Django Media Tree Documentation

Release 0.8.0

Samuel Luescher

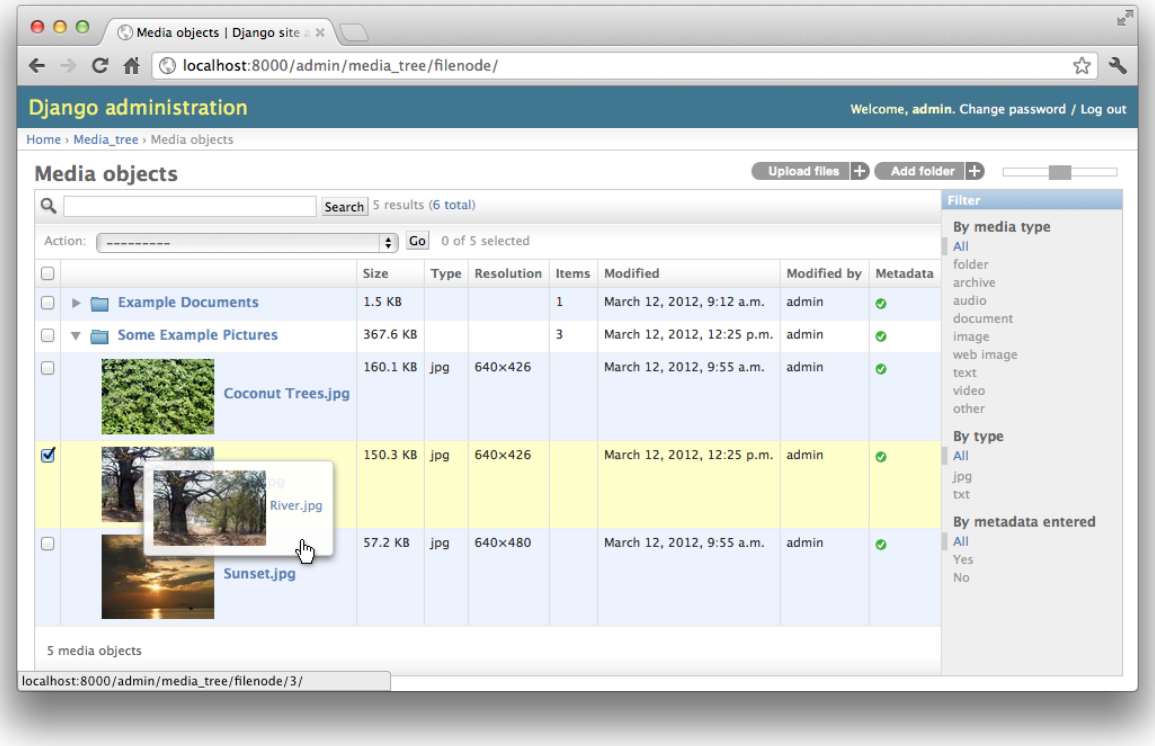
June 30, 2016

Contents

CHAPTER 1

Introduction

Django Media Tree is a Django app for managing your website's media files in a folder tree, and using them in your own applications.



Key features:

- Enables you to organize all of your site media in nested folders.
- Supports various media types (images, audio, video, archives etc).
- Extension system, enabling you to easily add special processing for different media types and extend the admin interface.
- Speedy AJAX-enhanced admin interface with drag & drop and dynamic resizing.
- Upload queue with progress indicators (using Fine Uploader).
- Add metadata to all media to improve accessibility of your web sites.

- Integration with Django CMS. Plugins include: image, slideshow, gallery, download list – create your own!

CHAPTER 2

The Media Tree application

2.1 Installing Media Tree

This install guide assumes you are familiar with Python and Django.

2.1.1 Dependencies

Make sure to install the following packages if you want to use Media Tree:

- Django >= 1.5
- South >= 0.8
- django-mptt > 0.4.2 (see *Note on django-mptt*)
- Pillow >= 2.3

Note: All required Python packages can easily be installed using pip (or, alternatively, easy_install).

2.1.2 Getting the code

For the latest stable version (recommended), use `pip`:

```
pip install django-media-tree
```

or download it from http://github.com/samluescher/django-media-tree and run the installation script:

```
python setup.py install
```

2.1.3 Demo project

A demo project is included for you to quickly test and evaluate Django Media Tree. It is recommended to use virtualenv for trying it out, as you'll be able to install all dependencies in isolation. Afer installing *virtualenv*, run the following commands to start the demo project:

```
mkdir django-media-tree-test && cd django-media-tree-test
virtualenv venv
source venv/bin/activate
```

```
curl -L https://github.com/samluescher/django-media-tree/archive/master.zip  \
  -o django-media-tree-master.zip && unzip django-media-tree-master
cd django-media-tree-master/demo_project
pip install -r requirements.txt
python manage.py collectstatic
python manage.py syncdb  # no need to create a superuser
python manage.py loaddata fixtures/initial_data.json
python manage.py runserver
```

Then open http://localhost:8000 in your web browser.

2.1.4 Basic setup

Please follow these steps to use Media Tree with your own application.

- In your project settings, add `mptt` and `media_tree` to the `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # ... your other apps here
    'mptt',
    'media_tree',
)
```

- Make sure your `STATIC_URL`, `STATIC_ROOT`, `MEDIA_URL` and `STATIC_ROOT` are properly configured.

  Note: Please refer to the Django documentation on how to configure your Django project to serve static files if you have not done that yet.

- If you are using `django.contrib.staticfiles` (recommended), just run the usual command to collect static files:

```
python manage.py collectstatic
```

  If you are **not** going to use the `staticfiles` app, you will have to copy the contents of the `static` folder to the location you are serving static files from.

- Create the database tables:

```
python manage.py syncdb
```

  Alternatively, if you are using South in your project, you'll have to use a slightly different command:

```
python manage.py syncdb --all
python migrate media_tree --fake
```

2.1.5 Configuring media backends (optional)

- If you want thumbnails to be generated – which will usually be the case – you need to install the appropriate media backend that takes care of this. Currently, easy-thumbnails is the only recommended and officially supported 3rd-party application.

  After you've installed the `easy_thumbnails` module, configure Media Tree to use it by defining `MEDIA_TREE_MEDIA_BACKENDS` in your project settings:

```
MEDIA_TREE_MEDIA_BACKENDS = (
    'media_tree.contrib.media_backends.easy_thumbnails.EasyThumbnailsBackend',
)
```

Note: In principle, Media Tree can work together with any other thumbnail generating app, provided that you write the appropriate media backend class to support it. Please have a look at one of the backends under `media_tree.contrib.media_backends` if you are interested in using your own specific 3rd-party app.

- **Optional**: Also add any Media Tree extensions that you are planning to use to your `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # ... your other apps here
    'media_tree.contrib.media_extensions.images.focal_point',
    'media_tree.contrib.media_extensions.zipfiles',
)
```

Note: See *Bundled extensions* for a list of default extensions included in the project.

2.1.6 Note on django-mptt

A version of `django-mptt` **newer than 0.4.2** is required because there is an issue with older versions not indenting the folder list correctly. **Either** install a recent version:

```
pip install django-mptt==0.5.1
```

or, if for some reason you can't install a recent version, you can resolve the situation by putting `legacy_mptt_support` in your `INSTALLED_APPS` **before** `mptt`. This will be deprecated in the future:

```
INSTALLED_APPS = (
  # ... your other apps here
  'media_tree.contrib.legacy_mptt_support', 'mptt', 'media_tree',
)
```

2.1.7 Installing icon sets (optional)

By default, Media Tree only comes with plain file and folder icons. If you would like to use custom icon sets that are more appropriate for your specific media types, you can install them like a Django application.

The following ready-to-use modules contain some nice icons:

- Teambox Icons

You will need to configure Media Tree to use an icon set as follows.

- In order to install an icon set, simply add the respective module to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # ... your other apps here
    'my_custom_icon_set',
)
```

- If you are using `django.contrib.staticfiles` (recommended), just run the usual command to collect static files:

```
./manage.py collectstatic
```

If you are **not** using the `staticfiles` app, copy the contents of the `static` folder to the static root of your project.

- Define `MEDIA_TREE_ICON_DIRS` in your project settings, and add the static path containing the new icon files, e.g.:

```
MEDIA_TREE_ICON_DIRS = (
    'my_custom_icons/64x64px', # the new folder under your static root
    'media_tree/img/icons/mimetypes', # default icon folder
)
```

Note: You can add several icon sets to this tuple, and for each media file the first appropriate icon that is encountered will be used. Please notice that on the last line we are specifying the default icon location, which will be used as a fallback in case no appropriate icon is found in one of the custom sets.

2.2 Admin interface overview

2.3 Models and Managers

2.4 Configuring Media Tree

The following settings can be specified in your Django project's settings module.

`MEDIA_TREE_STORAGE` File storage class to be used for any file-related operations when dealing with media files.

This is not set by default, meaning that Django's `DEFAULT_FILE_STORAGE` will be used. If you need to implement your custom storage, please refer to the relevant Django documentation on that setting and on file storage in general.

`MEDIA_TREE_DELETE_FROM_STORAGE_ON_OVERWRITE`

Determines whether existing files should be deleted from storage when the corresponding object is overwritten with a new file. This prevents the creation of orphaned files. As an added benefit in the case of Django's *FileSystemStorage*, if this setting is *True* and a file is supposed to be overwritten with a new file that has the same name, that name remains permanent because the old file will be deleted before the object is saved. If *False*, a random hash would be added to the new file's name due to the existing file already occupying the desired name on disk. “”“

Default: `True`

`MEDIA_TREE_MEDIA_BACKENDS` A tuple of media backends for thumbnail generation and other media-related tasks, i.e. a list of wrappers for the 3rd-party applications that take care of them.

Note: Please refer to the *installation instructions* for information on how to configure supported media backends.

For general information on media backends, see *Using FileNodes in templates* for more information.

`MEDIA_TREE_MEDIA_BACKEND_DEBUG` Specifies whether exceptions caused by media backends, such as `ThumbnailError`, should be raised or silently ignored.

Default: `settings.DEBUG`

MEDIA_TREE_LIST_DISPLAY A tuple containing the columns that should be displayed in the `FileNodeAdmin`. Note that the `browse_controls` column is necessary for the admin to function properly.

MEDIA_TREE_LIST_FILTER A tuple containing the fields that nodes can be filtered by in the `FileNodeAdmin`.

MEDIA_TREE_SEARCH_FIELDS A tuple containing the fields that nodes can be searched by in the `FileNodeAdmin`.

MEDIA_TREE_UPLOAD_SUBDIR Default: `'upload'`

The name of the folder under your `MEDIA_ROOT` where media files are stored.

MEDIA_TREE_PREVIEW_SUBDIR Default: `'upload/_preview'`

The name of the folder under your `MEDIA_ROOT` where cached versions of media files, e.g. thumbnails, are stored.

MEDIA_TREE_ICON_DIRS Default:

```
(
    'media_tree/img/icons/mimetypes',
)
```

A tuple containing all icon directories. See *Installing icon sets (optional)* for more information.

MEDIA_TREE_THUMBNAIL_SIZES A dictionary of default thumbnail sizes. You can pass the dictionary key to the `thumbnail` templatetag instead of a numeric size.

Default:

```
{
    'small': (80, 80),
    'default': (100, 100),
    'medium': (250, 250),
    'large': (400, 400),
    'full': None, # None means: use original size
}
```

MEDIA_TREE_ALLOWED_FILE_TYPES A whitelist of file extensions that can be uploaded. By default, this is a comprehensive list of many common media file extensions that generally shouldn't pose a security risk.

> **Warning:** Just because a file extension may be considered "safe", there is absolutely no guarantee that a skilled attacker couldn't find an exploit. You should only allow people you trust to upload files to your webserver. Be careful when adding potentially unsafe file extensions to this setting, such as executables or scripts, as this possibly opens a door to attackers.

MEDIA_TREE_THUMBNAIL_EXTENSIONS Default: `('jpg', 'png')`

A tuple of image extensions used for thumbnail files. Note that `png` is in there since you typically might want to preserve the file type of PNG images instead of converting them to JPG.

MEDIA_TREE_FILE_SIZE_LIMIT Default: `1000000000 # 1 GB`

Maximum file size for uploaded files.

MEDIA_TREE_GLOBAL_THUMBNAIL_OPTIONS A dictionary of options that should be applied by default when generating thumbnails. You might use this, for instance, to sharpen all thumbnails:

```
MEDIA_TREE_GLOBAL_THUMBNAIL_OPTIONS = {
    'sharpen': True
}
```

2.5 FileNode Utility functions

media_tree.utils.filenode.**get_file_link**(*node*, *use_metadata=False*, *include_size=False*, *include_extension=False*, *include_icon=False*, *href=None*, *extra_class=''*, *extra=''*)

Returns a formatted HTML link tag to the FileNode's file, optionally including some meta information about the file.

media_tree.utils.filenode.**get_merged_filenode_list**(*nodes*, *filter_media_types=None*, *exclude_media_types=None*, *filter=None*, *ordering=None*, *processors=None*, *max_depth=None*, *max_nodes=None*)

Almost the same as *get_nested_filenode_list()*, but returns a flat (one-dimensional) list. Using the same QuerySet as in the example for *get_nested_filenode_list*, this method would return:

```
[
    <FileNode: Empty folder>,
    <FileNode: Photo folder>,
    <FileNode: photo1.jpg>,
    <FileNode: photo2.jpg>,
    <FileNode: photo3.jpg>,
    <FileNode: file.txt>
]
```

media_tree.utils.filenode.**get_nested_filenode_list**(*nodes*, *filter_media_types=None*, *exclude_media_types=None*, *filter=None*, *ordering=None*, *processors=None*, *max_depth=None*, *max_nodes=None*)

Returns a nested list of nodes, applying optional filters and processors to each node. Nested means that the resulting list will be multi-dimensional, i.e. each item in the list that is a folder containing child nodes will be followed by a sub-list containing those child nodes.

Example of returned list:

```
[
    <FileNode: Empty folder>,
    <FileNode: Photo folder>,
        [<FileNode: photo1.jpg>, <FileNode: photo2.jpg>,
         <FileNode: another subfolder>],
            [<FileNode: photo3.jpg>]
    <FileNode: file.txt>
]
```

You can use this list in conjunction with Django's built-in list template filters to output nested lists in templates:

```
{{ some_nested_list|unordered_list }}
```

Using the FileNode structure from the example, the above line would result in the following output:

```
<ul>
    <li>Empty folder</li>
    <li>Photo folder
        <ul>
            <li>photo1.jpg</li>
            <li>photo2.jpg</li>
            <li>another subfolder
                <ul>
```

```
                        <li>photo3.jpg</li>
                    </ul>
                </li>
            </ul>
        </li>
        <li>file.txt</li>
    </ul>
```

Parameters

- **nodes** – A QuerySet or list of FileNode objects
- **filter_media_types** – A list of media types to include in the resulting list, e.g. `media_types.DOCUMENT`
- **exclude_media_types** – A list of media types to exclude from the resulting list
- **filter** – A dictionary of kwargs to be applied with `QuerySet.filter()` if `nodes` is a QuerySet
- **processors** – A list of callables to be applied to each node, e.g. `force_unicode` if you want the list to contain strings instead of FileNode objects
- **max_depth** – Can be used to limit the recursion depth (unlimited by default)
- **max_nodes** – Can be used to limit the number of items in the resulting list (unlimited by default)

2.6 Bundled extensions

Media Tree contains a few useful extensions in its `contrib` module. Since some of these extensions modify the `FileNode` model, you should install them before you run `syncdb` for the first time.

2.6.1 focal_point

The *focal_point* extension allows you to drag a marker on image thumbnails while editing, thus specifying the most relevant portion of the image. You can then use these coordinates in templates for image cropping.

- To install it, add the extension module to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # ... your apps here ...
    'media_tree.contrib.media_extensions.images.focal_point'
)
```

- If you are not using `django.contrib.staticfiles`, copy the contents of the `static` folder to the static root of your project. If you are using the `staticfiles` app, just run the usual command to collect static files:

```
$ ./manage.py collectstatic
```

Note: This extension adds the fields `focal_x` and `focal_y` to the `FileNode` model. You are going to have to add these fields to the database table yourself by modifying the `media_tree_filenode` table with a database client, **unless you installed it before running** `syncdb`).

2.6.2 zipfiles

The *zipfiles* extension adds support for ZIP archives to the `FileNodeAdmin`. If it is installed, you can select files and folders in the admin and download them as a ZIP archive.

To install it, add the extension module to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # ... your apps here ...
    'media_tree.contrib.media_extensions.zipfiles'
)
```

2.7 Management Commands

You can use the following management commands to assist you with media file management.

2.7.1 Orphaned files

Use the following command to list all orphaned files, i.e. media files existing in storage that are not in the database:

```
manage.py mediaorphaned
```

Use the following command to **delete** all orphaned files:

```
manage.py mediaorphaned --delete
```

2.7.2 Media cache

Use the following command to list all media cache files, such as thumbnails:

```
manage.py mediacache
```

Use the following command to **delete** all media cache files:

```
manage.py mediacache --delete
```

CHAPTER 3

Extending und using Media Tree with other applications

Your choices range from implementing file listing and detail views based on the bundled generic view classes, extending Media Tree itself and its admin interface, or writing custom plugins for use with your own applications.

3.1 Fields and forms

There are a number of field classes for conveniently using `FileNode` objects in your own Django applications.

The following example model contains a `ForeignKey` field linking to a `FileNode` object that is associated to a document file. Notice the parameters specifying which media types will be validated, and which should be visible in the widget:

```
from media_tree.fields import FileNodeForeignKey
from media_tree import media_types
from media_tree.models import FileNode
from django.db import models

class MyModel(models.Model):
    document = FileNodeForeignKey(allowed_media_types=(media_types.DOCUMENT,),
        null=True, limit_choices_to={'media_type__in':
        (FileNode.FOLDER, media_types.DOCUMENT)})
```

The following example model will allow the user to select a `FileNode` object associated to an image file:

```
from media_tree.fields import ImageFileNodeForeignKey
from django.db import models

class MyModel(models.Model):
    image_node = ImageFileNodeForeignKey(null=True)
```

The following example form will allow the user to select files that are under a specific parent folder named “Projects”:

```
from media_tree.models import FileNode
from media_tree.forms import FileNodeChoiceField
from django import forms

class MyForm(forms.Form):
    file_node = FileNodeChoiceField(queryset=FileNode.objects.get(
        name="Projects", node_type=FileNode.FOLDER).get_descendants())
```

For your own applications, the following field classes are available:

3.2 Using FileNodes in templates

Although Media Tree is designed to be agnostic of the module you use to generate image versions and thumbnails, it includes some tags to assist you with generating thumbnails from `FileNode` objects, since this is one of the most common tasks when working with image files in web applications.

3.2.1 A word about Media Backends

Media Tree's template tags do not use an imaging toolkit directly, but an abstraction class designed to wrap the actual image manipulation handled by a third-party module (such as `easy_thumbnails` or `sorl.thumbnail`, to name two popular choices).

The advantage of wrapping thumbnail generation like this is that Media Tree does not need to depend on a specific image generation library, with the additional benefit that you can just use the abstract template tags in your templates and switch to another `MediaBackend` at any time.

3.2.2 Thumbnail Template Tags

3.3 Class-based generic views

The module `media_tree.contrib.views` contains class-based generic views that enable you to access `FileNode` objects through public URLs. Please see below for specific examples. Of course you can also extend the generic view classes to create views that suit your specific requirements.

Note: As with any public views, you may want to restrict the objects that should be publicly visible by passing an appropriately filtered `queryset` when implementing a view. For instance, you may not want users to see the internal folder structure of your `FileNode` objects, hence using a `FileNodeListingView` with a QuerySet such as `FileNode.objects.all()` would be a bad idea.

3.3.1 List Views

3.3.2 Detail Views

3.4 Extending Django Media Tree

There are several ways in which you may want to add functionality to Media Tree. Suppose you need to add support for a specific image format, or you need custom maintenance actions in the admin, or you might need to add some Javascript or CSS code to the FileNode form. For each of these cases, there is a so-called **extender class**.

3.4.1 Overview

The extender base classes provided by Media Tree are `ModelExtender`, `AdminExtender` and `FormExtender`, and by subclassing them you create your custom extenders. The structure of extender classes is similar to that of a regular Django `Model`, `ModelAdmin`, or `Form` class, respectively, meaning that you can define several attributes such as model Fields or form Media, and they will be added to Media Tree during runtime.

Note: You can *package and install* your extender classes as a regular Django application module, and have Media Tree auto-discover installed extensions by providing a `media_extension.py` module. An application containing one or more extenders and a `media_extension.py` that registers them is what is called a **Media Tree extension**.

Media Tree already comes with some exemplary extensions in its `contrib.media_extensions` module. You should inspect these examples in order to get an idea of how to build an extension. There is also a tutorial below that should help you with creating your own extension.

3.4.2 Extender bases

An extender is created by subclassing one of the following base classes:

Extending the FileNode model

Extending the FileNode admin

Extending forms

3.4.3 Registering and installing Media Tree extensions

Each extension module is a regular Django application that is installed by putting the application in the `INSTALLED_APPS` setting.

An extension needs to contain a `media_extension.py` module that registers all extenders that the extension module contains:

Example of an `extension.py` file:

```
from media_tree import extension

class SomeModelExtender(extension.ModelExtender):
    """Example extender"""
    pass

extension.register(SomeModelExtender)
```

Notice that on the last line the extender is registered by calling the function `media_tree.extension.register()`.

3.4.4 Tutorial extension: Geotagging Photos

Assume you are using landscape photographs on your website, and in the FileNode admin you would like to be able to enter the latitude and longitude of the place where they were taken. This is called *geotagging*.

Getting started

The first step is to create a Django application that serves as the container for our new extender classes. You can do this as usual on the command line from your project folder:

```
django-admin startapp media_tree_geotagging
cd media_tree_geotagging
touch media_extension.py
```

Notice that on the last line we created a file called `media_extension.py`. Media Tree will scan all `INSTALLED_APPS` for such a file, so that all installed extensions will be auto-disovered.

We can delete most of the other files that the `startapp` command created, such as `models.py`, as we are probably not going to need them.

Extending the Model

Now you can create the model extender in the file `media_extension.py`, subclassing the parent class provided by Media Tree:

```
from media_tree import extension
from django.db import models

class GeotaggingModelExtender(extension.ModelExtender):
    lat = models.FloatField('latitude', null=True, blank=True)
    lng = models.FloatField('longitude', null=True, blank=True)

extension.register(GeotaggingModelExtender)
```

This class looks similar to a regular Model, but it does not have its own database table – instead, its fields are added to the `FileNode` class when you restart the development server.

Note: This extension adds the fields `lat` and `lng` to the `FileNode` model. You are going to have to add these fields to the database table yourself by modifying the `media_tree_filenode` table with a database client, **unless you installed it before running** `syncdb`).

Extending the form

Of course we want to be able to edit our two new fields in the admin, so we need to create a form extender and add a new fieldset. We do this by adding a new class to `media_extension.py`:

```
class GeotaggingFormExtender(extension.FormExtender):

    class Meta:
        fieldsets = [
            ('Geotagging', {
                'fields': ['lat', 'lng'],
                'classes': ['collapse']
            })
        ]

extension.register(GeotaggingFormExtender)
```

Installing the extension

After you have created the database fields, you can install the extension by adding it to the `INSTALLED_APPS` in your project's settings file:

```
INSTALLED_APPS = (
    # ... your apps here ...
    'media_tree',
    'media_tree_geotagging'
)
```

Adding an Admin Action

Let's assume you have a content editor on staff, and this person's job is to check if photographs were geotagged, and to notify the photographer of the ones that aren't. We can simplify this task by adding an admin action to the FileNode admin.

With this extender, the editor will be able to check the checkboxes next to image files, have them checked automatically to see if they are not yet geotagged, and email the photographer the admin links to those FileNode objects.

As you may be assuming by now, we create an admin extender in `media_extension.py`:

```
from django.core.mail import send_mail

class GeotaggingAdminExtender(extension.AdminExtender):

    def notify_of_non_geotagged(modeladmin, request, queryset):
        non_geotagged_links = []
        for node in queryset:
            # Check if node is JPG and not geotagged:
            if node.extension == 'jpg':
                if not node.lat or not node.lng:
                    non_geotagged_links.append(node.get_admin_url())
        # Send email with admin links for these nodes, and message
        # current user about status of the action.
        if len(non_geotagged_links):
            message = '\n'.join(non_geotagged_links) + '\n\nThanks!'
            send_mail('Please geotag these files', message,
                'from@example.com', ['to@example.com'])
            modeladmin.message_user(request, 'Notification sent for'  \
                + ' %i non-geotagged JPGs.' % len(non_geotagged_links))
        else:
            modeladmin.message_user(request, 'All selected images appear'  \
                +' to be OK.')
    notify_of_non_geotagged.short_description =  \
        'Notify photographer if selected JPGs are not geotagged'

    actions = [notify_of_non_geotagged]
```

This last example is a bit more verbose, but you will notice that it just contains one method with the exact same signature like a regular Django admin action, and on the last line we are specifying the list of actions that this extender will contribute to the FileNode admin. Also, we are giving the method a `short_description` that will appear in the drop-down menu above the list displaying all of our FileNodes.

And that's it! We are now able to geotag images in the Django admin.

Adding Form Media

Of course it would be great if we had a map widget in the form where we can just drop a pin on the location of the photograph. Creating such a widget is beyond the scope of this tutorial, but if we had created a Javascript containing the code that implements such a widget, we could easily add this file by adding a `Media` class to our form extender:

```
class GeotaggingFormExtender(extension.AdminExtender):

    class Media:
        js = (
            'map_widget.js',
        )
```

```
    # ...
```

This Media definition is merged with the default media loaded for the FileNode form, and we can use it to load any code or CSS files required by our hypothetical map widget.

Conclusion

Using this extension system, you can change many aspects of how Media Tree behaves. There are more attributes and also **signals** that you can define in your extenders than the ones described in this tutorial. Code away and, please, share your extensions with the Interested Public!

3.4.5 Tutorial extension: Creating an icon set

Icon sets are also packaged as Django applications, and creating a custom set is rather easy. Basically, an icon set is a Python module containing nothing but an empty `__init__.py` and a `static` folder with the respective image files. Here's an example of how that could look like:

```
my_custom_audio_icon_set
    __init__.py
    static
        audio_icons
            audio.png
            ogg.png
            mp3.png
```

Note that this package contains three icons: One for generic audio files and one for either `OGG` or `MP3` files.

Note: When displaying a file icon, Media Tree will scan all installed icon sets for an icon that is named like the media file's extension (e.g. `mp3.png`), then for one named like its mimetype (e.g. `audio/x-mpeg.png`), then for the mime supertype (e.g. `audio.png`). Icon discovery is handled by a class called `MimetypeStaticIconFileFinder`, which by default only finds `PNG` files.

To install this icon set, simply add `my_custom_audio_icon_set` to your `INSTALLED_APPS`, collect its static files, and configure the new icon folder using the `MEDIA_TREE_ICON_DIRS` setting. See *Installing icon sets (optional)* for more detailed instructions.

3.5 Creating custom plugins for use with 3rd-party applications

3.5.1 How to create custom plugins

Django Media Tree comes with some generic View classes and Mixins that make it relatively easy to use `FileNode` objects with your own applications.

The following pseudo code should give you an idea of how to implement your own custom plugin that will render a file listing and work together with the 3rd-party application of your choice. It loosely looks like a Django CMS plugin. Please notice that the `render()` method is passed an `options_instance`, which can be a dictionary or an object with attributes to initialize the generic View class we are using, which is `FileNodeListingView` in this case. See *Class-based generic views* for more information on the View classes themselves:

```
from media_tree.contrib.views.listing import FileNodeListingMixin
from third_party_app import YourPluginSuperclass
from django.shortcuts import render_to_response

# Notice we are subclassing our third-party plugin class,
# as well as the FileNodeListingMixin
class CustomFileNodeListingPlugin(YourPluginSuperclass, FileNodeListingMixin):

    # Assuming render() is a standard method of YourPluginSuperclass
    def render(self, request, options_instance):

        # Get the generic view class using the method inherited from
        # the Mixin class.
        # Notice that get_detail_view() is inherited from the
        # FileNodeListingMixin. We are also passing our options model
        # instance for configuring the view instance.
        view = self.get_detail_view(request,
            queryset=options_instance.selected_folders,
            opts=options_instance)

        # Get the template context as generated by the View class
        context_data = view.get_context_data()

        # Render with custom template
        return render_to_response('listing.html', context_data)
```

This is what our model classes (namely the class of the `options_instance` above) might look like:

```
from django.db import Models
from media_tree.fields import FileNodeForeignKey

class PluginOptions(models.Model):
    # These field names are derived from
    # media_tree.contrib.views.list.FileNodeListingView.
    list_max_depth = models.IntegerField()
    include_descendants = models.BooleanField()

class SelectedFolder(models.Model):
    plugin = models.ForeignKey(PluginOptions)
    folder = FileNodeForeignKey()
```

The first class contains our plugin option fields. Notice that when calling the `get_detail_view()` or `get_view()` methods provided by the `FileNodeListingMixin` and passing it an instance of this model, any fields that match attributes of the view object returned will be used to initialized the view object.

The second class creates a relationship between the options model and the `FileNode` model, i.e. you will be able to link `FileNode` objects to plugins.

3.5.2 View Mixins

View Mixins are classes that add methods useful for interfacing with Media Tree's generic view classes to your custom plugin classes, as demonstrated in the above example. You can use Mixins as superclasses for your custom plugins when interfacing with third-party applications, such as Django CMS. Please take a look at *How to create custom plugins* for more information.

Basically, a Mixin classes adds methods to your own class (which is subclassing a Mixin) for instantiating View classes. All attributes of your own class that also exist in the View class will be used to initialize View instances.

For instance, if your custom class has an attribute `template_name`, and an attribute with the same name also exists in the View class, then the View instance's `template_name` attribute will be set accordingly.

Please refer to *Class-based generic views* for an overview of attributes you can define.

3.6 Django CMS Plugins

The module `media_tree.contrib.cms_plugins` contains a number of plugins for using `FileNode` objects on pages created with Django CMS.

3.6.1 Installation

For optimum admin functionality when using these plugins, you should put `media_tree.contrib.cms_plugins` in your installed apps, and run `manage.py collectstatic`.

If you are not using the `staticfiles` app, you have to manually copy the contents of the `static` folder to your static root.

Note: Of course you can also create your own models and plugins using `FileNode` objects. Please take a look at *Fields and forms* and *How to create custom plugins* for more information on how to integrate Media Tree with your own applications.

3.6.2 Plugin: File listing

This plugin allows you to put a file listing on a page, displaying download links for the selected `FileNode` objects in a folder tree.

The folder tree that is rendered does not have to be identical to the actual tree in your media library. Instead, you can group arbitrary nodes, or output a merged (flat) list.

Installation

To use this plugin, put `media_tree.contrib.cms_plugins.media_tree_listing` in your installed apps, and run `manage.py syncdb`.

Template

Override the template `cms/plugins/media_tree_listing.html` if you want to customize the output. Please take a look at the default template for more information.

3.6.3 Plugin: Image

This plugin allows you to put a single picture on a page, as a figure complete with caption and other metadata.

Installation

To use this plugin, put `media_tree.contrib.cms_plugins.media_tree_image` in your installed apps, and run `manage.py syncdb`.

Template

Override the template `cms/plugins/media_tree_image.html` if you want to customize the output. Please take a look at the default template for more information.

By default, images are rendered to the output using the template `media_tree/filenode/includes/figure.html`, which includes captions.

3.6.4 Plugin: Slideshow

This plugin allows you to put a slideshow on a page, automatically displaying the selected image files with customizable transitions and intervals.

Installation

To use this plugin, put `media_tree.contrib.cms_plugins.media_tree_slideshow` in your installed apps, and run `manage.py syncdb`.

Template

Override the template `cms/plugins/media_tree_slideshow.html` if you want to customize the output. Please take a look at the default template for more information.

By default, images are rendered to the output using the template `media_tree/filenode/includes/figure.html`, which includes captions.

Note: The default template requires you to include jQuery in your pages, since it uses the jQuery Cycle Plugin (bundled) for image transitions.

3.6.5 Plugin: Gallery

This plugin allows you to put an image gallery on a page. Galleries can include nested folder structures or display merged (flat) compositions of all images in a range of subfolders. Pictures can be browsed or auto-played.

Installation

To use this plugin, put `media_tree.contrib.cms_plugins.media_tree_gallery` in your installed apps, and run `manage.py syncdb`.

Template

Override the template `cms/plugins/media_tree_gallery.html` if you want to customize the output. Please take a look at the default template for more information.

By default, images are rendered to the output using the template `media_tree/filenode/includes/figure.html`, which includes captions.

Note: The default template requires you to include jQuery in your pages, since it uses the jQuery Cycle Plugin (bundled) for image transitions.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

m

G

M