django-tagulous Documentation

Release 1.3.3

Richard Terry

Apr 10, 2022

CONTENTS

Tagulous is a fully-featured tagging library for Django built on `ForeignKey` and `ManyToManyField`, giving you all their normal power with a sprinkling of tagging syntactic sugar, and a full set of extra *features*.

See also:

Read this online at http://radiac.net/projects/django-tagulous/

CHAPTER ONE

CONTENTS

1.1 Introduction

1.1.1 Features

- Easy to install - simple requirements, simple syntax, lots of options
- Based on ForeignKey and ManyToManyField, so it's easy to query
- Autocomplete support built in, if you want it
- Supports multiple independent tag fields on a single model
- Can be used as a CharField with dynamic choices
- Supports trees of nested tags, for detailed categorisation
- Admin support for managing tags and tagged models

1.1.2 Quickstart

Install with `pip install django-tagulous`, add `tagulous` to Django's `INSTALLED_APPS` and *define the serializers*, then start adding tag fields to your model:

```
from django.db import models
from tagulous.models import SingleTagField, TagField

class Person(models.Model):
    name = models.CharField(max_length=255)
    title = SingleTagField(initial="Mr, Mrs, Miss, Ms")
    skills = TagField()
```

A `SingleTagField` is based on a `ForeignKey`, and a `TagField` is based on a `ManyToManyField`.

They have relationships to a `TagModel`, which is automatically created for you if you don't specify one.

Assign strings to the fields to create new tags:

```
myperson = Person.objects.create(name='Bob', title='Mr', skills='run, hop')
# myperson.skills == 'run, hop'
myperson.skills = ['jump', 'kung fu']
myperson.save()
# myperson.skills == 'jump, "kung fu"'
runners = Person.objects.filter(skills='run')
```

Use them like a normal Django relationship in your queries:

```
qs = MyRelatedModel.objects.filter(
    person__skills__name__in=['run', 'jump'],
)
```

As well as this you also get:

- tag field support in public *forms* and the *admin* site, including autocompletion
- easy to build *tag clouds*
- ability to nest tags in *trees* for more complex categorisation

Take a look at the *Example Usage* to see what else you can do, or read on through the *documentation* to see exactly how everything works.

1.1.3 Glossary

This documentation uses a few terms to explain the ways tags are stored and set:

Tagged model A model which has been tagged using *Model Fields*.

Tag model A model where the tag definition is stored. It must be a subclass of *tagulous.models.TagModel*, but will be auto-generated by a tag field if it is not explicitly set.

Tag An instance of a tag model

Tag name The unique name of a tag, eg `"run"`. This is the value stored on the `name` attribute of a tag model.

Tag string A tag string is a list of tag names stored in a single string, in tag format, eg `"run, jump, hop"`. The format of this string is defined by the *Tag String Parser*.

1.1.4 Comparison with other tagging libraries

Popular tagging libraries for Django include: * django-taggit * django-tagging * django-tagging-ng

If you are already using one of these, read *Converting to Tagulous* to see what is involved in switching to Tagulous.

Tagulous is easier to use and has more features, and is a proven library which has been in use since Django 1.4.

Real relations

The Tagulous `TagField` is based on `ManyToManyField`, so you can set and query tag objects like a normal M2M field, but also use tag strings and lists of tag names.

django-tagging and django-taggit both use generic relations, which tend to be second-class citizens in Django - they are often slower and lack functionality compared to native FK and M2M fields. This means they have a more convoluted syntax and queries are more complex and limited.

Separate tag models

In Tagulous, tag models can be independent or shared - this allows you to have multiple tag fields on one model which each have their own sets of tags, or share sets of tags between fields and models as you wish - see the *Tag Models* documentation for more details.

You can also easily define custom tag models in Tagulous, to store additional data on with tags - see the *Custom Tag Models* documentation and *this example* for more details.

django-taggit can be configured to use custom models so it can have separate sets of tags, but requires a bit more work. django-tagging does not support separate sets of tags or custom models.

More customisable

Tagulous is designed to be configurable. For example, it lets you protect tags from being removed when they're no longer in use, they can be case sensitive, forced to lowercase, you can specify a maximum number of tags for a field, and whether or not space should be used as a delimiter. See the *Tag Options* documentation for more details.

django-tagging only lets you force tags to lowercase, and django-taggit only lets you toggle case sensitivity.

Built-in autocomplete

Tagulous has built-in support for autocomplete; tags can either be embedded into the page, or queried using the ajax views provided. It uses Select2, but it has been designed to be easy to switch that out for something else using *autocomplete adaptors*.

The JavaScript and Python code is closely integrated - the same tag parser has been implemented in both to ensure tag strings are treated consistently.

Neither django-tagging and django-taggit support autocomplete out of the box; you need to add another library to do that.

Better admin support

Tagulous tag fields are first-class citizens in Django's admin site. You can show them in `list_display`, use them to filter your model, and can register tag models to rename and merge tags. Tag fields and autocomplete work throughout admin forms and inlines. See the *Admin* documentation for more details.

django-tagging and django-taggit tags cannot be shown in `list_display`, and there are no special admin tools.

Single tag mode

The standard `TagField` is based on a `ManyToManyField` for conventional tagging, but Tagulous also provides a `SingleTagField`, which is based on `ForeignKey`. This acts more like a `CharField` with dynamic `choices` that users can add to at runtime. See the *Model Fields* documentation for more details.

django-tagging and django-taggit don't have an equivalent feature.

Hierarchical tag trees

Tagulous has a tree mode, which lets you create sub-tags using the `/` character in a tag name. You can query and navigate a tag tree as you would expect (querying for parents, siblings, children, descendants etc), as well as rename and merge subtrees from your code or the Django admin. See the *Tag Trees* documentation for more details.

django-tagging and django-taggit don't have an equivalent feature.

And there's more

Tagulous is packed with small features which make it easy to work with, such as:

- a more robust *tag string parser* with better support for quoted tags.
- automatic *slug* generation, and *path* generation for tree tags.
- tag model managers and querysets have a *weight* method to make it easy to build custom tag clouds.

1.2 Installation

1.2.1 Instructions

1. Install `django-tagulous`:

   ```
   pip install django-tagulous
   ```

2. In your site settings, add Tagulous to `INSTALLED_APPS` and tell Django to use the Tagulous serialization modules:

   ```
   INSTALLED_APPS = (
       ...
       'tagulous',
   )

   SERIALIZATION_MODULES = {
       'xml':    'tagulous.serializers.xml_serializer',
       'json':   'tagulous.serializers.json',
       'python': 'tagulous.serializers.python',
       'yaml':   'tagulous.serializers.pyyaml',
   }
   ```

   There are other global *Settings* you can add here.

3. Add Tagulous fields to your project - see *Models*, *Forms* and *Example Usage*.

Remember to run `manage.py collectstatic` to collect the JavaScript and CSS resources.

When you want to upgrade your Tagulous installation in the future, check *Upgrading* to see if there are any special actions that you need to take.

Note: If you use MySQL there are some limitations you should be aware of - see:

- the *setting* for max length for limitations of maximum tag lengths
- the tag option *case_sensitive* for limitations of case sensitivity.

1.2.2 Settings

Note: Model and form field options are managed separately by *Tag Options*.

`TAGULOUS_NAME_MAX_LENGTH` `TAGULOUS_SLUG_MAX_LENGTH` `TAGULOUS_LABEL_MAX_LENGTH`

Default max lengths for tag models.

Note: When MySQL is using utf8mb4 charset, all unique fields have a max-length of 191 characters, because MySQL max key length in 767 bytes and utf8mb4 reserves 4 bytes per character, thus 767/4 = 191.

If you use MySQL, we therefore recommend the following settings:

TAGULOUS_NAME_MAX_LENGTH=191

Default:

```
TAGULOUS_NAME_MAX_LENGTH = 255
TAGULOUS_SLUG_MAX_LENGTH = 50
TAGULOUS_LABEL_MAX_LENGTH = TAGULOUS_NAME_MAX_LENGTH
```

`TAGULOUS_SLUG_TRUNCATE_UNIQUE` Number of characters to allow for the numerical suffix when finding a unique slug, ie if set to 5, the slug will be truncated by up to 5 characters to allow for a suffix of up to *_9999*.

Default: `5`

`TAGULOUS_SLUG_ALLOW_UNICODE` If `True` unicode will be allowed in slugs. If `False` tag slugs will be forced to ASCII.

As with Django's `slugify`, this is off by default.

Default: `False`

`TAGULOUS_AUTOCOMPLETE_JS` `TAGULOUS_ADMIN_AUTOCOMPLETE_JS`

List of static JavaScript files required for Tagulous autocomplete. These will be added to the form media when a Tagulous form field is used.

The order is important: the adaptor must appear last in the list, so that it is loaded after its dependencies.

If you use jQuery elsewhere on your site, you may need to remove *jquery.js* to avoid conflicts.

Default:

```
TAGULOUS_AUTOCOMPLETE_JS = (
    "tagulous/lib/jquery.js",
    "tagulous/lib/select2-4/js/select2.full.min.js",
    "tagulous/tagulous.js",
    "tagulous/adaptor/select2-4.js",
)
```

`TAGULOUS_AUTOCOMPLETE_CSS` `TAGULOUS_ADMIN_AUTOCOMPLETE_CSS`

List of static CSS files required for Tagulous autocomplete. These will be added to the form media when a Tagulous form field is used.

The default list will use the included version of Select2.

Default:

```
TAGULOUS_AUTOCOMPLETE_CSS = {
    'all': ['tagulous/lib/select2-4/css/select2.min.css']
}
```

TAGULOUS_AUTOCOMPLETE_SETTINGS Any settings to pass to the JavaScript via the adaptor. They can be overridden by a field's *autocomplete_settings* option.

For example, the select2 control defaults to use the same width as the form element it replaces; you can override this by passing their `width` option (see their docs on appearance) as an autocomplete setting:

```
TAGULOUS_AUTOCOMPLETE_SETTINGS = {"width": "75%"}
```

If set to `None`, no settings will be passed.

Default: `None`

TAGULOUS_WEIGHT_MIN The default minimum value for the *weight* queryset method.

Default: `1`

TAGULOUS_WEIGHT_MAX The default maximum value for the *weight* queryset method.

Default: `6`

TAGULOUS_ENHANCE_MODELS **Advanced usage** - only use this setting if you know what you're doing.

Tagulous automatically enhances models, managers and querysets to fully support tag fields. This has the theoretical potential for unexpected results, so this setting lets the cautious disable this enhancement.

If you set this to False you will need to manually add Tagulous mixins to your models, managers and querysets.

See *Tagged Models* for more information.

Default: `True`

1.2.3 System checks

Tagulous adds to the Django system check framework with the following:

tagulous.W001 `settings.SERIALIZATION_MODULES` has not been configured as expected

A common installation error is to forget to set `SERIALIZATION_MODULES` as described in the *installation instructions*.

This is a straight string comparison. If your serialisation modules don't match what Tagulous is expecting (you're subclassing the Tagulous modules, for example), you can disable this warning with the setting:

```
SILENCED_SYSTEM_CHECKS = ["tagulous.W001"]
```

1.2.4 Converting to Tagulous

If you're already using a tagging library which you'd like to replace with Tagulous, freeze the tags into a temporary column, remove the old tagging code, add a new tagulous TagField, then copy the tags back across.

Warning: This hasn't been tested with your data, so back up your database first, just in case.

1. Create a schema migration to add a `TextField` to your tagged model, where we'll temporarily store the tags for that instance.

   `django-taggit` example:

   ```
   class MyModel(models.Model):
       ...
       tags = TaggableManager()
       tags_store = models.TextField(blank=True)
   ```

   `django-tagging` example:

   ```
   class MyModel(models.Model):
       ...
       tags_store = models.TextField(blank=True)
   tagging.register(MyModel)
   ```

2. Create a data migration to copy the tags into the new field as a string.

   `django-taggit` example:

   ```
   def store_tags(apps, schema_editor):
       import tagulous
       model = apps.get_model('myapp', 'MyModel')
       for obj in model.objects.all():
           obj.tags_store = tagulous.utils.render_tags(obj.tags.all())

   class Migration(migrations.Migration):
       operations = [
           migrations.RunPython(store_tags)
       ]
   ```

   The example for `django-tagging` would be the same, only replace `obj.tags.all()` with `obj.tags`.

3. Remove the old tagging code from your model, and create a schema migration to clean up any unused fields or models.

4. Add a `TagField` to your tagged model and create a schema migration:

   ```
   import tagulous
   class MyModel(models.Model):
       tags = tagulous.models.TagField()
       tags_store = models.TextField(blank=True)
   ```

   Be careful to set appropriate arguments, ie `blank=True` if some of your `tags_store` fields may be empty.

5. Create a data migration to copy the tags into the new field.

   Example:

```python
def load_tags(apps, schema_editor):
    model = apps.get_model('myapp', 'MyModel')
    for obj in model.objects.all():
        obj.tags = obj.tags_store
        obj.tags.save()

class Migration(migrations.Migration):
    operations = [
        migrations.RunPython(load_tags)
    ]
```

6. Create a schema migration to remove the temporary tag storage field (`tag_store` in these examples)
7. Apply the migrations and start using tagulous

1.3 Example Usage

This section contains code examples of how to set up and use Tagulous. If you'd like a more interactive demonstration, there is a static demo showing the front-end, or an example project for you to install locally and play with some of these code examples.

1.3.1 Automatic tag models

This simple example creates a `SingleTagField` (a glorified `ForeignKey`) and two `TagField` (a typical tag field, using `ManyToManyField`):

```python
from django.db import models
import tagulous.models

class Person(models.Model):
    title = tagulous.models.SingleTagField(
        label="Your preferred title",
        initial="Mr, Mrs, Ms",
    )
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField(
        force_lowercase=True,
        max_count=5,
    )
```

- This will create two new models at runtime to store the tags, `Tagulous_Person_title` and `Tagulous_Person_skills`.
- These models will act like normal models, and can be managed in the database using Django migrations
- `Person.title` will now act as a `ForeignKey` to `Tagulous_Person_title`
- `Person.skills` will now act as a `ManyToManyField` to `Tagulous_Person_skills`

Initial tags need to be loaded into the database with the *Loading initial tags* management command.

You can use the fields to assign and query values:

```
# Person.skills.tag_model == Tagulous_Person_skills

# Set tags on an instance with a string
instance = Person()
instance.skills = 'run, "kung fu", jump'

# They're not committed to the database until you save
instance.save()

# Get a list of all tags
tags = Person.skills.tag_model.objects.all()

# Assign a list of tags
instance.skills = ['jump', 'kung fu']
# Tags are readable before saving
# str(instance.skills) == 'jump, "kung fu"'
instance.save()

# Step through the list of instances in the tag model
for skill in instance.skills.all():
    do_something(skill)

# Compare tag fields
if instance.skills == other.skills:
    return True
```

1.3.2 Custom models

You can create a tag model manually, and specify it in one or more tag fields:

```
import tagulous.models

class Hobbies(tagulous.models.TagModel):
    class TagMeta:
        # Tag options
        initial = "eating, coding, gaming"
        force_lowercase = True
        autocomplete_view = 'myapp.views.hobbies_autocomplete'

class Person(models.Model):
    name = models.CharField(max_length=255)
    hobbies = tagulous.models.TagField(to=Hobbies)
```

Options for a custom tag model must be set in *TagMeta* - you cannot pass them as arguments in tag fields.

See *Tag Models* to see which field names Tagulous uses internally.

1.3.3 Tag Trees

A tag field can specify `tree=True` to use slashes in tag names to denote children:

```
import tagulous.models
class Person(models.Model):
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField(
        force_lowercase=True,
        max_count=5,
        tree=True,
    )
```

This can't be set in the tag model's `TagMeta` object; the tag model must instead subclass *tagulous.models.TagTreeModel*:

```
class Hobbies(tagulous.models.TagTreeModel):
    class TagMeta:
        initial = "food/eating, food/cooking, gaming/football"
        force_lowercase = True
        autocomplete_view = 'myapp.views.hobbies_autocomplete'

class Person(models.Model):
    name = models.CharField(max_length=255)
    hobbies = tagulous.models.TagField(to=Hobbies)
```

You can add tags as normal, and then query using tree relationships:

```
person.hobbies = "food/eating/mexican, sport/football"
person.save()

# Get all root nodes: "food", "gaming" and "sport"
root_nodes = Hobbies.objects.filter(parent=None)

# Get the direct children of food: "food/eating", "food/cooking"
food_children = Hobbies.objects.get(name="food").children.all()

# Get all descendants of food:
#   "food/eating", "food/eating/mexican", "food/cooking"
food_children = Hobbies.objects.get(name="food").get_descendants()
```

See *Tag Trees* to see a full list of available tree methods and properties.

1.3.4 Tag URL

You can set the `get_absolute_url` tag option to a callable to give tag objects absolute URLs without needing to create a custom tag model:

```
from django.db import models
from django.core.urlresolvers import reverse
import tagulous.models

class Person(models.Model):
```

(continues on next page)

(continued from previous page)

```
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField(
        get_absolute_url=lambda tag: reverse(
            'myapp.views.by_skill', kwargs={'skill_slug': tag.slug}
        ),
    )
```

The `get_absolute_url` method can now be called as normal; for example, from a template:

```
{% for skill in person.skills.all %}
    <a href="{{ skill.get_absolute_url }}">{{ skill.name }}</a>
{% endfor %}
```

If you are using a tree, you will want to use the path instead:

```
skills = tagulous.models.TagField(
    tree=True,
    get_absolute_url=lambda tag: reverse(
        'myapp.views.by_skill', kwargs={'skill_path': tag.path}
    ),
)
```

See the *get_absolute_url* option for more details.

1.3.5 ModelForms

A `ModelForm` with tag fields needs no special treatment:

```
from django.db import models
from django import forms
import tagulous.models

class Person(models.Model):
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField()

class PersonForm(forms.ModelForm):
    class Meta:
        fields = ['name', 'skills']
        model = Person
```

They are normal forms so can be used in normal ways; for example, with class-based views:

```
from django.views.generic.edit import CreateView

class PersonCreate(CreateView):
    model = Person
    fields = ['name', 'skills']
```

or with view functions:

```python
def person_create(request, template_name="my_app/person_form.html"):
    form = PersonForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('home')
    return render(request, template_name, {'form': form})
```

However, because a `TagField` is based on a `ManyToManyField`, if you save your form using `commit=False`, you will need to call `save_m2m` to save the tags:

```python
class Pet(models.Model):
    owner = models.ForeignKey('auth.User')
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField()

class PetForm(forms.ModelForm):
    class Meta:
        fields = ['owner', 'name', 'skills']
        model = Pet

def pet_create(request, template_name="my_app/pet_form.html"):
    form = PetForm(request.POST or None)
    if form.is_valid():
        pet = form.save(commit=False)
        pet.owner = request.user

        # Next line will save all non M2M fields (including SingleTagField)
        pet.save()

        # Next line will save any ``TagField`` values
        form.save_m2m()

        return redirect('home')
    return render(request, template_name, {'form': form})
```

As shown above, this only applies to `TagField` - a `SingleTagField` is based on `ForeignKey`, so will be saved without needing `save_m2m`.

See *Forms* for how to use tag fields in forms.

1.3.6 Forms without models

Tagulous form fields take tag options as a single `TagOptions` object, rather than as separate arguments as a model form does:

```python
from django import forms
import tagulous.forms

class PersonForm(forms.ModelForm):
    title = tagulous.forms.SingleTagField(
        autocomplete_tags=['Mr', 'Mrs', 'Ms']
    )
    name = forms.CharField(max_length=255)
```

(continues on next page)

(continued from previous page)

```
    skills = tagulous.forms.TagField(
        tag_options=tagulous.models.TagOptions(
            force_lowercase=True,
        ),
        autocomplete_tags=['running', 'jumping', 'judo']
    )
```

A SingleTagField will return a string, and a TagField will return a list of strings:

```
form = PersonForm(data={
    'title':    'Mx',
    'skills':   'Running, judo',
})
assert form.is_valid()
assert form.cleaned_data['title'] == 'Mx'
assert form.cleaned_data['skills'] == ['running', 'judo']
```

See *Forms* for how to use tag fields in forms.

1.3.7 Filtering embedded autocomplete

Filtering autocomplete to initial tags only

If it often useful for autocomplete to only list your initial tags, and not those added by others; Tagulous makes this easy with the autocomplete_initial field option:

```
class Person(models.Model):
    title = tagulous.models.SingleTagField(
        label="Your preferred title",
        initial="Mr, Mrs, Ms",
        autocomplete_initial=True,
    )
```

Even if users add new tags, only the initial tags will ever be shown as autocomplete options.

See *autocomplete_initial* for more details.

Filtering autocomplete by related fields

This example will embed the tags into the HTML of the response; if you are using autocomplete views, see *Filtering an autocomplete view* instead.

Filter the autocomplete_tags queryset after the form initialises:

```
from django.db import models
from django import forms
import tagulous

class Pet(models.Model):
    owner = models.ForeignKey('auth.User')
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField()
```

(continues on next page)

(continued from previous page)

```
class PetForm(forms.ModelForm):
    def __init__(self, user, *args, **kwargs):
        super(PetForm, self).__init__(*args, **kwargs)

        # Filter skills to initial skills, or ones added by this user
        self.fields['skills'].autocomplete_tags = \
            self.fields['skills'].autocomplete_tags.filter_or_initial(
                pet__owner=user
            ).distinct()
    class Meta:
        model = Pet
```

Then call `PetForm` with the user as the first argument, for example:

```
def add_pet(request):
    form = PetForm(request.user)
    # ...
```

For more details, see filter_by_related and *Filtering autocomplete tags*.

1.3.8 Autocomplete AJAX Views

To use AJAX to populate your autocomplete using JavaScript, set the tag option `autocomplete_view` in your models to a value for `reverse()`:

```
class Person(models.Model):
    name = models.CharField(max_length=255)
    skills = tagulous.models.TagField(
        autocomplete_view='person_skills_autocomplete'
    )
```

You can then use the default autocomplete views directly in your urls:

```
import tagulous
from myapp.models import Person
urlpatterns = [
    url(
        r'^person/skills/autocomplete/',
        tagulous.views.autocomplete,
        {'tag_model': Person},
        name='person_skills_autocomplete',
    ),
]
```

See *Views and Templates* for more details.

Filtering an autocomplete view

Add a wrapper function which filters the queryset before it calls the normal `autocomplete` view:

```
@login_required
def autocomplete_pet_skills(request):
    return tagulous.views.autocomplete(
        request,
        Pet.skills.tag_model.objects.filter_or_initial(
            pet__owner=user
        ).distinct()
    )
```

1.3.9 Django REST Framework

The Django REST framework's `ModelSerializer` will serialize tag fields to their primary keys; for example:

```
class PersonKeySerializer(ModelSerializer):
    class Meta:
        model = Person
        fields = ["name", "title", "skills"]

person = Person.objects.create(name="adam", title="mr", skills="run, jump")
PersonKeySerializer(Person).data == {
    "name": "adam",
    "title": 1,
    "skills": [1, 2]
```

If you'd prefer to serialize to strings, use the Tagulous `TagSerializer`:

```
from tagulous.contrib.drf import TagSerializer

class PersonStringSerializer(TagSerializer):
    class Meta:
        model = Person
        fields = ["name", "title", "skills"]

person = Person.objects.create(name="adam", title="mr", skills="run, jump")
PersonStringSerializer(Person).data == {
    "name": "adam",
    "title": "mr",
    "skills": ["run", "jump"]
```

1.4 Tag String Parser

Tagulous model and form fields accept a tag string value - a list of tag names separated by spaces or commas:

```
# These will parse to 'run', 'jump', 'hop'
'run jump hop'
'run,jump,hop'
```

If the tag string contains both spaces and commas, commas take priority. Spaces at the start or end of a tag name are ignored by the parser:

```
# These will parse to 'run', 'shot put', 'hop'
# This is also how Tagulous will render these tags
'run, shot put, hop'
```

If a tag name contains a space or a comma it should be escaped by quote marks for clarity, and will be when Tagulous renders the tag string:

```
# This is how Tagulous will render 'run', 'shot put'
'run, "shot put"'
```

Again, quoted tag names can be separated by spaces or commas, and commas take priority:

```
# These will parse to 'run', 'shot put', 'hop'
'run "shot put" hop'
'run,"shot put",hop'

# But this will parse to 'run "shot put"' 'hop'
'run "shot put", hop'
```

If the tag model is a *tree*, the tag name is the full path, which is split on the `/` character into a path of tag nodes; the tag label is the final part of the path. The parser ignores a single slash if it is escaped with another, ie `slash//escaped`.

If the tag field has *space_delimiter* set to `False` then only commas will be used to separate tags.

The parser is implemented in both Python and JavaScript for consistency.

For more examples and how the parser treats odd edge cases, see the examples used for testing the parser in tests/test_utils.py and tests/spec/javascripts/tagulous.spec.js.

1.4.1 Using the parser directly

Normally Tagulous uses the parser automatically behind the scenes when needed; however, there may be times when you need to parse or render tag strings manually - for example, when *Converting to Tagulous* or *Writing a custom autocomplete adaptor*.

In Python

The python parser can be found in `tagulous.utils`:

`tag_names = tagulous.utils.parse_tags(tag_string, max_count=0, space_delimiter=True)`
Given a tag string, returns a sorted list of unique tag names.

The parser does not attempt to enforce *force_lowercase* or *case_sensitive* options - these should be applied before and after parsing, respectively.

The optional `max_count` argument defaults to `0`, which means no limit. For any other value, if more tags are returned than specified, the parser will raise a `ValueError`.

The optional `space_delimiter` argument defaults to `True`, to allow either spaces or commas to be used as deliminaters to separate the tags, with priority for commas. If `False`, only commas will be used as the delimiter.

`tag_string = tagulous.utils.render_tags(tag_names)` Given a list of tags or tag names, generate a tag string.

`node_labels = tagulous.utils.split_tree_name(tag_name)` Given a tree tag name, split it on valid / characters into a list of labels for each node in the tag's path.

`tag_name = tagulous.utils.join_tree_name(parts)` Given a list of node labels, return a tree tag name.

In JavaScript

The JavaScript parser will normally be automatically added to the page by tag fields, as one of the scripts in `TAGULOUS_AUTOCOMPLETE_JS` (see *Settings*). However, if for some reason you want to use it without a tag field, you can add it to your page manually with:

```
<script src="{% static "tagulous/tagulous.js %}"></script>
```

The parser adds the global variable `Tagulous`:

`tagNames = Tagulous.parseTags(tagString, spaceDelimiter=true, withRaw=false)` Given a tag string, returns a sorted list of unique tag names

If `spaceDelimiter=false`, only commas will be used to separate tag names. If it is unset or true, spaces are used as well as commas.

The option `withRaw=true` is intended for use when parsing live input; the function will instead return `[tags, raws]`, where `tags` is a list of tags which is unsorted and not unique, and `raws` is a list of raw strings which were left after the corresponding entry in `tags` was parsed. For example:

```
var result = Tagulous.parseTags('one,two,three', true, true),
    tags = result[0],
    raws = parsed[1];
tags === ['one', 'two', 'three'];
raws === ['two,three', 'three', null];
```

If the last tag is not explicitly ended with a delimiter, the corresponding item in `raws` will be `null` instead of an empty string, to indicate that the parser unexpectedly ran out of characters.

This is useful when parsing live input if the last item in `raws` is an empty string the tag has bee closed; if it is `null` then the tag is still being entered.

`tagString = Tagulous.renderTags(tagNames)` Given a list of tag names, generate a tag string.

1.5 Models

Tagulous provides two new *model fields* - *tagulous.models.TagField* and *tagulous.models.SingleTagField*, which you use to add tags to your existing models to make them *tagged models*. They provide extra tag-related functionality.

They can also be queried like a normal Django `ForeignKey` or `ManyToManyField`, but with extra *query enhancements* to make working with tags easier.

Tags are stored in *tag model* subclasses, which can either be unique to each different tag field, or can be shared between them. If you don't specify a tag model on your field definition, one will be created for you automatically.

Tags can be nested using *tag trees*. There is also support for *database migrations*.

1.5.1 Model Fields

Tagulous offers two new model field types:

- *tagulous.models.TagField* - conventional tags using a `ManyToManyField` relationship.
- *tagulous.models.SingleTagField* - the same UI and functionality as a `TagField` but for a single tag, using a `ForeignKey` relationship.

These will automatically create the models for the tags themselves, or you can provide a custom model to use instead with `to` - see *Custom Tag Models* for more details.

Tagulous lets you get and set string values using these fields, while still leaving the underlying relationships available. For example, not only can you assign a queryset or list of tag primary keys to a `TagField`, but you can also assign a list of tag names, or a tag string to parse.

Like a `CharField`, changes made by assigning a value will not be committed until the model is saved, although you can still make immediate changes by calling the standard m2m methods `add`, `remove` and `clear`.

If `TAGULOUS_ENHANCE_MODELS` is `True` (which it is by default - see *Settings*), you can also use tag strings and lists of tag names in `get` and `filter`, and model constructors and `object.create()` - see *Tagged Models* for more details.

Model Field Arguments

The `SingleTagField` supports most standard `ForeignKey` arguments, except for `to_field` and `rel_class`.

The `TagField` supports most normal `ManyToManyField` arguments, except for `db_table`, `through` and `symmetrical`. Also note that `blank` has no effect at the database level, it is just used for form validation - as is the case with a normal `ManyToManyField`.

The `related_name` will default to `<field>_set`, as is normal for a `ForeignKey` or `ManyToManyField`. If using the same tag table on multiple fields, you will need to set this to something else to avoid clashes.

Auto-generating a tag model

If the *to argument* is not set, a tag model will be auto-generated for you. It will be given a class name based on the names of the tagged model and tag field; for example, the class name of the auto-generated model for `MyModel.tags` would be `Tagulous_MyModel_tags`.

When auto-generating a model, any *model option* can be passed as a field argument - see the *Automatic tag models* example.

If you want to override the default base class, for convenience you can specify a custom base class for the auto tag model - see the *to_base=MyTagModelBase* argument for details.

Specifying a tag model

You can specify the tag model for the tag field to use with the *to argument*. You cannot specify any tag options.

`to=MyTagModel` (or first unnamed argument)

Manually specify a tag model for this tag field. This can either be a reference to the tag model class, or string reference to it in the format `app.model`.

This will normally be a *custom tag model*, which must be a subclass of `tagulous.models.TagModel`.

It can also be a reference to a tag model already auto-generated by another tag field, eg `to=MyOtherModel.tags.tag_model`, although you must be confident that `MyOtherModel` will always be defined first.

It can also be a string containing the name of the tag model, eg `to='MyTagModel'`. However, this is resolved using Django's standard model name resolution, so you have to reference auto-generated models by their class name, not via the field - eg `to='otherapp.Tagulous_MyOtherModel_tags'`.

If the tagged model for this field is also a custom tag model, you can specify a recursive relationship as normal, using `'self'`.

If it is a custom tag model, it should have a *TagMeta* class. Fields which specify their tag model cannot provide new tag model options; they will take their options from the model - see *Tag Options* for more details.

This argument is optional; if omitted, a tag model will be *auto-generated* for you.

Default: `Tagulous_<ModelName>_<FieldName>` (auto-generated)

`to_base=MyTagModelBase`

You can specify a base class to use for an auto-generated tag model, instead of using `TagModel`.

This can be useful on complex sites where multiple auto-generated tag models need to share common custom functionality - for example, tracking and filtering by user who creates the tags. This argument will allow you to define one base class and re-use it across your project with less boilerplate than defining many empty custom tag models.

Default: `tagulous.models.TagModel`

`tagulous.models.SingleTagField`

Unbound field

An unbound `SingleTagField` (called on a model class, eg `MyModel.tag`) acts in the same way an unbound `ForeignKey` field would, but also has:

`tag_model` The related tag model

`tag_options` A *TagOptions* class, containing the options from the tag model's *TagMeta* or passed as arguments when initialising the field.

Bound to an instance

A bound `SingleTagField` (called on an instance, eg `instance.tags`) acts in a similar way to a bound `ForeignKey`, but with some differences:

Assignment (setter) A bound `SingleTagField` can be assigned a tag (an instance of the tag model) or a tag name.

If it is passed `None`, a current tag will be cleared if it is set.

The instance must be saved afterwards.

Example:

```
person.title = "Mr"
person.save()
```

Evaluation (getter) The value of a bound `SingleTagField` will return an instance of the tag model. The tag may not exist in the database yet (its `pk` may be `None`).

Example:

```
tag = person.title
report = "Tag %s used %d times " % (tag.name, tag.count)
```

The `tag_model` and `tag_options` attributes are not available on a bound field. If you only have an instance of the tagged model, you can access them by finding its class, eg `type(person).title.tag_model`.

`tagulous.models.TagField`

Unbound field

An unbound `TagField` (called on a model class, eg `MyModel.tags`) acts in the same way an unbound `ManyToManyField` would, but also has:

`tag_model` The related tag model

`tag_options` A *TagOptions* class, containing the options from the tag model's *TagMeta* or passed as arguments when initialising the field.

Bound to an instance

A bound `TagField` (called on an instance, eg `instance.tags`) acts in a similar way to a bound `ManyToManyField`, but with some differences:

Assignment (setter) A bound `TagField` can be assigned a tag string or an iterable of tags or tag names, eg a list of strings, or a queryset of instances of the tag model.

If it is passed `None`, any current tags will be cleared.

The instance must be saved afterwards.

Example:

```
person.skills = 'Judo, "Kung Fu"'
person.save()
```

Evaluation (getter) A bound `TagField` will return a *tagulous.models.TagRelatedManager* object, which has functions to get and set tag values.

tagulous.models.TagRelatedManager

A TagRelatedManager is a subclass of Django's standard RelatedManager, so you can do anything you would normally do with a bound ManyToManyField:

```
person.skills.get(name='judo')
tags = person.skills.all()
person.skills.add(MyTag)
person.skills.clear()
```

Because it's a relationship to a *tag model*, you can also filter by its fields:

```
filtered_tags = person.skills.filter(name__startswith='a')
popular_tags = person.skills.filter(count__gte=10)
```

A TagRelatedManager also provides access to the field's tag_model and tag_options:

```
person.skills.tag_model.objects.all()
is_lowercase = person.skills.tag_options.force_lowercase
```

It also provides the following additional methods:

set_tag_string(tag_string)

Sets the tags for this instance, given a tag string.

```
person.skills.set_tag_string('Judo, "Kung Fu"')
person.save()
```

set_tag_list(tag_list)

Sets the tags for this instance, given an iterable of tag names or tag instances.

```
person.skills.set_tag_list(['Judo', kung_fu_tag])
person.save()
```

get_tag_string()

Gets the tags as a tag string.

```
tag_string = person.skills.get_tag_string()
# tag_string == 'Judo, "Kung Fu"'
```

get_tag_list()

Returns a list of tag names.

```
tag_list = person.skills.get_tag_list()
# tag_list == ['Judo', 'Kung Fu']
```

__str__(), __unicode__()

Same as `get_tag_string`

```
report = '%s' % person.skills
```

__eq__, __ne__

Compare the tags on this instance to a tag string, or an iterable of tags or tag names. Order does not matter, and case sensitivity is determined by the options *case_sensitive* and *force_lowercase*.

```
if (
    first.tags == second.tags
    or first.tags == ['Judo', kung_fu_tag]
    or first.tags != 'foo, bar'
    or first.tags != second.tags.filter(name__istartswith='k')
):
    ...
```

__contains__

See if the tag (or string of a tag name) is in the tags. Case sensitivity is determined by the options *case_sensitive* and *force_lowercase*.

```
if 'Judo' in person.skills and kung_fu_tag in person.skills:
    candidates.append(person)
```

reload()

Discard any unsaved changes to the tags and load tags from the database

```
person.skills = 'judo'
person.save()
person.skills = 'karate'
person.skills.reload()
# person.skills == 'judo'
```

`save(force=False)`

Commit any tag changes to the database.

If you are only changing the tags you can call this directly to reduce database operations.

Note: You do not need to call this if you are saving the instance; the manager listens to the instance's save signals and saves any changes to tags as part of that process.

In most circumstances you can ignore the `force` flag:

- The manager has a `.changed` flag which is set to `False` whenever the internal tag cache is loaded or saved. It is set to `True` when the tags are changed without being saved.
- If `force=False` (default), this method will only update the database if the `.changed` flag is `True` - in other words, the database will only be updated if there are changes to the internal cache since last load or save.
- If `force=True`, the `.changed` flag will be ignored, and the current tag status will be forced upon the database. This can be useful in the rare cases where you have multiple references to the same database object, and want the tags on this instance to override any changes other instances may have made.

For example:

```
person = Person.objects.create(name='Adam', skills='judo')
person.name = 'Bob'
person.skills = 'karate'
person.skills.save()
# person.name == 'Adam'
# person.skills == 'judo'
```

`add(tag, tag, ...)`

Based on the normal `RelatedManager.add` method, but has support for tag names.

Adds a list of tags or tag names directly to the instance - there is no need to save afterwards.

Note: This does not parse tag strings - you need to pass separate tags as either instances of the tag model, or as separate strings.

Will call `reload()` first, so any unsaved changes to tags will be lost.

```
person.skills.add('Judo', kung_fu_tag)
```

remove(tag, tag, ...)

Based on the normal `RelatedManager.remove` method, but has support for tag names.

Removes a list of tags or tag names directly from the instance - there is no need to save afterwards.

Note: This does not parse tag strings - you need to pass separate tags as either instances of the tag model, or as separate strings.

Will call `reload()` first, so any unsaved changes to tags will be lost.

```
person.skills.remove('Judo', kung_fu_tag)
```

1.5.2 Tag Models

Tags are stored in tag models which subclass *tagulous.models.TagModel*, and use a *tagulous.models.TagModelManager*. A tag model can either be generated automatically, or you can create a *custom model*.

Tags in tag models can be *protected* from automatic deletion when they are not referred to. *Initial tags* must be loaded using the *initial_tags command*.

Tag model classes

tagulous.models.TagModel

A `TagModel` subclass has the following fields and methods:

name

A `CharField` containing the name (string value) of the tag.

Must be unique.

slug

A unique `SlugField`, generated automatically from the name when first saved.

Slugs will support unicode if the `TAGULOUS_SLUG_ALLOW_UNICODE` *setting* is `True`. Empty slugs are not allowed; they will default to underscore. Slug clashes are avoided by adding an integer to the end.

`count`

An `IntegerField` holding the number of times this tag is in use.

`protected`

A `BooleanField` indicating whether this tag should be protected from deletion when the count reaches 0.

It also has several methods primarily for internal use, but some may be useful:

`get_related_objects()`

Return a list of instances of other models which refer to this tag; see the API for more details

`update_count()`

In case you're doing something weird which causes the count to get out of sync, call this to update the count, and delete the tag if appropriate.

`merge_tags(tags)`

Merge the specified tags into this tag.

`tags` can be a queryset, list of tags or tag names, or a tag string.

`tagulous.models.TagModelManager`

A `TagModelManager` is the standard manager for a *tagulous.models.TagModel*; it is a subclass of the normal Django model manager, but its queries return a *tagulous.models.TagModelQuerySet* instead.

It also provides the following additional methods:

`filter_or_initial(...)`

Calls the normal `filter(...)` method, but then adds on any initial tags which may be missing.

`weight(min=1, max=6)`

Annotates a `weight` field to the tags. This is a weighted count between the specified `min` and `max`, which default to `TAGULOUS_WEIGHT_MIN` and `TAGULOUS_WEIGHT_MAX` (see *Settings*).

This can be used to generate *tag clouds*, for example.

tagulous.models.TagModelQuerySet

This is returned by the *tagulous.models.TagModelManager*; it is a subclass of the normal Django `QuerySet` class, but implements the same additional methods as the `TagModelManager`.

Custom Tag Models

A custom tag model should subclass `tagulous.models.TagModel`, so that Tagulous can find the fields and methods it expects, and so it uses the appropriate tag model manager and queryset.

A custom tag model is a normal model in every other way, except it can have a *TagMeta* class to define default options for the class.

There is *an example* which illustrates how to create a custom tag model.

If you want to use tag trees, you will need to subclass `tagulous.models.TagTreeModel` instead. The only difference is that there will be extra fields on the model - see *Tag Trees* for more details.

TagMeta

The `TagMeta` class is a container for tag options, to be used when creating a custom tag model.

Set any *Model Options* as class properties. When the model is created by Python, the options will be available on the tag model class and tag fields which use it as `tag_options`.

Tag fields will not be able to override these options, and `SingleTagField` fields will ignore `max_count`.

If `tree` is specified, it must be appropriate for the base class of the tag model, eg if `tree=True` the tag model must subclass *tagulous.models.TagTreeModel* - but if it is not provided it will be set to the correct value.

`TagMeta` can be inherited, so it can be set on abstract models. Options in the `TagMeta` of a parent model can be overridden by options in the `TagMeta` of a child model.

Example:

```
import tagulous
class MyTagModel(tagulous.models.TagModel):
    class TagMeta:
        initial = 'judo, karate'
```

Protected tags

The tag model keeps a count of how many times each tag is referenced. When the tag count reaches `0`, the tag will be deleted unless its `protected` field is `True`, or the `protect_all` option has been used.

Note: This only happens when the count is updated, ie when the tag is added or removed; tags can therefore be created directly on the model with the default count of `0`, ready to be assigned later.

Loading initial tags

Initial tags must be loaded using the `initial_tags` management command. You can either load all initial tags in your site by not passing in any arguments, or specify an app, model or field to load:

```
python manage.py initial_tags [<app_name>[.<model_name>[.<field_name>]]]
```

- Tags which are new will be created
- Tags which have been deleted will be recreated
- Tags which exist will be untouched

1.5.3 Tag Trees

Tags can be nested using tag trees for detailed categorisation, with tags having parents, children and siblings.

Tags in tag trees denote parents using the forward slash character (`/`). For example, `Animal/Mammal/Cat` is a `Cat` with a parent of `Mammal` and grandparent of `Animal`.

To use a slash in a tag name, escape it with a second slash; for example the tag name `Animal/Vegetable` can be entered as `Animal//Vegetable`.

A custom tag tree model must be a subclass of *tagulous.models.TagTreeModel* instead of the normal *tagulous.models.TagModel*; for automatically-generated tag models, this is managed by setting the *tree* field option to `True`.

Tag Tree Model Classes

`tagulous.models.TagTreeModel`

Because tree tag names are fully qualified (include all ancestors) and unique, there is no difference to normal tags in how they are set or compared.

A `TagTreeModel` subclasses *tagulous.models.TagModel*; it inherits all the normal fields and methods, and adds the following:

Note: Field values are computed and set automatically in the `save()` method - so don't try to use them until the tag has been saved.

`parent`

A `ForeignKey` to the parent tag. Tagulous sets this automatically when saving, creating missing ancestors as needed.

`children`

The reverse relation manager for `parent`, eg `mytag.children.all()`.

`label`

A `CharField` containing the name of the tag without its ancestors.

Example: a tag named `Animal/Mammal/Cat` has the label `Cat`

`slug`

A `SlugField` containing the slug for the tag label.

Example: a tag named `Animal/Mammal/Cat` has the slug `cat`

`path`

A `TextField` containing the path for this tag - this slug, plus all ancestor slugs, separated by the `/` character, suitable for use in URLs. Tagulous sets this automatically when saving.

Example: a tag named `Animal/Mammal/Cat` has the path `animal/mammal/cat`

`level`

An `IntegerField` containing the level of this tag in the tree (starting from 1).

`merge_tags(tags, children=False)`

Merge the specified tags into this tag.

`tags` can be a queryset, list of tags or tag names, or a tag string.

If `children=False`, only the specified tags will be merged; tagged items will be reassigned to this tag, but if there are child tags they will not be touched. If child tags do exist, although the merged tags' counts will be 0, they will not be cleared.

If `children=True`, child tags will be merged into children of this tag, retaining structure; eg merging `Pet` into `Animal` will merge `Pet/Mammal` into `Animal/Mammal`, `Pet/Mammal/Cat` into `Animal/Mammal/Cat` etc. Tags will be created if they don't exist.

`get_ancestors()`

Returns a queryset of all ancestors, ordered by level.

`get_descendants()`

Returns a queryset of all descendants, ordered by level.

`get_siblings()`

Returns a queryset of all siblings, ordered by name.

This includes the node itself; if you don't want it in the results, exclude it afterwards, eg:

```
siblings = node.get_siblings().exclude(pk=node.pk)
```

`tagulous.models.TagTreeModelManager`

A `TagTreeModelManager` is the standard manager for a *tagulous.models.TagTreeModel*; it is a subclass of *tagulous.models.TagModelManager* so provides those methods, but its queries return a *tagulous.models.TagTreeModelQuerySet* instead.

`tagulous.models.TagTreeModelQuerySet`

This is returned by the *tagulous.models.TagTreeModelManager*; it is a subclass of *tagulous.models.TagModelQuerySet* so provides those methods, but also:

`with_ancestors()`

Returns a new queryset containing the nodes from the calling queryset, plus their ancestor nodes.

`with_descendants()`

Returns a new queryset containing the nodes from the calling queryset, plus their descendant nodes.

`with_siblings()`

Returns a new queryset containing the nodes from the calling queryset, plus theirm sibling nodes.

Converting from to tree tags from normal tags

When converting from a normal tag model to a tag tree model, you will need to add extra fields. One of those (`path`) is a unique field, which means extra steps are needed to build the migration.

These instructions will convert an existing `TagModel` to a `TagTreeModel`. Look through the code snippets and change the app and model names as required:

1. Create a data migration to escape the tag names.

   You can skip this step if you have been using slashes in normal tags and want them to be converted to nested tree nodes.

   Run `manage.py makemigrations myapp --empty` and add:

```
def escape_tag_names(apps, schema_editor):
    model = apps.get_model('myapp', 'Tagulous_MyModel_Tags')
    for tag in model.objects.all():
        tag.name = tag.name.replace('/', '//')
        tag.save()
operations = RunPython(escape_tag_names)
```

2. Create a schema migration to change the model fields. Because paths are not allowed to be null, you need to add the `path` field as a non-unique field, set some unique data on it (such as the object's `pk`), and then change the field to add back the unique constraint.

   To do this reliably on all database types, see Migrations that add unique fields in the official Django documentation.

   If you are only working with databases which support transactions, you can use a tagulous helper to add the unique field:

   1. When you create the migration, Django will prompt you for a default value for the unique `path` field; answer with `'x'` (do the same for the `label` field when asked).

      Change the new migration to use the Tagulous helper to add the `path` field.

   2. Add the unique field:

   ```
   import tagulous.models.migrations
   ...

   class Migration(migrations.Migration):
       # ... rest of Migration as generated
       operations = [
           ...
           # Leave other operations as they are, just replace AddField:
       ] + tagulous.models.migration.add_unique_field(
           model_name='_tagulous_mymodel_tags',
           name='path',
           field=models.TextField(unique=True),
           preserve_default=False,
           set_fn=lambda obj: setattr(obj, 'path', str(obj.pk)),
       ) + [
           ...
       ]
   ```

   Warning: Although `add_unique_column` and `add_unique_field` do work with non-transactional databases, it is not without risk. See *Database Migrations* for more details.

3. We have changed the abstract base class of the tag model, but Django migrations have no native way to do this. You will need to use the Tagulous helper operation `ChangeModelBases` to do it manually, otherwise future data migrations will think it is a `TagModel`, not a `TagTreeModel`.

   Modify the migration from step 2; if you followed the official Django documentation and have several migrations, modify the last one. Add the `ChangeModelBases` to the end of your `operations` list, as the last operation:

```
import tagulous.models.migrations

```

(continues on next page)

(continued from previous page)

```
class Migration(migrations.Migration):
    # ... rest of Migration as generated
    operations = [
        # ... rest of operations
        tagulous.models.migrations.ChangeModelBases(
            name='_tagulous_mymodel_tags',
            bases=(tagulous.models.models.BaseTagTreeModel, models.Model),
        )
    ]
```

4. Create another data migration to rebuild the tag model and set the paths:

```
def rebuild_tag_model(apps, schema_editor):
    model = apps.get_model('myapp', 'Tagulous_MyModel_Tags')
    model.objects.rebuild()
operations = RunPython(rebuild_tag_model)
```

   If you skipped step 1, this will also create and set parent tags as necessary.

5. Run the migrations

You can see a working migration using steps 2 and 3 in the Tagulous tests, for Django migrations.

1.5.4 Tagged Models

Models which have tag fields are called tagged models. In most situations, all you need to do is add the tag field to the model and Tagulous will do the rest.

Because Tagulous's fields work by subclassing `ForeignKey` and `ManyToManyField`, there are some places in Django's models where you would expect to use tag strings but cannot - constructors and filtering, for example. Tagulous therefore adds this functionality through the *tagulous.models.TaggedModel* base class for tagged models.

If `TAGULOUS_ENHANCE_MODELS = True` (which it is by default - see *Settings*), this base class will be applied automatically, otherwise read on to *Setting tagged base classes manually*.

Note: Tagulous sets `TaggedModel` as the base class for your existing tagged model by listening for the `class_prepared` signal, sent when a model has been constructed. If the model contains tag fields, Tagulous will dynamically add `TaggedModel` to the model's base classes and `TaggedManager` to the manager's base classes, which in turn adds `TaggedQuerySet` to the querysets the manager creates. It does this by calling the `cast_class` class method on each of the base classes, which change the original classes in place.

This all happens seamlessly behind the scenes; the only thing you may notice is that the names of your manager and queryset classes now have the prefix `CastTagged` to indicate that they have been automatically cast to their equivalents for tagged models.

Tagged model classes

`tagulous.models.TaggedModel`

This is the base class for all tagged models. It changes the model constructor so that `TagField` values can be passed as keywords.

`tagulous.models.TaggedManager`

The base class for managers of tagged models. It only exists to ensure querysets are subclasses of `tagulous.TaggedQuerySet`.

`tagulous.models.TaggedQuerySet`

The base class for querysets on tagged models. It changes `get`, `filter` and `exclude` to work with string values, and `create` and `get_or_create` to work with string and `TagField` values.

It also adds `get_similar_objects()` - see finding_similar_objects for usage.

See *Querying using tag fields* for more details.

Setting tagged base classes manually

However, if you want to avoid this automatic subclassing, you can set `TAGULOUS_ENHANCE_MODELS = False` and manage this yourself:

The three tagged base classes each have a class method `cast_class` which can change existing classes so that they become `CastTagged` subclasses of themselves; for example:

```
class MyModel(tagulous.TaggedModel):
    name = models.CharField(max_length=255)
    tags = tagulous.models.TagField()
    objects = tagulous.models.TaggedManager.cast_class(MyModelManager)
    other_manager = MyOtherManager
tagulous.models.TaggedManager.cast_class(MyModel.other_manager)
```

This can be useful when working with other third-party libraries which insist on you doing things a certain way.

Querying using tag fields

When querying a tagged model, remember that a `SingleTagField` is really a `ForeignKey`, and a `TagField` is really a `ManyToManyField`. You can query using these relationships in conventional ways.

If you have correctly made your tagged model subclass *tagulous.models.TaggedModel*, you can also compare a tag field to a tag string in `get`, `filter` and `exclude`:

```
qs = MyModel.objects.get(name="Bob", title="Mr", tags="red, blue, green")
```

When querying a tag field, case sensitivity will default to whatever the tag field option was. For example, if the `title` tag field above was defined with `case_sensitive=False`, `.filter(title='Mr')` will match `Mr`, `mr` etc.

Note that when querying a `TagField` in this way, the returned queryset will include (or exclude) any object which contains all the specified tags - but it may also have other tags. To only return objects which have the specified tags and no others, use the `__exact` field lookup suffix:

```
# Find all MyModel objects which have the tag 'red':
qs = MyModel.objects.filter(tags='red')
# (will include those tagged 'red, blue' etc)

# Find all MyModel objects which are only tagged 'red':
qs = MyModel.objects.filter(tags__exact='red')
# (will not include those tagged 'red, blue')
```

This currently does not work across database relations; you will need to use the `name` field on the tag model for those:

```
# Find
qs = MyRelatedModel.objects.filter(
    foreign_model__tags__name__in=['red', 'blue', 'green'],
)
```

Because tag fields use standard database relationships, you can easily filter the tags by other fields in your model.

For example, if your model `Record` has a `tags` TagField and an `owner` foreign key to `auth.User`, to get a list of tags which that user has used:

```
myobj.tags.tag_model.objects.filter(record__owner=user)
```

There is a `filter_or_initial` helper method on a `TagModel`'s manager and queryset, which will add initial tags to your filtered queryset:

```
myobj.tags.tag_model.objects.filter_or_initial(record__owner=user)
```

The QuerySet on a tagged model provides the method `get_similar_objects`, which takes the instance and field name to compare similarity by, and returns a queryset of similar objects from that tagged model, ordered by similarity:

```
myobj = MyModel.objects.first()
similar = MyModel.objects.get_similar_objects(myobj, 'tags')
```

There is a convenience wrapper on the related manager which detects the instance and field to compare by:

```
similar = myobj.tags.get_similar_objects()
```

Although less useful, there is a similar function for single tag fields, which finds all objects with the same tag:

```
similar = myobj.singletag.get_similar_objects()
```

The similar querysets will exclude the object being compared - in the above examples, `myobj` will not be in the queryset.

1.5.5 Database Migrations

Tagulous supports Django migrations.

Both `SingleTagField` and `TagField` work in schema and data migrations. Tagged models will be subclasses of `TaggedModel` as normal (as long as `TAGULOUS_ENHANCE_MODELS` is `True`), and tag fields will work as normal. The only difference is that tag models will be instances of `BaseTagModel` and `BaseTagTreeModel` rather than their normal non-base versions - but this is just how migrations work, and it will makes no practical difference.

Adding unique columns

Migrating a model to a `TagModel` or `TagTreeModel` involves adding unique fields (`slug` and `path` for example), which normally requires 3 separate migrations. To simplify this process, Tagulous provides the helper method `add_unique_field` to add them in a single migration - see step 2 in *Converting from to tree tags from normal tags* for examples of their use.

However, use these with care - should part of the function fail for some reason when using a non-transactional database, it won't be able to roll back and may be left in an unmigrateable state. It is therefore recommended that you either make a backup of your database before using this function, or that you follow the steps in the official Django documentation to perform the action in 3 separate migrations.

Limitations of Django migrations

Django migrations do not support changing the tag model's base class - for example, changing a plain model to a `TagModel`, or a `TagModel` to a `TagTreeModel`). Django migrations have no way to store or apply this change, so you will need to use the Tagulous helper operation `ChangeModelBases` - see step 3 of *Converting from to tree tags from normal tags* for more details, or the working example in 0003_tree.py.

Django migrations also cannot serialise lambda expressions, so the `get_absolute_url` argument is not available during data migrations, neither when defined on a tag field, nor when in a tag model. If you need to call this in a data migration, it is recommended that you embed the logic into your migration.

1.6 Forms

Normally tag fields will be used in a `ModelForm`; they will automatically use the correct form field and widget to render tag fields with your selected *autocomplete adaptor*.

To save tag fields, just call the `form.save()` method as you would normally. However, because *tagulous.models.TagField* is based on a `ManyToManyField`, if you call `form.save(commit=False)` you will need to call `form.m2m_save()` after to save the tags.

See the *ModelForms* example for how this works in practice.

1.6.1 Form field classes

You can also use Tagulous form fields outside model forms by using the *tagulous.forms.SingleTagField* and *tagulous.forms.TagField* form fields - see the *Forms without models* example for how this works in practice.

Tag forms fields take standard Django core field arguments such as `label` and `required`.

`tagulous.forms.SingleTagField`

This field accepts two new arguments:

`tag_options` A *TagOptions* instance, containing *form options* (model options will be ignored).

`autocomplete_tags` An iterable of tags to be embedded for autocomplete. This can either be a queryset of tag objects, or a list of tag objects or strings.

The `clean` method returns a single tag name string, or `None` if the value is empty.

`tagulous.forms.TagField`

This field accepts the same two new arguments as a `SingleTagField`:

`tag_options` A *TagOptions* instance, containing *form options* (model options will be ignored).

`autocomplete_tags` An iterable of tags to be embedded for autocomplete. This can either be a queryset of tag objects, or a list of tag objects or strings.

The `clean` method returns a sorted list of unique tag names (a list of strings) - or an empty list if there are no tags.

`tagulous.forms.TaggedInlineFormSet`

In most cases Tagulous works with Django's default inline model formsets, and you don't need to do anything special.

However, there is a specific case where it doesn't: when you create an inline formset for tagged models, with a tag as their parent model - eg when you edit a tag and its corresponding instances of the tagged model. That is when you must use the `TaggedInlineFormSet` class. For example:

```
class Person(models.Model):
    name = models.CharField(max_length=255)
    title = tagulous.models.SingleTagField(initial='Mr, Mrs')

PersonInline = forms.models.inlineformset_factory(
    Person.title.tag_model,
    Person,
    formset=tagulous.forms.TaggedInlineFormSet,
)
```

This would allow you to generate a formset for all `Person` objects which use a specific `title` tag.

Tagulous will automatically apply this fix in the admin site, as long as the tag admin class is registered using `tagulous.admin.register`.

Without the `TaggedInlineFormSet` class in this situation, the tag count will be incorrect when adding tagged model instances, and editing will fail because the default formset will try to use the tag name as a primary key.

The `TaggedInlineFormSet` class will only perform actions under this specific relationship, so is safe to use in other situations.

1.6.2 Filtering autocomplete tags

By default the tag field widget will autocomplete using all tags on the tag model. However, you will often only want to use a subset of your tags - for example, just the initial tags, or tags which the current user has used, or tags which have been used in conjunction with another field on your model.

Because model tag fields are normal Django relationships, you can filter embedded autocomplete tags by overriding the form's `__init__` method. To filter an ajax autocomplete view, wrap `tagulous.views.autocomplete` in your own view function which filters for you.

For examples of these approaches, see *Filtering embedded autocomplete* and *Filtering an autocomplete view*.

1.6.3 Autocomplete Adaptors

Tagulous uses a javascript file it calls an `adaptor` to apply your chosen autocomplete library to the Tagulous form field.

Only Select2 is included with Tagulous; if you want to use a different library, you will need to add it to your project's static files, and add the relative path under `STATIC_URL` to the appropriate `TAGULOUS_` settings.

Tagulous includes the following adaptors:

Select2 (version 4)

The default adaptor, for Select2.

Path: `tagulous/adaptor/select2-4.js`

Autocomplete settings should be a dict:

`defer` If `True`, the tag field will not be initialised automatically; you will need to call `Tagulous.select2(el)` on it from your own javascript. This is useful for fields which are used as templates to dynamically generate more.

For example, to use this adaptor with a django-dynamic-formset which uses a `formTemplate`, set `{'defer': True}`, then configure the formset with:

```
added: function ($row) {
    Tagulous.select2($row.find('input[data-tagulous]'));
}
```

Note that when used with inline formsets which raise the `formset:added` event (like in the Django admin site), Tagulous will automatically try to register tag fields in new formsets if `defer=False`.

`width` This is the same as in Select2's documentation, but the Tagulous default is `resolve` instead of `off`, for the best chance of working without complication.

All other settings will be passed to the Select2 constructor.

1.6.4 Writing a custom autocomplete adaptor

Writing a custom adaptor should be fairly self-explanatory - take a look at the included adaptors to see how they work. It's mostly just a case of pulling data out of the HTML field, and fiddling with it a bit to pass it into the library's constructor.

Tagulous puts certain settings on the HTML field's `data-` attribute:

`data-tagulous` Always `true` - used to identify a tagulous class to JavaScript

`data-tag-type` Set to `single` when a `SingleTagField`, otherwise not present.

`data-tag-list` JSON-encoded list of tags.

`data-tag-url` URL to request tags

`data-tag-options` JSON-encoded dict of tag options

> In addition to the dict from `TagOptions` containing the field's *Form Options*, there will also be:
>
> **`required`** A boolean indicating whether the form field is required or not

These settings can be used to initialise your autocomplete library of choice. You should initialise it using `data-tag-options`'s `autocomplete_settings` for default values.

For consistency with Tagulous's *python parser*, try to replace your autocomplete library's parser with Tagulous's *javascript parser*.

If you write an adaptor which you think would make a good addition to this project, please do send it in or make a pull request on github - see *Contributing* for more information.

1.7 Tag Options

Model options define how a tag model behaves. They can either be set in the *model field arguments*, or in the tag model's *TagMeta* class. Once defined, they are then stored in a *TagOptions* instance on the tag model, accessible at `MyTagModel.tag_options` (and shared with tag model fields at `MyTaggedModel.tags.tag_options`).

Tagulous only lets you set options for a tag model in one place - if you use a custom model you must set options using `TagMeta`, and if you share an auto-generated model between fields, only the first field can set options.

Form options are a subset of the model options, and are also used to control tag form fields, and are also stored in a `TagOptions` instance. If the field is part of a `ModelForm` it will inherit options from the model, otherwise options can be passed in the field arguments.

1.7.1 Model Options

The tag model options are:

`initial`

List of initial tags for the tag model. Must be loaded into the database with the management command *initial_tags*.

Value can be a tag string to be parsed, or an array of strings with one tag in each string.

To change initial tags, you can change the `initial` option and re-run the command *initial_tags*.

You should not find that you need to update `initial` regularly; if you do, it would be better to use the Tagulous *admin tools* to add tags to the model directly.

If provided as a tag string, it will be parsed using spaces and commas, regardless of the *space_delimiter* option.

Default: `''`

`protect_initial`

The `protected` state for any tags created by the `initial` argument - see *Protected tags*.

Default: `True`

`protect_all`

Whether all tags with count 0 should be protected from automatic deletion.

If false, will be decided by `tag.protected` - see *Protected tags*.

Default: `False`

`case_sensitive`

If `True`, tags will be case sensitive. For example, `"django, Django"` would be two separate tags.

If `False`, tags will be capitalised according to the first time they are used.

Note when using sqlite: substring matches on tag names, and matches on tag names with non-ASCII characters, will never be case sensitive - see the databases django documentation for more information.

See also *force_lowercase*

Note: MySQL struggles to offer string case sensitivity at the database level - see the django documentation for more details. Tagulous therefore offers no formal support for this option when running on MySQL - the relevant tests are bypassed, and you should assume that `case_sensitive` is always `False`. Patches welcome.

Default: `False`

`force_lowercase`

Force all tags to lower case

Default: `False`

`max_count`

`TagField` only - this is not supported by `SingleTagField`.

Specifies the maximum number of tags allowed.

Set to `0` to have no limit.

If you are setting it to 1, consider using a `SingleTagField` instead.

Default: `0`

`space_delimiter`

`TagField` only - this is not supported by `SingleTagField`.

If `True`, both commas and spaces can be used to separate tags. If `False`, only commas can be used to separate tags.

Default: `True`

`tree`

Field argument only - this cannot be set in *TagMeta*

If `True`, slashes in tag names will be used to denote children, eg `grandparent/parent/child`, and these relationships can be traversed. See *Tag Trees* for more details.

If `False`, slashes in tag names will have no significance, and no tree properties or methods will be present on tag objects.

Default: `False`

`autocomplete_initial`

If `True`, override all other autocomplete settings and use the tags defined in the `initial` argument for autocompletion, embedded in the form field HTML.

For more advanced autocomplete filtering options (ie filter tags by user), see the example *Filtering autocomplete by related fields*.

Default: `False`

`autocomplete_view`

Specify the view to use for autocomplete queries.

This should be a value which can be passed to Django's `reverse()`, eg the name of the view.

If `None`, all tags will be embedded into the form field HTML as the `data-autocomplete` attribute.

If this is an invalid view, a `ValueError` will be raised.

Default: `None`

`autocomplete_view_args`

Optional `args` passed to the `autocomplete_view`.

Default: `None`

autocomplete_view_kwargs

Optional kwargs passed to the autocomplete_view.

Default: None

autocomplete_limit

Maximum number of tags to provide at once, when autocomplete_view is set.

If the autocomplete adaptor supports pages, this will be the number shown per page, otherwise any after this limit will not be returned.

If 0, there will be no limit and all results will be returned

Default: 100

autocomplete_view_fulltext

Whether to perform a start of word match (__startswith) or full text match (__contains) in the autocomplete view.

Has no effect if not using autocomplete_view.

Default: False (start of word)

autocomplete_settings

Override the default TAGULOUS_AUTOCOMPLETE_SETTINGS.

For example, the select2 control defaults to use the same width as the form element it replaces; you can override this by passing their width option (see their docs on appearance) as an autocomplete setting:

```
myfield = TagField(... autocomplete_settings={"width": "75%"})
```

Default: None

get_absolute_url

A shortcut for defining a get_absolute_url method on the tag model. Only used when defined in tag fields which auto-generate models.

It is common to need to get a URL for a tag, so rather than converting your tag field to use a custom TagModel just to implement a get_absolute_url method, you can pass this argument a callback function.

The callback function will be passed the tag object, and should return the URL for the tag. See the *Tag URL* example for a simple lambda argument.

If not set, the method get_absolute_url will not be available and an AttributeError will be raised.

Note: Due to the way Django migrations freeze model fields, this attribute is not available during data migrations. See *Limitations of Django migrations* for more information.

Default: None

verbose_name_singular, verbose_name_plural

When a tag model is auto-generated from a field, it is given a `verbose_name` based on the tagged model's name and the tag field's name; the `verbose_name_plural` is the same, but with an added `s` at the end. This is primarily used in the admin.

However, this will sometimes not make grammatical sense; these two arguments can be used to override the field name component of the model name.

The `verbose_name_singular` will usually be used with a `TagField` - for example, the auto-generated model for `MyModel.tags` will have the singular name `My model tags`; this can be corrected by setting `verbose_name_singular="tag"` in the field definition.

The `verbose_name_plural` will usually be used with a `SingleTagField` - for example, the auto-generated model for `MyModel.category` will have the plural name `My model categorys`; this can be corrected by setting `verbose_name_plural="categories"` in the field definition.

If one or both of these are not set, Tagulous will try to find the field name from its `verbose_name` argument, falling back to the field name.

Note: When Tagulous automatically generates verbose names, it intentionally performs no checks on how long they will be. When Django attempts to create permissions for the model, if the generated verbose name is longer than 39 characters, it may raise a `ValidationError`. To resolve this, set `verbose_name_singular` to a string which is 38 characters or less.

1.7.2 Form Options

The following options are used by form fields:

- *case_sensitive*
- *force_lowercase*
- *max_count*
- *tree*
- *autocomplete_limit*
- *autocomplete_settings*

1.7.3 The TagOptions Class

The `TagOptions` class is a simple container for tag options. The options for a model field are available from the `tag_options` property of unbound *tagulous.models.SingleTagField* or *tagulous.models.TagField* fields.

All options listed in *Model Options* are available directly on the object, except for `to`. It also provides two instance methods:

`items(with_defaults=True)` Get a dict of all options

If with_defaults is true, any missing settings will be taken from the defaults in `constants.OPTION_DEFAULTS`.

`form_items(with_defaults=True)` Get a dict of just the options for a form field.

If with_defaults is true, any missing settings will be taken from the defaults in `constants.OPTION_DEFAULTS`.

Example:

```
initial_tags = MyModel.tags.tag_options.initial
if "force_lowercase" in MyModel.tags.tag_options.items():
    ...
```

`TagOptions` instances can be added together to create a new merged set of options; note though that this is a shallow merge, ie the value of `autocomplete_settings` on the left will be replaced by the value on the right:

```
merged_options = TagOptions(
    autocomplete_settings={'width': 'resolve'}
) + TagOptions(
    autocomplete_settings={'allowClear': True}
)
# merged_options.autocomplete_settings == {'allowClear': True}
```

In the same way, setting `autocomplete_settings` on the field will replace any default value.

1.8 Views and Templates

1.8.1 Form templates

To render Tagulous fields in forms outside the admin site, add `{{ form.media }}` to your template to include the JavaScript and CSS resources; for example:

```
{% block content %}
    {{ form.media }}
    {{ form }}
{% endblock %}
```

For an example of adding the JavaScript and CSS separately, see the example project templates

1.8.2 Autocomplete views

Although Tagulous doesn't need any views by default, it does provide generic views in tagulous/views.py to support AJAX autocomplete requests.

`response = autocomplete(request, tag_model)` This takes the request object from the dispatcher, and a reference to the tag model which this is autocompleting.

You can also pass in a QuerySet of the tag model, instead of the tag model itself, in order to filter the tags which will be returned.

It returns an `HttpResponse` with content type `application/json`. The response content is a JSON-encoded object with one key, `results`, which is a list of tags.

`response = autocomplete_login(request, tag_model)` Same as `autocomplete`, except is decorated with Django auth's `login_required`.

These views look for two GET parameters:

`q` A query string to filter results by - used to match against the start of the string.

Note: if using a sqlite database, matches on a case sensitive tag model may not be case sensitive - see the *case_sensitive* option for more details.

p The page number to return, if *autocomplete_limit* is set on the tag model.

Default: 1

For an example, see the *Autocomplete AJAX Views* example.

1.8.3 Tag clouds

Tag clouds are a common way to display tags. Rather than have a template tag with templates and options for every eventuality, Tagulous simply offers a *weight()* method on tag querysets, which adds a `weight` annotation to tag objects:

```
# myapp/view.py
def tag_cloud(request):
    ...
    tags = MyModel.tags.tag_model.objects.weight()
    ...
```

The `weight` value will be a number between `TAGULOUS_WEIGHT_MIN` and `TAGULOUS_WEIGHT_MAX` (see *Settings*), although these can be overridden by passing arguments to `weight()` for new min and/or max values, eg:

```
tags = TagModel.objects.weight(min=2, max=4)
```

You can then render the tag cloud in your template as any other queryset, with complete control over how they are displayed:

```
{% if tags %}
    <h2>Tags</h2>
    <p>
    {% for tag in tags %}
        <a href="{{ tag.get_absolute_url }}" class="tag_{{ tag.weight }}">
            {{ tag.name }}
        </a>
    {% endfor %}
{% endif %}
```

In that example, you would then define CSS classes for `tag_1` to `tag_6`, which set the appropriate font styles.

If you wanted to insert the tag cloud on every page, it would be easy to wrap up in a custom template tag:

```
# myapp/templatetags/myapp_tagcloud.py
from django import template
from myapp import models

register = template.Library()
@register.inclusion_tag('myapp/include/tagcloud.html')
def show_results(poll):
    tags = models.MyModel.tags.tag_model.objects.weight()
    return {'tags': tags}

# myapp/templates/tagcloud.html - see template example above
```

1.9 Admin

1.9.1 Tag fields in ModelAdmin

To support TagField and SingleTagField fields in the admin, you need to register the Model and ModelAdmin using Tagulous's `register()` function, instead of the standard one:

```
import tagulous.admin
class MyAdmin(admin.ModelAdmin):
    list_display = ['name', 'tags']
tagulous.admin.register(MyModel, MyAdmin)
```

This will make a few changes to `MyAdmin` to add tag field support (detailed below), and then register it with the default admin site using the standard `site.register()` call.

As with the normal registration call, the admin class is optional:

```
tagulous.admin.register(myModel)
```

You can also pass a custom admin site into the *register()* function:

```
# These two lines are equivalent:
tagulous.admin.register(myModel, MyAdmin)
tagulous.admin.register(myModel, MyAdmin, site=admin.site)
```

The changes Tagulous's `register()` function makes to the `ModelAdmin` are:

- Changes your `ModelAdmin` to subclass `TaggedAdmin`
- Checks `list_display` for any tag fields, and adds functions to the `ModelAdmin` to display the tag string (unless an attribute with that name already exists)
- Switches an inline class to a `TaggedInlineFormSet` when necessary

Note:

- You can only provide the Tagulous `register()` function with one model.
- The admin class will be modified; bear that in mind if registering it with multiple admin sites. In that case, you may want to enhance the class manually, as described below.

1.9.2 Manually enhancing your ModelAdmin

The `tagulous.admin.register` function is the short way to enhance your admin classes. If for some reason you can't use it (eg another library which has its own `register` function, or you're registering it with more than one admin site), you can do what it does manually:

1. Change your admin class to subclass `tagulous.admin.TaggedModelAdmin`.

   This disables Django's green button to add a related field, which is incompatible with Tagulous.

2. Call `tagulous.admin.enhance(model_class, admin_class)`.

   This finds the tag fields on the model class, and adds support for them to `list_display`.

3. Register the admin class as normal

For example:

```
import tagulous
class MyAdmin(tagulous.admin.TaggedModelAdmin):
    list_display = ['name', 'tags']
tagulous.admin.enhance(MyModel, MyAdmin)
admin.site.register(MyModel, MyAdmin)
```

1.9.3 Autocomplete settings

The admin site can use different autocomplete settings to the public site by changing the settings `TAGULOUS_ADMIN_AUTOCOMPLETE_JS` and `TAGULOUS_ADMIN_AUTOCOMPLETE_CSS`. You may want to do this to avoid jQuery being loaded more than once, for example - assuming the version in Django's admin site is compatible with the autocomplete library of your choice.

See *Settings* for more information.

Because the select2 control defaults to use the same width as the form element it replaces, you may find this a bit too small in some versions of the Django admin. You could override this with *autocomplete_settings*, but that will change non-admin controls too, so the best option would be to add a custom stylesheet to `TAGULOUS_ADMIN_AUTOCOMPLETE_CSS` with a rule such as:

```
.select2 {
    width: 75% !important;
}
```

1.9.4 Managing the tag model

Tagulous provides additional tag-related functionality for tag models, such as the ability to merge tags. You can use Tagulous's `register` function to do this for you - just pass it the tag field:

```
tagulous.admin.register(MyModel.tags)
```

You can also specify the tag model directly:

```
tagulous.admin.register(MyModel.tags.tag_model)
tagulous.admin.register(MyCustomTagModel)
```

If you have a custom tag model and want to extend the admin class for extra fields on your custom model, you can subclass the `TagModelAdmin` class to get the extra tag management functionality:

```
class MyModelTagsAdmin(tagulous.admin.TagModelAdmin):
    list_display = ['name', 'count', 'protected', 'my_extra_field']
admin.site.register(MyCustomTagModel, MyModelTagsAdmin)
```

When overriding options, you should base them on the options in the default `TagModelAdmin`:

```
list_display = ['name', 'count', 'protected']
list_filter = ['protected']
exclude = ['count']
actions = ['merge_tags']
```

The `TagTreeModelAdmin` also excludes the `path` field.

Remember that the relationship between your entries and tags are standard `ForeignKey` or `ManyToMany` relationships, so deletion propagation will work as it would normally.

1.10 Changelog

Tagulous follows semantic versioning in the format `BREAKING.FEATURE.BUG`:

- `BREAKING` will be marked with links to the details and upgrade instructions in *Upgrading*.
- `FEATURE` and `BUG` releases will be safe to install without reading the upgrade notes.

Changes for upcoming releases will be listed without a release date - these are available by installing the develop branch from github.

1.10.1 1.3.3, 2021-12-25

Features:

- Add Django 4.0 support

Bugfix:

- Slug uniqueness now works when there are more than 11 collisions (#152)

1.10.2 1.3.2, 2021-12-23

Changes:

- Remove tag lookup from model getstate to improve pickling performance (#143)
- Manager and QuerySet cast classes are now placed in the module of the original class so they can be imported and found by serializers and picklers
- Cast class names prefixes changed from `CastTagged` to `TagulousCastTagged` to further reduce risk of clashes
- Class casting detects and reuses classes which have already been cast

Bugfix:

- QuerySets can be pickled (#142)

1.10.3 1.3.1, 2021-12-21

Changes:

- Switch to pytest and enforce linting

Bugfix:

- Fix `_filter_or_exclude` exception missed by tests (#144, #149)

Thanks to:

- nschlemm for the ``_filter_or_exclude” fix (#144, #149)

1.10.4 1.3.0, 2021-09-07

Features:

- Add `similarly_tagged` to tagged model querysets, and `get_similar_objects` to instantiated tag fields (#115)
- New DRF serializer to serialize tags as strings (#111)
- Initial `TagField` values passed on `Form(initial=...)` can now be a string, list or tuple of strings or tags, or queryset (#107)
- Add system check for `settings.SERIALIZATION_MODULES` (#101)

Bugfix:

- Fix incorrect arguments for the TagField's `RelatedManager.set`
- Upgrade select2 to fix composed characters (#138)
- Fix select2 input where quotes in quoted tags could be escaped
- The select2 control is applied when the formset:added event adds a tag field (#97)
- Fix edge case circular import (#124)

Thanks to:

- valentijnscholten for the form `initial=` solution (#107)

1.10.5 1.2.1, 2021-08-31

Bugfix:

- Fix issue with update_or_create (#135)

1.10.6 1.2.0, 2021-08-25

Upgrade notes: *Upgrading from 1.1.0*

Features:

- Django 3.2 support
- Option `autocomplete_view_fulltext` for full text search in autocomplete view (#102)

Changes:

- Slugification now uses standard Django for unicode for consistency
- Add `autocomplete_view_args` and `autocomplete_view_kwargs` options (#119, #120)
- Documentation updates (#105, #113, #131)
- Fix division by zero issue in `weight()` (#102)

Bugfix:

- Fix issue where the Select2 adaptor for SingleTagField didn't provide an empty value, which meant it would look like it had defaulted to a value which wasn't set. (#116)
- Fix issue where the Select2 adaptor didn't correctly handle the `required` attribute, which meant browser field validation would fail silently. (#116)
- Fix dark mode support in Django admin (#125)

- Fix collapsed select2 in Django admin (#123)
- Fix duplicate migration issue (#93)
- Tagged models can now be pickled (#109)

Thanks to:

- BoPeng for the `autocomplete_view_args` config options
- valentijnscholten for the select2 doc fix
- Jens Diemer (jedie) for the readme update
- dany-nonstop for `autocomplete_view_fulltext` and weight division issue
- poolpoolpoolpool for form.media docs (#131)

1.10.7 1.1.0, 2020-12-06

Feature:

- Add Django 3.0 and 3.1 support (#85)

Changes:

- Drops support for Python 2 and 3.5
- Drops support for Django 1.11 and earlier
- Drops support for South migrations

Bugfix:

- Resolves `ManyToManyRel` issue sometimes seen in loaddata (#110)

Thanks to:

- Diego Ubirajara (dubirajara) for `FieldDoesNotExist` fix for Django 3.1
- Andrew O'Brien (marxide) for `admin.helpers` fix for Django 3.1

1.10.8 1.0.0, 2020-10-08

Upgrade notes: *Upgrading from 0.14.1*

Feature:

- Added adaptor for Select2 v4 and set as default for Django 2.2+ (#11, #12, #90)
- Support full unicode slugs with new `TAGULOUS_SLUG_ALLOW_UNICODE` setting (#22)

Changes:

- Drops support for Django 1.8 and earlier

Bugfix:

- Tag fields work with abstract and concrete inheritance (#8)
- Ensure weighted values are integers not floats (#69, #70)
- The admin site in Django 2.2+ now uses the Django vendored versions of jQuery and select2 (#76)
- Fix support for single character tags in trees (#82)
- Fix documentation for adding registering tagged models in admin (#83)

- Fix division by zero in weight() (#59, #61)
- Fix support for capitalised table name in PostgreSQL (#60, #61)
- Tag fields are stripped before parsing, preventing whitespace tags in SingleTagFields (#29)
- Fix documentation for quickstart (#41)
- Fix `prefetch_related()` on tag fields (#42)
- Correctly raise an `IntegrityError` when saving a tree tag without a name (#50)

Internal:

- Signals have been refactored to global handlers (instead of multiple independent handlers bound to descriptors)
- Code linting improved; project now uses black and isort, and flake8 pases

Thanks to:

- Khoa Pham (phamk) for `prefetch_related()` fix (#42, #87)
- Erik Van Kelst (4levels) for division by zero and capitalised table fixes (#60, #61, #62)
- hagsteel for weighted values fix (#69, #70)
- Michael Röttger (mcrot) for single character tag fix (#81, #82)
- Frank Lanitz (frlan) for admin documentation fix (#83)

1.10.9 0.14.1, 2019-09-04

Upgrade notes: *Upgrading from 0.14.0*

Feature:

- Add Django 2.2 support (closes #71)
- Upgrade example project to Django 2.2 on Python 3.7

Bugfix:

- Correct issue with multiple databases (#72)

Thanks to:

- Dmitry Ivanchenko (ivanchenkodmitry) for multiple database fix (#72)

1.10.10 0.14.0, 2019-02-24

Feature:

- Add Django 2.0 support (fixes #48, #65)
- Add Django 2.1 support (fixes #56, #58)

Bugfix:

- Fix example project (fixes #64)

Thanks to:

- Diego Ubirajara (dubirajara) for Widget.render() fix (#58)

1.10.11 0.13.2, 2018-05-28

Feature:

- Tag fields now support the argument *to_base=MyTagModelBase*

1.10.12 0.13.1, 2018-05-19

Upgrade notes: *Upgrading from 0.13.0*

Bugfix:

- `TagField(null=...)` now raises a warning about the `TagField`, rather than the parent `ManyToManyField`.

Changes:

- Reduce support for Python 3.3

1.10.13 0.13.0, 2018-04-30

Upgrade notes: *Upgrading from 0.12.0*

Feature:

- Add Django 1.11 support (fixes #28)

Changes:

- Reduce support for Django 1.4 and Python 3.2
- Remove deprecated `TagField` manager's `__len__` (#10, fixes #9)

Bugfix:

- Fix failed search in select2 v3 widget when pasting multiple tags (fixes #26)
- Fix potential race condition when creating new tags (#31)
- Temporarily disabled some migration tests which only failed under Python 2.7 with Django 1.9+ due to logic issues in the tests.
- Fix deserialization exception for model with `ManyToOneRel` (fixes #14)

Thanks to:

- Martín R. Guerrero (slackmart) for removing `__len__` method (#9, #10)
- Mark London for select2 v3 widget fix when pasting tags (#26)
- Peter Baumgartner (ipmb) for fixing race condition (#31)
- Raniere Silva (rgaics) for fixing deserialization exeption (#14, #45)

1.10.14 0.12.0, 2017-02-26

Upgrade notes: *Upgrading from 0.11.1*

Feature:

- Add Django 1.10 support (fixes #18, #20)

Bugfix:

- Remove `unique=True` from tag tree models' `path` field (fixes #1)
- Implement slug field truncation (fixes #3)
- Correct MySQL slug clash detection in tag model save
- Correct `.weight(..)` to always return floored integers instead of decimals
- Correct max length calculation when adding and removing a value through assignment
- *TagDescriptor* now has a *through* attribute to match *ManyToManyDescriptor*

Deprecates:

- *TagField* manager's *__len__* method is now deprecated and will be removed in 0.13

Thanks to:

- Pamela McA'Nulty (PamelaM) for MySQL fixes (#1)
- Mary (minidietcoke) for max count fix (#16)
- James Pic (jpic) for documentation corrections (#13)
- Robert Erb (rerb) at AASHE (http://www.aashe.org/) for Django 1.10 support (#18, #20)
- Gaël Utard (gutard) for tag descriptor *through* fix (#19)

1.10.15 0.11.1, 2015-10-05

Internal:

- Fix package configuration in setup.py

1.10.16 0.11.0, 2015-10-04

Feature:

- Add support for Python 3.2 to 3.5

Internal:

- Change `tagulous.models.initial.field_initialise_tags` and `model_initialise_tags` to take a file handle as `report`.

1.10.17 0.10.0, 2015-09-28

Upgrade notes: *Upgrading from 0.9.0*

Feature:

- Add fields `level` and `label` to *tagulous.models.TagTreeModel* (were properties)
- Add `TagTreeModel.get_siblings()`
- Add *tagulous.models.TagTreeModelQuerySet* methods `with_ancestors()`, `with_descendants()` and `with_siblings()`
- Add *space_delimiter* tag option to disable space as a delimiter
- Tagulous available from pypi as `django-tagulous`
- *TagModel.merge_tags* can now accept a tag string
- *TagTreeModel.merge_tags* can now merge recursively with new argument `children=True`
- Support for recursively merging tree tags in admin site

Internal:

- Add support for Django 1.9a1
- `TagTreeModel.tag_options.tree` will now always be `True`
- JavaScript `parseTags` arguments have changed
- Added example project to github repository

Bugfix:

- `TagRelatedManager` instances can be compared to each other
- Admin inlines now correctly suppress popup buttons
- Select2 adaptor correctly parses ajax response
- *TagMeta* raises an exception if *tree* is set
- Default help text no longer changes for *tagulous.models.SingleTagField*

1.10.18 0.9.0, 2015-09-14

Upgrade notes: *Upgrading from 0.8.0*

Internal:

- Add support for Django 1.7 and 1.8

Removed:

- `tagulous.admin.tag_model` has been removed

Bugfix:

- Using a tag field with a non-tag model raises exception

1.10.19 0.8.0, 2015-08-22

Upgrade notes: *Upgrading from 0.7.0 or earlier*

Feature:

- Tag cloud support
- Improved admin.register
- Added tag-aware serializers

Deprecated:

- `tagulous.admin.tag_model` will be removed in the next version

Bugfix:

- Setting tag options twice raises exception
- Tagged inline formsets work correctly

Internal:

- South migration support improved
- Tests moved to top level, tox support added
- Many small code improvements and bug fixes

1.10.20 0.7.0, 2015-07-01

Feature:

- Added tree support

1.10.21 0.6.0, 2015-05-11

Feature:

- Initial public preview

1.11 Upgrading

This document details breaking changes between versions, with any necessary steps to safely upgrade.

For an overview of what has changed between versions, see the *Changelog*.

1.11.1 Instructions

Tagulous follows semantic versioning in the format `BREAKING.FEATURE.BUG`:

- Read the upgrade notes for a `BREAKING` release to see if you need to take further action when upgrading.
- `FEATURE` and `BUG` releases will be safe to install without reading the upgrade notes.

1. Check which version of Tagulous you are upgrading from:

```
python
>>> import tagulous
>>> tagulous.__version__
```

2. Upgrade the Tagulous package:

```
pip install --upgrade django-tagulous
```

3. Scroll down to the earliest instructions relevant to your version, and follow them up to the latest version.

Upgrading from 1.1.0

Slugify behaviour

In Tagulous 1.2.0 the slugify logic has been replaced with Django's now all supported Django versions support the `allow_unicode` slugify option.

If unicode tag slugs are not enabled with `TAGULOUS_SLUG_ALLOW_UNICODE` *setting*, Django's implementation of unicode to ASCII does not support logographic characters, so these will be stripped as per Django's standard `slugify()` output, rather than Tagulous' old behaviour of replacing them with underscore characters. This can now lead to empty slugs, which will now default to a single underscore.

As a result of this change, the optional dependency `unidecode` and its corresponding extra installation requirements `[i18n]` have been removed.

Upgrading from 0.14.1

Django and Python support

Tagulous 0.14.1 was the last version to support Django 1.10 and earlier. Tagulous 1.0.0 requires Django 1.11 or later, and Python 2.7 or 3.5 or later.

Autocomplete upgrade

Tagulous 1.0.0 changes the default JavaScript adaptor to use select2 v4. This may necessitate some styling changes on your user-facing pages if you have overridden the default styles.

Single tag behaviour

Tagulous 1.0.0 no longer allows whitespace tags in `SingleTagField`.

Upgrading from 0.14.0

Tagulous 0.14.0 was the last version to officially support Django 1.10 or earlier.

Upgrading from 0.13.0

1. Setting `null` in a model `TagField` has raised a warning in the parent `ManyToManyField` since Django 1.9. The warning now correctly blames a `TagField` instead. The `null` argument in a model `TagField` is deprecated and has no effect, so should not be used.
2. Version 0.13.1 reduces support for Python 3.3. No known breaking changes have been introduced, but this version of Python will no longer be tested against due to lack of support in third party tools.

Upgrading from 0.12.0

1. Auto-generated tag models have been renamed.

   Django 1.11 introduced a rule that models cannot start with an underscore. Prior to this, Tagulous auto-generated tag models started `_Tagulous_`, to indicate they are auto-generated. From now on, they will start `Tagulous_`.

   Django migrations should detect this model name change:

   ```
   ./manage.py makemigrations
   Did you rename the myapp._Tagulous_MyModel model to Tagulous_MyModel? [y/N]
   ```

   Answer *y* for all Tagulous auto-generated models, and migrate:

   ```
   ./manage.py migrate
   ```

   Troubleshooting:

   - If you do not see the rename prompt when running `makemigrations`, you will need to edit the migration manually - see *RenameModel <https://docs.djangoproject.com/en/1.11/ref/migration-operations/#renamemodel>* in the Django documentation for more details.
   - If any `AlterField` changes to the `SingleTagField` or `TagField` definitions are included in this set of migrations, the new tag model's name will not be correctly detected and you will see the errors `Related model ... cannot be resolved` or `AttributeError: 'TagField' object has no attribute 'm2m_reverse_field_name'`. To resolve these, manually change the `to` parameter in your `AlterField`'s tag field definition from `myapp._Tagulous_...` to `myapp.Tagulous_...`.
   - If you see an error `Renaming the table while in a transaction is not supported because it would break referential integrity`, add `atomic = False` to your migration class.

2. Version 0.13.0 reduces support for Django 1.4 and Python 3.2. No known breaking changes have been introduced, but these versions of Django and Python will no longer be tested against due to lack of support in third party tools.
3. The `TagField` manager's `__len__` has now been removed, following its deprecation in 0.12.0. If you are currently using `len(instance.tagfield)` then you should switch to using `instance.tagfield.count()` instead (see *upgrade instructions*).

Upgrading from 0.11.1

1. Starting with version 0.12.0, Tagulous no longer enforces uniqueness for tree `path` fields. This means that Django will detect a change to your models, and warn you that your migrations are out of sync. It is safe for you to create and apply a standard migration with:

   ```
   ./manage.py makemigrations
   ./manage.py migrate
   ```

   This change is to avoid MySQL (and possibly other databases) from the errror `"BLOB/TEXT column 'path' used in key specification without a key length"` - see https://github.com/radiac/django-tagulous/issues/1 for discussion.

2. Version 0.12.0 deprecates the model tag manager's *__len__* method in preparation for merging https://github.com/radiac/django-tagulous/pull/10 to resolve https://github.com/radiac/django-tagulous/issues/9.

   If you are currently using *len(instance.tagfield)* then you should switch to using *instance.tagfield.count()* instead.

Upgrading from 0.9.0

1. Starting with version 0.10.0, Tagulous is available on pypi. You can continue to run the development version direct from github, but if you would prefer to use stable releases you can reinstall:

   ```
   pip uninstall django-tagulous
   pip install django-tagulous
   ```

2. Version 0.10.0 adds `label` and `level` fields to the `TagTreeModel` base class (they were previously properties). This means that each of your tag tree models will need a schema and data migration.

   The schema migration will require a default value for the label; enter any valid string, eg `'.'`

   The data migration will need to call `mytagtreemodel.objects.rebuild()` to set the correct values for `label` and `level`.

   You will need to create and apply these migrations to each of your tag tree models

   Django migrations:

   ```
   python manage.py makemigrations myapp
   python manage.py migrate myapp
   python manage.py makemigrations myapp --empty
   # Add data migration operation below
   python manage.py migrate myapp
   ```

   Your Django data migration should include:

   ```
   def rebuild_tree(apps, schema_editor):
       # For an auto-generated tag tree model:
       model = apps.get_model('myapp', '_Tagulous_MyModel_tagtreefield')

       # For a custom tag tree model:
       #model = apps.get_model('myapp', 'MyTagTreeModel')

       model.objects.rebuild()

   class Migration(migrations.Migration):
   ```

(continues on next page)

(continued from previous page)

```
    # ... rest of Migration as generated
    operations = [
        migrations.RunPython(rebuild_tree)
    ]
```

South migrations:

```
python manage.py schemamigration --auto myapp
python manage.py migrate myapp
python manage.py datamigration myapp upgrade_trees
# Add data migration function below
python manage.py migrate myapp
```

Your South data migration function should be:

```
def forwards(self, orm):
    # For an auto-generated tag tree model:
    model = orm['myapp._Tagulous_MyModel_tagtreefield'].objects.rebuild()

    # For a custom tag tree model:
    #model = orm['myapp.MyTagTreeModel'].objects.rebuild()
```

3. Since version 0.10.0 *tree* cannot be set in *TagMeta*; custom tag models must get their tree status from their base class.
4. In version 0.10.0, `TagOptions.field_items` was renamed to `TagOptions.form_items`, and `constants.FIELD_OPTIONS` was renamed to `constants.FORM_OPTIONS`. These were internal, so should not affect your code.
5. The tag parsers now accept a new argument to control whether space is used as a delimiter or not. These are internal, so should not affect your code, unless you have written a custom adaptor.

Upgrading from 0.8.0

1. Since 0.9.0, `SingleTagField` and `TagField` raise an exception if the tag model isn't a subclass of TagModel.
2. The documentation for `tagulous.models.migrations.add_unique_column` has been clarified to illustrate the risk of using it with a non-transactional database. If you use this in your migrations, read the documentation to be sure you understand the problem involved.

Upgrading from 0.7.0 or earlier

1. `tagulous.admin.tag_model` was deprecated in 0.8.0 and removed in 0.9.0; use `tagulous.admin.register` instead:

```
tagulous.admin.tag_model(MyModel.tags)
tagulous.admin.tag_model(MyModel.tags, my_admin_site)
# becomes:
tagulous.admin.register(MyModel.tags)
tagulous.admin.register(MyModel.tags, site=my_admin_site)
```

2. Since 0.8.0, a `ValueError` exception is raised if a tag model field definition specifies both a tag model and tag options.

For custom tag models, tag options must be set by adding a `class TagMeta` to your model. You can no longer set tag options in the tag field.

Where an auto-generated tag model is shared with another tag field, the first tag field must set all tag options.

3. Any existing South migrations with `SingleTagField` or `TagField` definitions which automatically generate their tag models will need to be manually modified in the `Migration.models` definition to have the attribute `'_set_tag_meta': 'True'`. For example, the line:

```
'labels': ('tagulous.models.fields.TagField', [], {'force_lowercase': 'True', 'to':␣
↪u"orm['myapp._Tagulous_MyModel_labels']", 'blank': 'True'}),
```

becomes:

```
'labels': ('tagulous.models.fields.TagField', [], {'force_lowercase': 'True', 'to':␣
↪u"orm['myapp._Tagulous_MyModel_labels']", 'blank': 'True', '_set_tag_meta': 'True
↪'}),
```

Any *db.add_column* calls will need to be changed too:

```
db.add_column(u'myapp_mymodel', 'singletag',
              self.gf('tagulous.models.fields.SingleTagField')(null=True, ...),
              ...)
```

becomes:

```
db.add_column(u'myapp_mymodel', 'singletag',
              self.gf('tagulous.models.fields.SingleTagField')(_set_tag_meta=True,␣
↪null=True, ...),
              ...)
```

This will use the keyword tag options to update the tag model's objects, rather than raising the new `ValueError`.

1.12 Contributing

Contributions are welcome, preferably via pull request. Check the github issues to see what needs work. Tagulous aims to be a comprehensive tagging solution, but try to keep new features from having a significant impact on people who won't use them (eg tree support is optional).

When submitting UI changes, please aim to support the latest versions of Chrome, Firefox and Internet Explorer through progressive enhancement - users of old browsers must still be able to tag things, even if they don't get all the bells and whistles.

1.12.1 Installing

The easiest way to work on Tagulous is to fork the project on github, then install it to a virtualenv:

```
virtualenv django-tagulous
cd django-tagulous
source bin/activate
pip install -e git+git@github.com:USERNAME/django-tagulous.git#egg=django-tagulous
pip install -r src/django-tagulous/requirements.test.txt
```

(replacing USERNAME with your username).

This will install the development dependencies too, and you'll find the tagulous source ready for you to work on in the src folder of your virtualenv.

1.12.2 Testing

It is greatly appreciated when contributions come with unit tests.

Pytest is the test runner of choice:

```
pytest
pytest tests/test_file.py
pytest tests/test_file::TestClass::test_method
```

Use tox to run them on one or more supported versions:

```
tox [-e py39-django3.2]
```

To use a different database (mysql, postgres etc) use the environment variables DATABASE_ENGINE, DATABASE_NAME, DATABASE_USER, DATABASE_PASSWORD, DATABASE_HOST and DATABASE_PORT, eg:

```
DATABASE_ENGINE=pgsql DATABASE_NAME=tagulous_test [...] tox
```

Most Tagulous python modules have corresponding test modules, with test classes which subclass tests.lib.TagTestManager. They use test apps defined under the tests dir where required.

Run the javascript tests using Jasmine:

```
pip install jasmine
cd tests
jasmine
# open http://127.0.0.1:8888/ in your browser
```

Javascript tests are defined in tests/spec/javascripts/*.spec.js.

1.12.3 Code overview

Tag model fields start in tagulous/models/fields.py; when they are added to models, the models call the field's contribute_to_class method, which adds the descriptors in tagulous/models/descriptors.py onto the model in their place. These descriptors act as getters and setters, channeling data to and from the managers in tagulous/models/managers.py.

Models which have tag fields are called tagged models. For tags to be fully supported in constructors, managers and querysets, those classes need to use the classes defined in tagulous/models/tagged.py as base classes. That file contains a class_prepared signal listener which tries to dynamically change the base classes of any models which contain tag fields.

Model fields take their arguments and store them in a TagOptions instance, defined in tagulous/models/options.py. Any initial tags in the options can be loaded into the database using the functions in tagulous/models/initial.py, which is the same code the initial_tags management command uses.

When a ModelForm is created for a model with a tag field, the model field's formfield method is called. This creates a tag form field, defined in tagulous/forms.py, which is passed the TagOptions from the model. A tag form field can also be created directly on a plain form. Tag form fields in turn uses tag widgets (also in tagulous/forms.py) to render the field to HTML with the data from TagOptions.

Tag strings are parsed and rendered (tags joined back to a tag string) by the functions in tagulous/utils.py.

Everything for enhancing the admin site with support for tag fields is in tagulous/admin.py. It is in two sections; registration (which adds tag field functionality to a normal `ModelAdmin`, and replaces the widgets with tag widgets) and tag model admin (for managing tag models).