django-tagging Documentation

Release 0.4.5

Fantomas42

February 26, 2017

Contents

A generic tagging application for Django projects, which allows association of a number of tags with any Django model instance and makes retrieval of tags simple.

CHAPTER 1

Installation

Installing an official release

Official releases are made available from https://pypi.python.org/pypi/django-tagging/

Source distribution

Download the a distribution file and unpack it. Inside is a script named `setup.py`. Enter this command:

```
$ python setup.py install
```

...and the package will install automatically.

More easily with **pip**:

```
$ pip install django-tagging
```

Installing the development version

Alternatively, if you'd like to update Django Tagging occasionally to pick up the latest bug fixes and enhancements before they make it into an official release, clone the git repository instead. The following command will clone the development branch to `django-tagging` directory:

```
git clone git@github.com:Fantomas42/django-tagging.git
```

Add the resulting folder to your PYTHONPATH or symlink (junction, if you're on Windows) the `tagging` directory inside it into a directory which is on your PYTHONPATH, such as your Python installation's `site-packages` directory.

You can verify that the application is available on your PYTHONPATH by opening a Python interpreter and entering the following commands:

```
>>> import tagging
>>> tagging.__version__
0.4.dev0
```

When you want to update your copy of the Django Tagging source code, run the command `git pull` from within the `django-tagging` directory.

Caution: The development version may contain bugs which are not present in the release version and introduce backwards-incompatible changes.
If you're tracking git, keep an eye on the CHANGELOG before you update your copy of the source code.

Using Django Tagging in your applications

Once you've installed Django Tagging and want to use it in your Django applications, do the following:

1. Put `'tagging'` in your `INSTALLED_APPS` setting.
2. Run the command `manage.py migrate`.

The `migrate` command creates the necessary database tables and creates permission objects for all installed apps that need them.

That's it!

CHAPTER 2

Settings

Some of the application's behaviour can be configured by adding the appropriate settings to your project's settings file.

The following settings are available:

FORCE_LOWERCASE_TAGS

Default: `False`

A boolean that turns on/off forcing of all tag names to lowercase before they are saved to the database.

MAX_TAG_LENGTH

Default: `50`

An integer which specifies the maximum length which any tag is allowed to have. This is used for validation in the `django.contrib.admin` application and in any forms automatically generated using `ModelForm`.

CHAPTER 3

Registering your models

Your Django models can be registered with the tagging application to access some additional tagging-related features.

Note: You don't *have* to register your models in order to use them with the tagging application - many of the features added by registration are just convenience wrappers around the tagging API provided by the `Tag` and `TaggedItem` models and their managers, as documented further below.

The `register` function

To register a model, import the `tagging.registry` module and call its `register` function, like so:

```
from django.db import models

from tagging.registry import register

class Widget(models.Model):
    name = models.CharField(max_length=50)

register(Widget)
```

The following argument is required:

`model` The model class to be registered.

An exception will be raised if you attempt to register the same class more than once.

The following arguments are optional, with some recommended defaults - take care to specify different attribute names if the defaults clash with your model class' definition:

`tag_descriptor_attr` The name of an attribute in the model class which will hold a tag descriptor for the model. Default: `'tags'`

See *TagDescriptor* below for details about the use of this descriptor.

`tagged_item_manager_attr` The name of an attribute in the model class which will hold a custom manager for accessing tagged items for the model. Default: `'tagged'`.

See *ModelTaggedItemManager* below for details about the use of this manager.

TagDescriptor

When accessed through the model class itself, this descriptor will return a `ModelTagManager` for the model. See *ModelTagManager* below for more details about its use.

When accessed through a model instance, this descriptor provides a handy means of retrieving, updating and deleting the instance's tags. For example:

```
>>> widget = Widget.objects.create(name='Testing descriptor')
>>> widget.tags
[]
>>> widget.tags = 'toast, melted cheese, butter'
>>> widget.tags
[<Tag: butter>, <Tag: melted cheese>, <Tag: toast>]
>>> del widget.tags
>>> widget.tags
[]
```

ModelTagManager

A manager for retrieving tags used by a particular model.

Defines the following methods:

- `get_queryset()` – as this method is redefined, any `QuerySets` created by this model will be initially restricted to contain the distinct tags used by all the model's instances.
- `cloud(*args, **kwargs)` – creates a list of tags used by the model's instances, with `count` and `font_size` attributes set for use in displaying a tag cloud.

  See the documentation on `Tag`'s manager's *cloud_for_model method* for information on additional arguments which can be given.
- `related(self, tags, *args, **kwargs)` – creates a list of tags used by the model's instances, which are also used by all instance which have the given `tags`.

  See the documentation on `Tag`'s manager's *related_for_model method* for information on additional arguments which can be given.
- `usage(self, *args, **kwargs))` – creates a list of tags used by the model's instances, with optional usages counts, restriction based on usage counts and restriction of the model instances from which usage and counts are determined.

  See the documentation on `Tag`'s manager's *usage_for_model method* for information on additional arguments which can be given.

Example usage:

```
# Create a ``QuerySet`` of tags used by Widget instances
Widget.tags.all()

# Retrieve a list of tags used by Widget instances with usage counts
Widget.tags.usage(counts=True)

# Retrieve tags used by instances of WIdget which are also tagged with
# 'cheese' and 'toast'
Widget.tags.related(['cheese', 'toast'], counts=True, min_count=3)
```

`ModelTaggedItemManager`

A manager for retrieving model instance for a particular model, based on their tags.

- `related_to(obj, queryset=None, num=None)` – creates a list of model instances which are related to `obj`, based on its tags. If a `queryset` argument is provided, it will be used to restrict the resulting list of model instances.

  If `num` is given, a maximum of `num` instances will be returned.

- `with_all(tags, queryset=None)` – creates a `QuerySet` containing model instances which are tagged with *all* the given tags. If a `queryset` argument is provided, it will be used as the basis for the resulting `QuerySet`.
- `with_any(tags, queryset=None)` – creates a `QuerySet` containing model instances which are tagged with *any* the given tags. If a `queryset` argument is provided, it will be used as the basis for the resulting `QuerySet`.

CHAPTER 4

Tags

Tags are represented by the `Tag` model, which lives in the `tagging.models` module.

API reference

Fields

`Tag` objects have the following fields:

- `name` – The name of the tag. This is a unique value.

Manager functions

The `Tag` model has a custom manager which has the following helper methods:

- `update_tags(obj, tag_names)` – updates tags associated with an object.

  `tag_names` is a string containing tag names with which `obj` should be tagged.

  If `tag_names` is `None` or `''`, the object's tags will be cleared.

- `add_tag(obj, tag_name)` – associates a tag with an an object.

  `tag_name` is a string containing a tag name with which `obj` should be tagged.

- `get_for_object(obj)` – returns a `QuerySet` containing all `Tag` objects associated with `obj`.

- `usage_for_model(model, counts=False, min_count=None, filters=None)` – returns a list of `Tag` objects associated with instances of `model`.

  If `counts` is `True`, a `count` attribute will be added to each tag, indicating how many times it has been associated with instances of `model`.

  If `min_count` is given, only tags which have a `count` greater than or equal to `min_count` will be returned. Passing a value for `min_count` implies `counts=True`.

  To limit the tags (and counts, if specified) returned to those used by a subset of the model's instances, pass a dictionary of field lookups to be applied to `model` as the `filters` argument.

- `related_for_model(tags, Model, counts=False, min_count=None)` – returns a list of tags related to a given list of tags - that is, other tags used by items which have all the given tags.

  If `counts` is `True`, a `count` attribute will be added to each tag, indicating the number of items which have it in addition to the given list of tags.

If `min_count` is given, only tags which have a `count` greater than or equal to `min_count` will be returned. Passing a value for `min_count` implies `counts=True`.

- `cloud_for_model(Model, steps=4, distribution=LOGARITHMIC, filters=None, min_count=None)` – returns a list of the distinct `Tag` objects associated with instances of `Model`, each having a `count` attribute as above and an additional `font_size` attribute, for use in creation of a tag cloud (a type of weighted list).

  `steps` defines the number of font sizes available - `font_size` may be an integer between `1` and `steps`, inclusive.

  `distribution` defines the type of font size distribution algorithm which will be used - logarithmic or linear. It must be either `tagging.utils.LOGARITHMIC` or `tagging.utils.LINEAR`.

  To limit the tags displayed in the cloud to those associated with a subset of the Model's instances, pass a dictionary of field lookups to be applied to the given Model as the `filters` argument.

  To limit the tags displayed in the cloud to those with a `count` greater than or equal to `min_count`, pass a value for the `min_count` argument.

- `usage_for_queryset(queryset, counts=False, min_count=None)` – Obtains a list of tags associated with instances of a model contained in the given queryset.

  If `counts` is True, a `count` attribute will be added to each tag, indicating how many times it has been used against the Model class in question.

  If `min_count` is given, only tags which have a `count` greater than or equal to `min_count` will be returned.

  Passing a value for `min_count` implies `counts=True`.

Basic usage

Tagging objects and retrieving an object's tags

Objects may be tagged using the `update_tags` helper function:

```
>>> from shop.apps.products.models import Widget
>>> from tagging.models import Tag
>>> widget = Widget.objects.get(pk=1)
>>> Tag.objects.update_tags(widget, 'house thing')
```

Retrieve tags for an object using the `get_for_object` helper function:

```
>>> Tag.objects.get_for_object(widget)
[<Tag: house>, <Tag: thing>]
```

Tags are created, associated and unassociated accordingly when you use `update_tags` and `add_tag`:

```
>>> Tag.objects.update_tags(widget, 'house monkey')
>>> Tag.objects.get_for_object(widget)
[<Tag: house>, <Tag: monkey>]
>>> Tag.objects.add_tag(widget, 'tiles')
>>> Tag.objects.get_for_object(widget)
[<Tag: house>, <Tag: monkey>, <Tag: tiles>]
```

Clear an object's tags by passing `None` or `''` to `update_tags`:

```
>>> Tag.objects.update_tags(widget, None)
>>> Tag.objects.get_for_object(widget)
[]
```

Retrieving tags used by a particular model

To retrieve all tags used for a particular model, use the `get_for_model` helper function:

```
>>> widget1 = Widget.objects.get(pk=1)
>>> Tag.objects.update_tags(widget1, 'house thing')
>>> widget2 = Widget.objects.get(pk=2)
>>> Tag.objects.update_tags(widget2, 'cheese toast house')
>>> Tag.objects.usage_for_model(Widget)
[<Tag: cheese>, <Tag: house>, <Tag: thing>, <Tag: toast>]
```

To get a count of how many times each tag was used for a particular model, pass in `True` for the `counts` argument:

```
>>> tags = Tag.objects.usage_for_model(Widget, counts=True)
>>> [(tag.name, tag.count) for tag in tags]
[('cheese', 1), ('house', 2), ('thing', 1), ('toast', 1)]
```

To get counts and limit the tags returned to those with counts above a certain size, pass in a `min_count` argument:

```
>>> tags = Tag.objects.usage_for_model(Widget, min_count=2)
>>> [(tag.name, tag.count) for tag in tags]
[('house', 2)]
```

You can also specify a dictionary of field lookups to be used to restrict the tags and counts returned based on a subset of the model's instances. For example, the following would retrieve all tags used on Widgets created by a user named Alan which have a size greater than 99:

```
>>> Tag.objects.usage_for_model(Widget, filters=dict(size__gt=99, user__username='Alan'))
```

The `usage_for_queryset` method allows you to pass a pre-filtered queryset to be used when determining tag usage:

```
>>> Tag.objects.usage_for_queryset(Widget.objects.filter(size__gt=99, user__username='Alan'))
```

Tag input

Tag input from users is treated as follows:

- If the input doesn't contain any commas or double quotes, it is simply treated as a space-delimited list of tag names.
- If the input does contain either of these characters, we parse the input like so:
  - Groups of characters which appear between double quotes take precedence as multi-word tags (so double quoted tag names may contain commas). An unclosed double quote will be ignored.
  - For the remaining input, if there are any unquoted commas in the input, the remainder will be treated as comma-delimited. Otherwise, it will be treated as space-delimited.

Examples:

Tag input string	Resulting tags	Notes
apple ball cat	`[apple]`, `[ball]`, `[cat]`	No commas, so space delimited
apple, ball cat	`[apple]`, `[ball cat]`	Comma present, so comma delimited
"apple, ball" cat dog	`[apple, ball]`, `[cat]`, `[dog]`	All commas are quoted, so space delimited
"apple, ball", cat dog	`[apple, ball]`, `[cat dog]`	Contains an unquoted comma, so comma delimited
apple "ball cat" dog	`[apple]`, `[ball cat]`, `[dog]`	No commas, so space delimited
"apple" "ball dog	`[apple]`, `[ball]`, `[dog]`	Unclosed double quote is ignored

CHAPTER 5

Tagged items

The relationship between a `Tag` and an object is represented by the `TaggedItem` model, which lives in the `tagging.models` module.

API reference

Fields

`TaggedItem` objects have the following fields:

- `tag` – The `Tag` an object is associated with.
- `content_type` – The `ContentType` of the associated model instance.
- `object_id` – The id of the associated object.
- `object` – The associated object itself, accessible via the Generic Relations API.

Manager functions

The `TaggedItem` model has a custom manager which has the following helper methods, which accept either a `QuerySet` or a `Model` class as one of their arguments. To restrict the objects which are returned, pass in a filtered `QuerySet` for this argument:

- `get_by_model(queryset_or_model, tag)` – creates a `QuerySet` containing instances of the specififed model which are tagged with the given tag or tags.
- `get_intersection_by_model(queryset_or_model, tags)` – creates a `QuerySet` containing instances of the specified model which are tagged with every tag in a list of tags.

  `get_by_model` will call this function behind the scenes when you pass it a list, so you can use `get_by_model` instead of calling this method directly.
- `get_union_by_model(queryset_or_model, tags)` – creates a `QuerySet` containing instances of the specified model which are tagged with any tag in a list of tags.
- `get_related(obj, queryset_or_model, num=None)` - returns a list of instances of the specified model which share tags with the model instance `obj`, ordered by the number of shared tags in descending order.

  If `num` is given, a maximum of `num` instances will be returned.

Basic usage

Retrieving tagged objects

Objects may be retrieved based on their tags using the `get_by_model` manager method:

```
>>> from shop.apps.products.models import Widget
>>> from tagging.models import Tag
>>> house_tag = Tag.objects.get(name='house')
>>> TaggedItem.objects.get_by_model(Widget, house_tag)
[<Widget: pk=1>, <Widget: pk=2>]
```

Passing a list of tags to `get_by_model` returns an intersection of objects which have those tags, i.e. tag1 AND tag2 ... AND tagN:

```
>>> thing_tag = Tag.objects.get(name='thing')
>>> TaggedItem.objects.get_by_model(Widget, [house_tag, thing_tag])
[<Widget: pk=1>]
```

Functions which take tags are flexible when it comes to tag input:

```
>>> TaggedItem.objects.get_by_model(Widget, Tag.objects.filter(name__in=['house', 'thing']))
[<Widget: pk=1>]
>>> TaggedItem.objects.get_by_model(Widget, 'house thing')
[<Widget: pk=1>]
>>> TaggedItem.objects.get_by_model(Widget, ['house', 'thing'])
[<Widget: pk=1>]
```

Restricting objects returned

Pass in a `QuerySet` to restrict the objects returned:

```
# Retrieve all Widgets which have a price less than 50, tagged with 'house'
TaggedItem.objects.get_by_model(Widget.objects.filter(price__lt=50), 'house')

# Retrieve all Widgets which have a name starting with 'a', tagged with any
# of 'house', 'garden' or 'water'.
TaggedItem.objects.get_union_by_model(Widget.objects.filter(name__startswith='a'),
                                      ['house', 'garden', 'water'])
```

CHAPTER 6

Utilities

Tag-related utility functions are defined in the `tagging.utils` module:

`parse_tag_input(input)`

Parses tag input, with multiple word input being activated and delineated by commas and double quotes. Quotes take precedence, so they may contain commas.

Returns a sorted list of unique tag names.

See *tag input* for more details.

`edit_string_for_tags(tags)`

Given list of `Tag` instances, creates a string representation of the list suitable for editing by the user, such that submitting the given string representation back without changing it will give the same list of tags.

Tag names which contain commas will be double quoted.

If any tag name which isn't being quoted contains whitespace, the resulting string of tag names will be comma-delimited, otherwise it will be space-delimited.

`get_tag_list(tags)`

Utility function for accepting tag input in a flexible manner.

If a `Tag` object is given, it will be returned in a list as its single occupant.

If given, the tag names in the following will be used to create a `Tag` `QuerySet`:

- A string, which may contain multiple tag names.
- A list or tuple of strings corresponding to tag names.
- A list or tuple of integers corresponding to tag ids.

If given, the following will be returned as-is:

- A list or tuple of `Tag` objects.
- A `Tag` `QuerySet`.

calculate_cloud(tags, steps=4, distribution=tagging.utils.LOGAR

Add a font_size attribute to each tag according to the frequency of its use, as indicated by its count attribute.

steps defines the range of font sizes - font_size will be an integer between 1 and steps (inclusive).

distribution defines the type of font size distribution algorithm which will be used - logarithmic or linear. It must be one of tagging.utils.LOGARITHMIC or tagging.utils.LINEAR.

CHAPTER 7

Model Fields

The `tagging.fields` module contains fields which make it easy to integrate tagging into your models and into the `django.contrib.admin` application.

Field types

`TagField`

A `CharField` that actually works as a relationship to tags "under the hood".

Using this example model:

```
class Link(models.Model):
    ...
    tags = TagField()
```

Setting tags:

```
>>> l = Link.objects.get(...)
>>> l.tags = 'tag1 tag2 tag3'
```

Getting tags for an instance:

```
>>> l.tags
'tag1 tag2 tag3'
```

Getting tags for a model - i.e. all tags used by all instances of the model:

```
>>> Link.tags
'tag1 tag2 tag3 tag4 tag5'
```

This field will also validate that it has been given a valid list of tag names, separated by a single comma, a single space or a comma followed by a space.

CHAPTER 8

Form fields

The `tagging.forms` module contains a `Field` for use with Django's forms library which takes care of validating tag name input when used in your forms.

Field types

`TagField`

A form `Field` which is displayed as a single-line text input, which validates that the input it receives is a valid list of tag names.

When you generate a form for one of your models automatically, using the `ModelForm` class, any `tagging.fields.TagField` fields in your model will automatically be represented by a `tagging.forms.TagField` in the generated form.

CHAPTER 9

Generic views

The `tagging.views` module contains views to handle simple cases of common display logic related to tagging.

`tagging.views.TaggedObjectList`

Description:

A view that displays a list of objects for a given model which have a given tag. This is a thin wrapper around the `django.views.generic.list.ListView` view, which takes a model and a tag as its arguments (in addition to the other optional arguments supported by `ListView`), building the appropriate `QuerySet` for you instead of expecting one to be passed in.

Required arguments:

- `tag`: The tag which objects of the given model must have in order to be listed.

Optional arguments:

Please refer to the ListView documentation for additional optional arguments which may be given.

- `related_tags`: If `True`, a `related_tags` context variable will also contain tags related to the given tag for the given model.
- `related_tag_counts`: If `True` and `related_tags` is `True`, each related tag will have a `count` attribute indicating the number of items which have it in addition to the given tag.

Template context:

Please refer to the ListView documentation for additional template context variables which may be provided.

- `tag`: The `Tag` instance for the given tag.

Example usage

The following sample URLconf demonstrates using this generic view to list items of a particular model class which have a given tag:

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

from tagging.views import TaggedObjectList

from shop.apps.products.models import Widget
```

```
urlpatterns = patterns('',
    url(r'^widgets/tag/(?P<tag>[^/]+(?u))/$',
        TaggedObjectList.as_view(model=Widget, paginate_by=10, allow_empty=True),
        name='widget_tag_detail'),
)
```

The following sample view demonstrates wrapping this generic view to perform filtering of the objects which are listed:

```
from myapp.models import People

from tagging.views import TaggedObjectList

class TaggedPeopleFilteredList(TaggedObjectList):
    queryset = People.objects.filter(country__code=country_code)
```

CHAPTER 10

Template tags

The `tagging.templatetags.tagging_tags` module defines a number of template tags which may be used to work with tags.

Tag reference

tags_for_model

Retrieves a list of `Tag` objects associated with a given model and stores them in a context variable.

Usage:

```
{% tags_for_model [model] as [varname] %}
```

The model is specified in `[appname].[modelname]` format.

Extended usage:

```
{% tags_for_model [model] as [varname] with counts %}
```

If specified - by providing extra `with counts` arguments - adds a `count` attribute to each tag containing the number of instances of the given model which have been tagged with it.

Examples:

```
{% tags_for_model products.Widget as widget_tags %}
{% tags_for_model products.Widget as widget_tags with counts %}
```

tag_cloud_for_model

Retrieves a list of `Tag` objects for a given model, with tag cloud attributes set, and stores them in a context variable.

Usage:

```
{% tag_cloud_for_model [model] as [varname] %}
```

The model is specified in `[appname].[modelname]` format.

Extended usage:

```
{% tag_cloud_for_model [model] as [varname] with [options] %}
```

Extra options can be provided after an optional `with` argument, with each option being specified in `[name]=[value]` format. Valid extra options are:

`steps` Integer. Defines the range of font sizes.

`min_count` Integer. Defines the minimum number of times a tag must have been used to appear in the cloud.

`distribution` One of `linear` or `log`. Defines the font-size distribution algorithm to use when generating the tag cloud.

Examples:

```
{% tag_cloud_for_model products.Widget as widget_tags %}
{% tag_cloud_for_model products.Widget as widget_tags with steps=9 min_count=3 distribution=log %}
```

tags_for_object

Retrieves a list of `Tag` objects associated with an object and stores them in a context variable.

Usage:

```
{% tags_for_object [object] as [varname] %}
```

Example:

```
{% tags_for_object foo_object as tag_list %}
```

tagged_objects

Retrieves a list of instances of a given model which are tagged with a given `Tag` and stores them in a context variable.

Usage:

```
{% tagged_objects [tag] in [model] as [varname] %}
```

The model is specified in `[appname].[modelname]` format.

The tag must be an instance of a `Tag`, not the name of a tag.

Example:

```
{% tagged_objects comedy_tag in tv.Show as comedies %}
```