog/
uploads/testfolder/
```

**files_walk_filtered**()
Returns a sorted and filtered list of `FileObjects` for *walk()*:

```
>>> for item in filelisting.files_walk_filtered():
...     print item
uploads/blog/
uploads/blog/1/
uploads/blog/1/images/
uploads/blog/1/images/blogimage.jpg
uploads/testfolder/
uploads/testfolder/testimage.jpg
```

---

**Note:** The versions are not listed (compared with files_walk_total) because of filter_func.

---

**results_listing_total**()
Number of total files, based on *files_listing_total()*:

```
>>> filelisting.results_listing_total()
2
```

**results_walk_total**()
Number of total files, based on *files_walk_total()*:

```
>>> filelisting.results_walk_total()
10
```

**results_listing_filtered**()
Number of filtered files, based on *files_listing_filtered()*:

```
>>> filelisting.results_listing_filtered()
2
```

**results_walk_filtered**()
Number of filtered files, based on *files_walk_filtered()*:

```
>>> filelisting.results_walk_filtered()
6
```

### 2.1.2 FileObject

**class FileObject**(*path*, *site=None*)
An object representing a media file.

**Parameters**

- **path** – Relative path to a location within *site.storage.location*.
- **site** – An optional FileBrowser Site.

For example:

```
from filebrowser.sites import site
from filebrowser.base import FileObject
fileobject = FileObject(os.path.join(site.directory,"testfolder","testimage.jpg"))
version = FileObject(os.path.join(fileobject.versions_basedir, "testfolder",
↪"testimage_medium.jpg"))
```

## Attributes

### Initial Attributes

**path**
Path relative to a storage location (including `site.directory`):

```
>>> fileobject.path
'uploads/testfolder/testimage.jpg'
```

**head**
The directory name of pathname `path`:

```
>>> fileobject.head
'uploads/testfolder'
```

**filename**
Name of the file (including the extension) or name of the folder:

```
>>> fileobject.filename
'testimage.jpg'
```

**filename_lower**
Lower type of `filename`.

**filename_root**
Filename without extension:

```
>>> fileobject.filename_root
'testimage'
```

**extension**
File extension, including the dot. With a folder, the extensions is `None`:

```
>>> fileobject.extension
'.jpg'
```

**mimetype**
Mimetype, based on http://docs.python.org/library/mimetypes.html:

```
>>> fileobject.mimetype
('image/jpeg', None)
```

### General Attributes

**filetype**
Type of the file, as defined with `EXTENSIONS`:

```
>>> fileobject.filetype
'Image'
```

**format**
Type of the file, as defined with `SELECT_FORMATS`:

```
>>> fileobject.format
'file'
```

**filesize**
Filesize in Bytes:

```
>>> fileobject.filesize
870037L
```

**date**
Date, based on `time.mktime`:

```
>>> fileobject.date
1299760347.0
```

**datetime**
Datetime object:

```
>>> fileobject.datetime
datetime.datetime(2011, 3, 10, 13, 32, 27)
```

**exists**
`True`, if the path exists, `False` otherwise:

```
>>> fileobject.exists
True
```

### Path and URL attributes

**path**
Path relative to a storage location (including `site.directory`):

```
>>> fileobject.path
'uploads/testfolder/testimage.jpg'
```

**path_relative_directory**
Path relative to `site.directory`:

```
>>> fileobject.path_relative_directory
'testfolder/testimage.jpg'
```

**path_full**
Absolute server path (based on `storage.path`):

```
>>> fileobject.path_full
'/absolute/path/to/server/location/testfolder/testimage.jpg'
```

**dirname**
New in version 3.4.

The directory (not including `site.directory`):

```
>>> fileobject.dirname
'testfolder'
```

**url**
URL for the file/folder (based on `storage.url`):

```
>>> fileobject.url
'/media/uploads/testfolder/testimage.jpg'
```

### Image attributes

The image attributes are only useful if the `FileObject` represents an image.

**dimensions**
Image dimensions as a tuple:

```
>>> fileobject.dimensions
(1000, 750)
```

**width**
Image width in px:

```
>>> fileobject.width
1000
```

**height**
Image height in px:

```
>>> fileobject.height
750
```

**aspectratio**
Aspect ratio (float format):

```
>>> fileobject.aspectratio
1.33534908
```

**orientation**
Image orientation, either `Landscape` or `Portrait`:

```
>>> fileobject.orientation
'Landscape'
```

### Folder attributes

The folder attributes make sense when the `FileObject` represents a directory (not a file).

**is_folder**
`True`, if path is a folder:

```
>>> fileobject.is_folder
False
```

**is_empty**
`True`, if the folder is empty. `False` if the folder is not empty or the `FileObject` is not a folder:

```
>>> fileobject.is_empty
False
```

### Version attributes

**is_version**
`true` if the File is a `version` of another File:

```
>>> fileobject.is_version
False
>>> version.is_version
True
```

**versions_basedir**
The relative path (from storage location) to the main versions folder. Either `VERSIONS_BASEDIR` or `site.directory`:

```
>>> fileobject.versions_basedir
'_versions'
>>> version.versions_basedir
'_versions'
```

**original**
Returns the original FileObject:

```
>>> fileobject.original
<FileObject: uploads/testfolder/testimage.jpg>
>>> version.original
<FileObject: uploads/testfolder/testimage.jpg>
```

**original_filename**
Get the filename of an original image from a version:

```
>>> fileobject.original_filename
'testimage.jpg'
>>> version.original_filename
'testimage.jpg'
```

### Methods

### Version methods

**versions**()
List all filenames based on `VERSIONS`:

```
>>> fileobject.versions()
['_versions/testfolder/testimage_admin_thumbnail.jpg',
'_versions/testfolder/testimage_thumbnail.jpg',
'_versions/testfolder/testimage_small.jpg',
'_versions/testfolder/testimage_medium.jpg',
'_versions/testfolder/testimage_big.jpg',
'_versions/testfolder/testimage_large.jpg']
>>> version.versions()
[]
```

**Note:** The versions are not being generated.

**admin_versions**()

List all filenames based on ADMIN_VERSIONS:

```
>>> fileobject.admin_versions()
['_versions/testfolder/testimage_thumbnail.jpg',
'_versions/testfolder/testimage_small.jpg',
'_versions/testfolder/testimage_medium.jpg',
'_versions/testfolder/testimage_big.jpg',
'_versions/testfolder/testimage_large.jpg']
>>> version.admin_versions()
[]
```

**Note:** The versions are not being generated.

**version_name**(*version_suffix*, *extra_options=None*)

**Parameters**

- **version_suffix** – A suffix to compose the version name accordingly to the *VERSION_NAMER* in use.
- **extra_options** – An optional `dict` to be used in the version generation.

Get the filename for a version:

```
>>> fileobject.version_name("medium")
'testimage_medium.jpg'
```

**Note:** The version is not being generated.

**See also:**

Files names can be customized using *VERSION_NAMER*.

**version_path**(*version_suffix*, *extra_options=None*)

**Parameters**

- **version_suffix** – A suffix to compose the version name accordingly to the *VERSION_NAMER* in use.
- **extra_options** – An optional `dict` to be used in the version generation.

Get the path for a version:

```
>>> fileobject.version_path("medium")
'_versions/testfolder/testimage_medium.jpg'
```

**Note:** The version is not being generated.

**version_generate**(*version_suffix*, *extra_options=None*)

**Parameters**

- **version_suffix** – A suffix to compose the version name accordingly to the *VERSION_NAMER* in use.
- **extra_options** – An optional `dict` to be used in the version generation.

An image version is generated by passing the source image through a series of *image processors*. Each processor may alter the image, often dependent on the options it receives.

The options used in the processors chain is composed of the *version definition*, if `version_suffix` is a key in *VERSIONS*, plus any `extra_options` provided. If no version definition was found and no extra options are provided, an empty dict will be used. A key in `extra_options` will take precedence over the version definition.

Generate a version:

```
>>> fileobject.version_generate("medium")
<FileObject: uploads/testfolder/testimage_medium.jpg>
```

Please note that a version is only generated, if it does not already exist or if the original image is newer than the existing version.

### Delete methods

**delete**()
Delete the `File` or `Folder` from the server.

**Warning:** If you delete a **Folder**, all items within the folder are being deleted.

**delete_versions**()
Delete all `VERSIONS`.

**delete_admin_versions**()
Delete all `ADMIN_VERSIONS`.

CHAPTER 3

# Fields & Widgets

## 3.1 Fields & Widgets

The *FileBrowseField* is a custom model field which returns a *FileObject*.

### 3.1.1 FileBrowseField

**class FileBrowseField**(*max_length*[, *site*, *directory*, *extensions*, *format*, ***options*])

A subclass of CharField, referencing a media file within. Returns a *FileObject*.

**Parameters**

- **site** – A FileBrowser site (defaults to the main site), see *FileBrowser Site*.
- **directory** – Directory to browse when clicking the search icon.
- **extensions** – List of allowed extensions, see *Extensions and Formats*.
- **format** – A key from SELECT_FORMATS in order to restrict the selection to specific filetypes, see *Extensions and Formats*.

For example:

```
from filebrowser.fields import FileBrowseField

class BlogEntry(models.Model):
    image = FileBrowseField("Image", max_length=200, directory="images/", extensions=[
↪".jpg"], blank=True)
    document = FileBrowseField("PDF", max_length=200, directory="documents/",␣
↪extensions=[".pdf",".doc"], blank=True)
```

If you define *extensions*, you'll get a validation error if the selected file/folder doesn't match the extension defined with the field.

If you define *format*, the pop-up for selecting files will only show items which match the definition.

#### FileBrowseField in Templates

You can use all attributes from *FileObject*:

```
{{ blogentry.image }}
<img src="{{ publication.image.url }}" />

{% ifequal blogentry.image.image_orientation "landscape" %}
    <img src="{{ blogentry.image.url }}" class="landscape" />
{% endifequal %}
```

#### Showing Thumbnail in the Changelist

To show a thumbnail with the changelist, you can define a ModelAdmin method:

```
from filebrowser.settings import ADMIN_THUMBNAIL

def image_thumbnail(self, obj):
    if obj.image and obj.image.filetype == "Image":
        return '<img src="%s" />' % obj.image.version_generate(ADMIN_THUMBNAIL).url
    else:
        return ""
image_thumbnail.allow_tags = True
image_thumbnail.short_description = "Thumbnail"
```

#### Using the FileBrowseField with TinyMCE

In order to replace the TinyMCE image/file manager with the FileBrowser, you have to use a FileBrowser Callback. There's an example TinyMCE configuration file in /static/js/ called TinyMCEAdmin.js. You can either copy the FileBrowserCallback to your own file or just use tinymce_setup.js (which comes with django-grappelli).

Just add these lines to your ModelAdmin asset definitions:

```
class Media:
    js = ['/path/to/tinymce/jscripts/tiny_mce/tiny_mce.js',
          '/path/to/your/tinymce_setup.js']
```

### 3.1.2 Django FileField and the FileBrowser

Return a *FileObject* from a FileField or ImageField with:

```
from filebrowser.base import FileObject

image_upload = models.ImageField(u"Image (Upload)", max_length=250, upload_to=image_
↪upload_path, blank=True)

def image(self):
    if self.image_upload:
        return FileObject(self.image_upload.path)
    return None
```

In order show a thumbnail with your changelist, you could use a ModelAdmin method:

```
from filebrowser.base import FileObject

def image_thumbnail(self, obj):
    if obj.image_upload:
        image = FileObject(obj.image_upload.path)
        if image.filetype == "Image":
            return '<img src="%s" />' % image.version_generate(ADMIN_THUMBNAIL).url
    else:
        return ""
image_thumbnail.allow_tags = True
image_thumbnail.short_description = "Thumbnail"
```

**Note:** There are different ways to achieve this. The above examples show one of several options.

CHAPTER 4

# Admin Interface

## 4.1 Admin Interface

The main FileBrowser admin application is an extension for the Django admin interface in order to browser your media folder, upload and rename/delete files.

### 4.1.1 FileBrowser Site

**class FileBrowserSite**(*name=None*, *app_name='filebrowser'*, *storage=default_storage*)

Respresents the FileBrowser admin application (similar to Django's admin site).

**Parameters**

- **name** – A name for the site, defaults to None.
- **app_name** – Defaults to 'filebrowser'.
- **storage** – A custom storage engine, defaults to Djangos default storage.

Similar to `django.contrib.admin`, you first need to add a `filebrowser.site` to your admin interface. In your `urls.py`, import the default FileBrowser site (or your custom site) and add the site to your URL-patterns (before any admin-urls):

```
from filebrowser.sites import site

urlpatterns = patterns('',
   url(r'^adminurl/filebrowser/', include(site.urls)),
)
```

Now you are able to browse the location defined with the storage engine associated to your site.

```
from django.core.files.storage import DefaultStorage
from filebrowser.sites import FileBrowserSite

```

(continues on next page)

(continued from previous page)

```
# Default FileBrowser site
site = FileBrowserSite(name='filebrowser', storage=DefaultStorage())

# My Custom FileBrowser site
custom_site = FileBrowserSite(name='custom_filebrowser', storage=DefaultStorage())
custom_site.directory = "custom_uploads/"
```

**Note:** The module variable `site` from `filebrowser.sites` is the default FileBrowser application.

### 4.1.2 Custom Actions

Similar to Django's admin actions, you can define your FileBrowser actions and thus automate the typical tasks of your users. Registered custom actions are listed in the detail view of a file and a user can select a single action at a time. The selected action will then be applied to the file.

The default FileBrowser image actions, such as "Flip Vertical" or "Rotate 90° Clockwise" are in fact implemented as custom actions (see the module `filebrowser.actions`).

#### Writing Your Own Actions

Custom actions are simple functions:

```
def foo(request, fileobjects):
    # Do something with the fileobjects
```

The first parameter is a `HttpRequest` object (representing the submitted form in which a user selected the action) and the second parameter is a list of `FileObjects` to which the action should be applied.

The list contains exactly one instance of FileObject (representing the file from the detail view), but this may change in the future, as custom actions may become available also in browse views (similar to admin actions applied to a list of checked objects).

#### Registering an Action

In order to make your action visible, you need to register it with a FileBrowser site:

```
site.add_action(foo)
```

Once registered, the action will appear in the detail view of a file. You can also give your action a short description:

```
foo.short_description = 'Do foo with the File'
```

This short description will then appear in the list of available actions. If you do not provide a short description, the function name will be used instead and FileBrowser will replace any underscores in the function name with spaces.

#### Associating Actions with Specific Files

Each custom action can be associated with a specific file type (e.g., images, audio file, etc) to which it applies. In order to do that, you need to define a predicate/filter function, which takes a single argument (FileObject) and returns `True` if your action is applicable to that FileObject. Finally, you need to register this filter function with your action:

```
foo.applies_to(lambda fileobject: fileobject.filetype == 'Image')
```

In the above example, foo will appear in the action list only for image files. If you do not specify any filter function for your action, FileBrowser considers the action as applicable to all files.

#### Messages & Intermediate Pages

You can provide a feedback to a user about a successful or failed execution of an action by using a message. For example:

```
from django.contrib import messages

def desaturate_image(request, fileobjects):
    for f in fileobjects:
        # Desaturate the image
        messages.add_message(request, messages.SUCCESS, _("Image '%s' was desaturated.
↪") % f.filename)
```

Some actions may require user confirmation (e.g., in order to prevent accidental and irreversible modification to files). In order to that, follow the same pattern as with Django's admin action and return a `HttpResponse` object from your action. Good practice for intermediate pages is to implement a confirm view and have your action return `HttpResponseRedirect`:

```
def crop_image(request, fileobjects):
    files = '&f='.join([f.path_relative for f in fileobjects])
    return HttpResponseRedirect('/confirm/?action=crop_image&f=%s' % files)
```

### 4.1.3 File Storages

You have the option to specify which file storage engine a FileBrowser should use to browse/upload/modify your media files. This enables you to use a FileBrowser even if your media files are located at some remote system. See also the Django's documentation on storages https://docs.djangoproject.com/en/dev/topics/files/.

To associate a FileBrowser site with a particular storage engine, set the `storage` property of a site object:

```
from django.core.files.storage import FileSystemStorage
site.storage = FileSystemStorage(location='/path/to/media/directory', base_url='/
↪media/')
```

For storage classes other than FileSystemStorage (or those that inherit from that class), there's more effort involved in providing a storage object that can be used with FileBrowser. See *StorageMixin Class*

#### StorageMixin Class

A FileBrowser uses the Django's Storage class to access media files. However, the API of the Storage class does not provide all methods necessary for FileBrowser's functionality. A `StorageMixin` class from `filebrowser.storage` module therefore defines all the additional methods that a FileBrowser requires:

**isdir** (*self*, *name*)
Returns true if name exists and is a directory.

**isfile** (*self*, *name*)
Returns true if name exists and is a regular file.

**move** (*self*, *old_file_name*, *new_file_name*, *allow_overwrite=False*)
Moves safely a file from one location to another. If `allow_ovewrite==False` and `new_file_name` exists, raises an exception.

**makedirs** (*self*, *name*)
Creates all missing directories specified by name. Analogue to os.mkdirs().

### 4.1.4 Views

All views use the `staff_member_requird` and `path_exists` decorator in order to check if the server path actually exists. Some views also use the `file_exists` decorator.

- **Browse, `fb_browse`** Browse a directory on your server. Returns a *FileListing*.
  - Optional query string args: `dir`, `o`, `ot`, `q`, `p`, `filter_date`, `filter_type`, `type`
- **Create directory, `fb_createdir`** Create a new folder on your server.
  - Optional query string args: `dir`
  - Signals: *filebrowser_pre_createdir*, *filebrowser_post_createdir*
- **Upload, `fb_upload`** Multiple upload.
  - Optional query string args: `dir`, `type`
  - Signals: *filebrowser_pre_upload*, *filebrowser_post_upload*
- **Edit, `fb_edit`** Edit a file or folder.
  - Required query string args: `filename`
  - Optional query string args: `dir`
  - Signals: *filebrowser_pre_rename*, *filebrowser_post_rename*

  You are able to apply custom actions (see *Custom Actions*) to the edit-view.
- **Confirm delete, `fb_confirm_delete`** Confirm the deletion of a file or folder.
  - Required query string args: `filename`
  - Optional query string args: `dir`

  If you try to delete a folder, all files/folders within this folder are listed on this page.
- **Delete, `fb_delete`** Delete a file or folder.
  - Required query string args: `filename`
  - Optional query string args: `dir`
  - Signals: *filebrowser_pre_delete*, *filebrowser_post_delete*

> **Warning:** If you delete a Folder, all items within this Folder are being deleted.

- **Version, `fb_version`** Generate a version of an image as defined with `ADMIN_VERSIONS`.
  - Required query string args: `filename`
  - Optional Query string args: `dir`

  This is a helper used by the `FileBrowseField` and TinyMCE for selecting a version.

### 4.1.5 Signals

The FileBrowser sends a couple of different signals. Please take a look at the module *filebrowser.signals* for further explanation on the provided arguments.

- **filebrowser_pre_upload** Sent before a an Upload starts.
- **filebrowser_post_upload** Sent after an Upload has finished.
- **filebrowser_pre_delete** Sent before an Item (File, Folder) is deleted.
- **filebrowser_post_delete** Sent after an Item (File, Folder) has been deleted.
- **filebrowser_pre_createdir** Sent before a new Folder is created.
- **filebrowser_post_createdir** Sent after a new Folder has been created.
- **filebrowser_pre_rename** Sent before an Item (File, Folder) is renamed.
- **filebrowser_post_rename** Sent after an Item (File, Folder) has been renamed.
- **filebrowser_actions_pre_apply** Sent before a custom action is applied.
- **filebrowser_actions_post_apply** Sent after a custom action has been applied.

#### Example for using these Signals

Here's a small example for using the above Signals:

```
from filebrowser import signals

def pre_upload_callback(sender, **kwargs):
    """
    Receiver function called before an upload starts.
    """
    print "Pre Upload Callback"
    print "kwargs:", kwargs
signals.filebrowser_pre_upload.connect(pre_upload_callback)

def post_upload_callback(sender, **kwargs):
    """
    Receiver function called each time an upload has finished.
    """
    print "Post Upload Callback"
    print "kwargs:", kwargs
    # You can use all attributes available with the FileObject
    # This is just an example ...
    print "Filesize:", kwargs['file'].filesize
    print "Orientation:", kwargs['file'].orientation
    print "Extension:", kwargs['file'].extension
signals.filebrowser_post_upload.connect(post_upload_callback)
```

CHAPTER 5

# Image Versions

## 5.1 Versions

With the FileBrowser, you are able to define different versions/sizes for images. This enables you to save an original image on your server while having different versions of that image to automatically fit your websites grid. Versions are also useful for responsive/adaptive layouts.

To generate a version of a source image, you specify *options* which are used by the image processors (see *VERSION_PROCESSORS*) to generate the required version.

### 5.1.1 Defining Versions

First you need to know which versions/sizes of an image you'd like to generate with your website. Let's say you're using a 12 column grid with 60px for each column and 20px margin (which is a total of 940px). With this grid, you could (for example) define these image *VERSIONS*:

```
FILEBROWSER_VERSIONS_BASEDIR = '_versions'
FILEBROWSER_VERSIONS = {
  'admin_thumbnail': {'verbose_name': 'Admin Thumbnail', 'width': 60, 'height': 60,
↪'opts': 'crop'},
  'thumbnail': {'verbose_name': 'Thumbnail (1 col)', 'width': 60, 'height': 60, 'opts
↪': 'crop'},
  'small': {'verbose_name': 'Small (2 col)', 'width': 140, 'height': '', 'opts': ''},
  'medium': {'verbose_name': 'Medium (4col )', 'width': 300, 'height': '', 'opts': ''}
↪,
  'big': {'verbose_name': 'Big (6 col)', 'width': 460, 'height': '', 'opts': ''},
  'large': {'verbose_name': 'Large (8 col)', 'width': 680, 'height': '', 'opts': ''},
}
```

Use the `methods` argument, if you need to add a filter:

```
def grayscale(im):
    "Convert image to grayscale"
```

(continues on next page)

(continued from previous page)

```
    if im.mode != "L":
        im = im.convert("L")
    return im

FILEBROWSER_VERSIONS = {
    'big': {'verbose_name': 'Big (6 col)', 'width': 460, 'height': '', 'opts': '',
↪'methods': [grayscale]},
})
```

### 5.1.2 Custom processors

New in version 3.7.2.

Custom processors can be created using a simple method like this:

```
def grayscale_processor(im, grayscale=False, **kwargs):
    if grayscale:
        if im.mode != "L":
            im = im.convert("L")
    return im
```

The first argument for a processor is the source image.

All other arguments are keyword arguments which relate to the list of options received from the *version_generate method*.

Ensure that you explicitly declare all params that could be used by your processor, as the processors arguments can be inspected to get a list of valid options.

In order to turn your processor optional, define the params that your processor expects with a falsy default, and in this case you could return the original image without any modification.

You must also use `**kwargs` at the end of your argument list because all *options* used to generate the version are available to all processors, not just the ones defined in your processor.

Whether a processor actually modifies the image or not, they must always return an image.

#### Using the processor

Override the *VERSION_PROCESSORS* setting:

```
FILEBROWSER_VERSION_PROCESSORS = [
    'filebrowser.utils.scale_and_crop',
    'my_project.my_processors.grayscale_processor',
]
```

And in your versions definition:

```
FILEBROWSER_VERSIONS = {
    'big_gray': {'verbose_name': 'Big (6 col)', 'width': 460, 'grayscale': True},
})
```

### 5.1.3 Versions and the Admin

When using the FileBrowser with the admin interface, you need to define `ADMIN_VERSIONS` and `ADMIN_THUMBNAIL` (see *Settings*). `ADMIN_VERSIONS` are available with the admin, i.e. you are able to see these versions with the image detail view and you are able to select the versions with the *FileBrowseField* model field.

```
FILEBROWSER_ADMIN_VERSIONS = ['thumbnail', 'small', 'medium', 'big', 'large']
FILEBROWSER_ADMIN_THUMBNAIL = 'admin_thumbnail'
```

### 5.1.4 Versions and the Frontend

With the templatetag `version` a version will be generated if it doesn't already exist OR if the original image is newer than the version. In order to update an image, you just overwrite the original image and the versions will be generated automatically (as you request them within your template).

A Model example:

```
from filebrowser.fields import FileBrowseField

class BlogEntry(models.Model):
    image = FileBrowseField("Image", max_length=200, blank=True)
```

With your templates, use `version` if you simply need to retrieve the URL or `version as var` if you need to get a *FileObject*:

```
<!-- load filebrowser templatetags -->
{% load fb_versions %}

<!-- get the url with version -->
<img src="{% version blogentry.image 'medium' %}" />

<!-- get a fileobject with version -->
{% version blogentry.image 'medium' as version_medium %}
{{ version_medium.width }}
<img src="{{ version_medium.url }}" />
```

#### Templatetag `version`

Retrieves/Generates a version and returns an URL:

```
{% version model.field_name version_prefix %}
```

Retrieves/Generates a version and returns a FileObject:

```
{% version model.field_name version_prefix as variable %}
```

**Note:** `version_prefix` can either be a string or a variable. If `version_prefix` is a string, use quotes.

### 5.1.5 Versions in Views

If you have a `FileObject` you can generate/retrieve a version with:

```
v = obj.image.version_generate(version_prefix) # returns a FileObject
```

### 5.1.6 Placeholder

When developing on a locale machine or a development-server, you might not have all the images (resp. media-files) available that are on your production instance and downloading these files on a regular basis might not be an option.

In that case, you can use a placeholder instead of a version. You just need to define the `PLACEHOLDER` and overwrite the settings `SHOW_PLACEHOLDER` and/or `FORCE_PLACEHOLDER` (see *Placeholder*).

### 5.1.7 Management Commands

**`fb_version_generate`**
If you need to generate certain (or all) versions, type:

```
python manage.py fb_version_generate
```

**`fb_version_remove`**
If you need to remove certain (or all) versions, type:

```
python manage.py fb_version_remove
```

> **Warning:** Please be very careful with this command.

CHAPTER 6

# Help

## 6.1 Help

### 6.1.1 FAQ

#### Why should I use the FileBrowser?

If you need your editors or customers to manage files, the FileBrowser is an alternative to an FTP-client. Moreover, you are able to define different image versions according to your websites grid. Alternatives to the FileBrowser can be found at http://djangopackages.com/grids/g/file-managers/.

#### Do I need Grappelli?

Grappelli is a requirement for using the FileBrowser. There are several filebrowser-no-grappelli repositories (most of them on GitHub), but we don't follow the development.

#### I need help!

see *Troubleshooting*.

#### Why are there no fancy effects?

The FileBrowser is about managing files. We think that you should prepare your files *before* uploading them to the server.

#### How do I upload to another server?

Use a custom storage engine, see https://docs.djangoproject.com/en/1.11/howto/custom-file-storage/.

#### Why do I need image-versions?

You need image-versions if your website is based on a *grid*.

#### Is the FileBrowser stable?

We've developed the FileBrowser for a couple of years and use it with almost all of our clients. That said, Grappelli is the more stable and mature application.

#### How can I contribute?

Help is very much needed and appreciated. Test the FileBrowser and submit feedback/patches.

#### Who develops the FileBrowser?

The FileBrowser is developed and maintained by Patrick Kranzlmüller & Axel Swoboda of vonautomatisch.

### 6.1.2 Troubleshooting

#### Check your setup

Please check if the problem is caused by your setup.

- Read *Quick start guide*.
- Check if the static/media-files are served correctly.
- Make sure you have removed all custom FileBrowser templates from all locations in `TEMPLATE_DIRS` or check that these templates are compatible with the FileBrowser.

#### Run the tests

Start the shell and type:

```
python manage.py test filebrowser
```

**Warning:** Please note that the tests will copy files to your filesystem.

#### Check issues

If your setup is fine, please check if your problem is a known issue.

- Take a look at all FileBrowser Issues (incuding closed) and search the FileBrowser Google-Group.

#### Add a ticket

If you think you've found a bug, please add a ticket.

- Try to describe your problem as precisely as possible.
- Tell us what you did in order to solve the problem.
- Tell us what version of the FileBrowser you are using.
- Tell us what version of Django you are using.
- Please do NOT add tickets if you're having problems with serving static/media-files (because this is not related to the FileBrowser).
- Please do NOT add tickets referring to Djangos trunk version.
- At best: add a patch.

---

**Note:** Be aware that we may close issues not following these guidlines without further notifications.

---

### 6.1.3 Translation

Translation is done via Transifex.

#### Supported Languages

see https://www.transifex.net/projects/p/django-filebrowser/resource/djangopo/

### 6.1.4 FileBrowser 3.11 Release Notes

FileBrowser 3.11 is compatible with Django 2.1 as well as Grappelli 2.12.

#### Updates

- Compatibility with Django 2.1 and Grappelli 2.12

#### Update from FileBrowser 3.10.x

- Update Django to 2.1 and check https://docs.djangoproject.com/en/dev/releases/2.1/
- Update Grappelli to 2.12.x
- Update FileBrowser to 3.11.x

## 6.2 Testing

Filebrowser is shipped with a minimal django project for testing.

Run the FileBrowser tests:

```
tox
```

**Warning:** Please note that the tests will copy files to your filesystem.

### 6.2.1 Travis

See https://travis-ci.org/sehmaschine/django-filebrowser.

## 6.3 Changelog

### 6.3.1 3.11.4 (not yet released)

### 6.3.2 3.11.3 (November 14th, 2019)

- FIXED: requirements installation.

### 6.3.3 3.11.2 (November 14th, 2019)

- FIXED: updated translations.
- FIXED: allow default string values with *FileBrowseField*.
- FIXED: use empty_values(s) with *FileBrowseField*. Makes sure that the field always returns the expected value.
- FIXED: selecting files based on *SELECT_FORMATS*.
- FIXED: changed admin_static to static, removed Django warning.
- IMPROVED: always show folders with pop-ups.
- IMPROVED: ensure that pillow is processed as a requirement at installation time.
- IMPROVED: support for TinyMCE v4.

### 6.3.4 3.11.1 (November 2nd, 2018)

- Compatibility with Django 2.1 and Grappelli 2.12

For further information, see *FileBrowser 3.11 Release Notes*.

CHAPTER 7

# Main Features

- Browse your media files with the admin interface.
- Multiple upload, including a progress bar.
- Automatic thumbnails.
- Image versions to fit your websites grid (esp. useful with adaptive/responsive layouts).
- Integration with TinyMCE.
- FileBrowseField to select images/documents.
- Signals for upload, rename and delete.
- Custom actions.
- Custom file storage engines.

CHAPTER 8

# Code

https://github.com/sehmaschine/django-filebrowser

CHAPTER 9

# Discussion

Use the FileBrowser Google Group to ask questions or discuss features.

CHAPTER 10

# Versions and Compatibility

**FileBrowser is always developed against the latest stable Django release and is NOT tested with Djangos trunk.**

- FileBrowser 3.11.3 (November 14th, 2019): Compatible with Django 2.1
- FileBrowser 3.10.2 (November 2nd, 2018): Compatible with Django 2.0
- FileBrowser 3.9.2 (November 2nd, 2018): Compatible with Django 1.11

Current development branches:

- FileBrowser 3.11.4 (Development Version for Django = 2.1, see Branch Stable/3.11.x)
- FileBrowser 3.10.3 (Development Version for Django = 2.0, see Branch Stable/3.10.x)
- FileBrowser 3.9.3 (Development Version for Django = 1.11, see Branch Stable/3.9.x)

Older versions are available at GitHub, but are not supported anymore. Support for 3.10.x and 3.9.x is limited to security issues and very important bugfixes.

# Index

## V



## W