django-taggit Documentation

Release 0.9.3

Alex Gaynor

May 31, 2013

CONTENTS

`django-taggit` is a reusable Django application designed to making adding tagging to your project easy and fun.

`django-taggit` works with Django 1.1 and 1.2 (see *Known Issues* for known issues with older versions of Django), and Python 2.4-2.X.

CHAPTER

ONE

GETTING STARTED

To get started using `django-taggit` simply install it with `pip`:

```
$ pip install django-taggit
```

Add `"taggit"` to your project's `INSTALLED_APPS` setting.

And then to any model you want tagging on do the following:

```
from django.db import models

from taggit.managers import TaggableManager

class Food(models.Model):
    # ... fields here

    tags = TaggableManager()
```

CHAPTER

TWO

TAGS IN FORMS

The `TaggableManager` will show up automatically as a field in a `ModelForm` or in the admin. Tags input via the form field are parsed 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:
  - 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.
  - Otherwise, if there are any unquoted commas in the input, it will be treated as comma-delimited. If not, 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

2.1 `commit=False`

If, when saving a form, you use the `commit=False` option you'll need to call `save_m2m()` on the form after you save the object, just as you would for a form with normal many to many fields on it:

```
if request.method == "POST":
    form = MyFormClass(request.POST)
    if form.is_valid():
        obj = form.save(commit=False)
        obj.user = request.user
        obj.save()
        # Without this next line the tags won't be saved.
        form.save_m2m()
```

CHAPTER

THREE

USING TAGS IN THE ADMIN

By default if you have a `TaggableManager` on your model it will show up in the admin, just as it will in any other form. One important thing to note is that you *cannot* include a `TaggableManager` in `ModelAdmin.list_display`, if you do you'll see an exception that looks like:

```
AttributeError: 'TaggableManager' object has no attribute 'flatchoices'
```

This is for the same reason that you cannot include a `ManyToManyField`, it would result in an unreasonable number of queries being executed. If you really would like to add it, you can read the Django documentation.

CHAPTER

FOUR

THE API

After you've got your `TaggableManager` added to your model you can start playing around with the API.

class **`TaggableManager`**(*[verbose_name="Tags", help_text="A comma-separated list of tags.", through=None, blank=False]*)

Parameters

- **verbose_name** – The verbose_name for this field.
- **help_text** – The help_text to be used in forms (including the admin).
- **through** – The through model, see *Using a Custom Tag or Through Model* for more information.
- **blank** – Controls whether this field is required.

`add`(**tags*)

This adds tags to an object. The tags can be either `Tag` instances, or strings:

```
>>> apple.tags.all()
[]
>>> apple.tags.add("red", "green", "fruit")
```

`remove`(**tags*)

Removes a tag from an object. No exception is raised if the object doesn't have that tag.

`clear`()

Removes all tags from an object.

`set`(**tags*)

Adds and removes tags from an object to match the tags specified.

`similar_objects`()

Returns a list (not a lazy `QuerySet`) of other objects tagged similarly to this one, ordered with most similar first. Each object in the list is decorated with a `similar_tags` attribute, the number of tags it shares with this object.

If the model is using generic tagging (the default), this method searches tagged objects from all classes. If you are querying on a model with its own tagging through table, only other instances of the same model will be returned.

4.1 Filtering

To find all of a model with a specific tags you can filter, using the normal Django ORM API. For example if you had a `Food` model, whose `TaggableManager` was named `tags`, you could find all the delicious fruit like so:

```
>>> Food.objects.filter(tags__name__in=["delicious"])
[<Food: apple>, <Food: pear>, <Food: plum>]
```

If you're filtering on multiple tags, it's very common to get duplicate results, because of the way relational databases work. Often you'll want to make use of the `distinct()` method on `QuerySets`:

```
>>> Food.objects.filter(tags__name__in=["delicious", "red"])
[<Food: apple>, <Food: apple>]
>>> Food.objects.filter(tags__name__in=["delicious", "red"]).distinct()
[<Food: apple>]
```

You can also filter by the slug on tags. If you're using a custom `Tag` model you can use this API to filter on any fields it has.

4.2 Aggregation

Unfortunately, due to a bug in Django, it is not currently possible to use aggregation in conjunction with `taggit`. This is a documented interaction of generic relations (which `taggit` uses internally) and aggregates.

CHAPTER

FIVE

USING A CUSTOM TAG OR THROUGH MODEL

By default django-taggit uses a "through model" with a GenericForeignKey on it, that has another ForeignKey to an included Tag model. However, there are some cases where this isn't desirable, for example if you want the speed and referential guarantees of a real ForeignKey, if you have a model with a non-integer primary key, or if you want to store additional data about a tag, such as whether it is official. In these cases django-taggit makes it easy to substitute your own through model, or Tag model.

Your intermediary model must be a subclass of taggit.models.TaggedItemBase with a foreign key to your content model named content_object. Pass this intermediary model as the through argument to TaggableManager:

```
from django.db import models

from taggit.managers import TaggableManager
from taggit.models import TaggedItemBase


class TaggedFood(TaggedItemBase):
    content_object = models.ForeignKey('Food')

class Food(models.Model):
    # ... fields here

    tags = TaggableManager(through=TaggedFood)
```

Once this is done, the API works the same as for GFK-tagged models.

To change the behavior in other ways there are a number of other classes you can subclass to obtain different behavior:

Class name	Behavior
TaggedItemBase	Allows custom ForeignKeys to models.
GenericTaggedItemBase	Allows custom Tag models.
ItemBase	Allows custom Tag models and ForeignKeys to models.

When providing a custom Tag model it should be a ForeignKey to your tag model named "tag".

class TagBase

slugify(*tag*, *i=None*)

By default taggit uses django.template.defaultfilters.slugify() to calculate a slug for a given tag. However, if you want to implement your own logic you can override this method, which receives the tag (a string), and i, which is either None or an integer, which signifies how many times the slug for this tag has been attempted to be calculated, it is `None` on the first time, and the counting begins at `1` thereafter.

CHAPTER

SIX

KNOWN ISSUES

Currently there is 1 known issue:

- When run under Django 1.1, doing `Model.objects.all().delete()` (or any bulk deletion operation) on a model with a `TaggableManager` will result in losing the tags for items beyond just those assosciated with the deleted objects. This issue is not present in Django 1.2.

CHAPTER

SEVEN

EXTERNAL APPLICATIONS

In addition to the features included in `django-taggit` directly, there are a number of external applications which provide additional features that may be of interest.

Note: Despite their mention here, the following applications are in no way official, nor have they in any way been reviewed or tested.

If you have an application that you'd like to see listed here, simply fork `taggit` on github, add it to this list, and send a pull request.

- `django-taggit-suggest`: Provides support for defining keyword and regular expression rules for suggesting new tags for content. This used to be available at `taggit.contrib.suggest`. Available on github.
- `django-taggit-templatetags`: Provides several templatetags, including one for tag clouds, to expose various `taggit` APIs directly to templates. Available on github.

CHAPTER

EIGHT

CHANGELOG

8.1 0.9.4

Unreleased.

- *Backwards incompatible* The `name` field of the `Tag` model is now marked as unique. You should update your database schema using `ALTER TABLE taggit_tag ADD UNIQUE (name).`

8.2 0.9.2

Unreleased.

- *Backwards incompatible* Forms containing a `TaggableManager` by default now require tags, to change this provide `blank=True` to the `TaggableManager`.

8.3 0.9.0

- Added a Hebrew locale.
- Added an index on the `object_id` field of `TaggedItem`.
- When displaying tags always join them with commas, never spaces.
- The docs are now available online.
- Custom `Tag` models are now allowed.
- *Backwards incompatible* Filtering on tags is no longer `filter(tags__in=["foo"])`, it is written `filter(tags__name__in=["foo"])`.
- Added a German locale.
- Added a Dutch locale.
- Removed `taggit.contrib.suggest`, it now lives in an external application, see *External Applications* for more information.

8.4 0.8.0

- Fixed querying for objects using `exclude(tags__in=tags)`.

- Marked strings as translatable.
  - Added a Russian translation.
- Created a mailing list.
- Smarter tagstring parsing for form field; ported from Jonathan Buchanan's django-tagging. Now supports tags containing commas. See *Tags in forms* for details.
- Switched to using savepoints around the slug generation for tags. This ensures that it works fine on databases (such as Postgres) which dirty a transaction with an `IntegrityError`.
- Added Python 2.4 compatibility.
- Added Django 1.1 compatibility.

CHAPTER

NINE

INDICES AND TABLES

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