django-model-utils Documentation

Release 2.1.1

Carl Meyer

January 12, 2017

Contents

Django model mixins and utilities.

CHAPTER 1

Contents

1.1 Setup

1.1.1 Installation

Install from PyPI with `pip`:

```
pip install django-model-utils
```

To use `django-model-utils` in your Django project, just import and use the utility classes described in this documentation; there is no need to modify your `INSTALLED_APPS` setting.

1.1.2 Dependencies

`django-model-utils` supports Django 1.4.2 and later on Python 2.6, 2.7, 3.2, and 3.3.

1.2 Fields

1.2.1 StatusField

A simple convenience for giving a model a set of "states." `StatusField` is a `CharField` subclass that expects to find a class attribute called `STATUS` on its model or you can pass `choices_name` to use a different attribute name, and uses that as its `choices`. Also sets a default `max_length` of 100, and sets its default value to the first item in the `STATUS` choices:

```
from model_utils.fields import StatusField
from model_utils import Choices

class Article(models.Model):
    STATUS = Choices('draft', 'published')
    # ...
    status = StatusField()
```

(The `STATUS` class attribute does not have to be a *Choices* instance, it can be an ordinary list of two-tuples).

Using a different name for the model's choices class attribute

```
from model_utils.fields import StatusField
from model_utils import Choices

class Article(models.Model):
    ANOTHER_CHOICES = Choices('draft', 'published')
    # ...
    another_field = StatusField(choices_name='ANOTHER_CHOICES')
```

`StatusField` does not set `db_index=True` automatically; if you expect to frequently filter on your status field (and it will have enough selectivity to make an index worthwhile) you may want to add this yourself.

1.2.2 MonitorField

A `DateTimeField` subclass that monitors another field on the model, and updates itself to the current date-time whenever the monitored field changes:

```
from model_utils.fields import MonitorField, StatusField

class Article(models.Model):
    STATUS = Choices('draft', 'published')

    status = StatusField()
    status_changed = MonitorField(monitor='status')
```

(A `MonitorField` can monitor any type of field for changes, not only a `StatusField`.)

If a list is passed to the `when` parameter, the field will only update when it matches one of the specified values:

```
from model_utils.fields import MonitorField, StatusField

class Article(models.Model):
    STATUS = Choices('draft', 'published')

    status = StatusField()
    published_at = MonitorField(monitor='status', when=['published'])
```

1.2.3 SplitField

A `TextField` subclass that automatically pulls an excerpt out of its content (based on a "split here" marker or a default number of initial paragraphs) and stores both its content and excerpt values in the database.

A `SplitField` is easy to add to any model definition:

```
from django.db import models
from model_utils.fields import SplitField

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = SplitField()
```

`SplitField` automatically creates an extra non-editable field `_body_excerpt` to store the excerpt. This field doesn't need to be accessed directly; see below.

Accessing a SplitField on a model

When accessing an attribute of a model that was declared as a `SplitField`, a `SplitText` object is returned. The `SplitText` object has three attributes:

`content`: The full field contents.

`excerpt`: The excerpt of `content` (read-only).

`has_more`: True if the excerpt and content are different, False otherwise.

This object also has a `__unicode__` method that returns the full content, allowing `SplitField` attributes to appear in templates without having to access `content` directly.

Assuming the `Article` model above:

```
>>> a = Article.objects.all()[0]
>>> a.body.content
u'some text\n\n<!-- split -->\n\nmore text'
>>> a.body.excerpt
u'some text\n'
>>> unicode(a.body)
u'some text\n\n<!-- split -->\n\nmore text'
```

Assignment to `a.body` is equivalent to assignment to `a.body.content`.

Note: a.body.excerpt is only updated when a.save() is called

Customized excerpting

By default, `SplitField` looks for the marker `<!-- split -->` alone on a line and takes everything before that marker as the excerpt. This marker can be customized by setting the `SPLIT_MARKER` setting.

If no marker is found in the content, the first two paragraphs (where paragraphs are blocks of text separated by a blank line) are taken to be the excerpt. This number can be customized by setting the `SPLIT_DEFAULT_PARAGRAPHS` setting.

1.3 Models

1.3.1 TimeFramedModel

An abstract base class for any model that expresses a time-range. Adds `start` and `end` nullable DateTimeFields, and a `timeframed` manager that returns only objects for whom the current date-time lies within their time range.

1.3.2 TimeStampedModel

This abstract base class just provides self-updating `created` and `modified` fields on any model that inherits from it.

1.3.3 StatusModel

Pulls together *StatusField*, *MonitorField* and *QueryManager* into an abstract base class for any model with a "status."

Just provide a `STATUS` class-attribute (a *Choices* object or a list of two-tuples), and your model will have a `status` field with those choices, a `status_changed` field containing the date-time the `status` was last changed, and a manager for each status that returns objects with that status only:

```
from model_utils.models import StatusModel
from model_utils import Choices

class Article(StatusModel):
    STATUS = Choices('draft', 'published')

# ...

a = Article()
a.status = Article.STATUS.published

# this save will update a.status_changed
a.save()

# this query will only return published articles:
Article.published.all()
```

1.4 Model Managers

1.4.1 InheritanceManager

This manager (contributed by Jeff Elmore) should be attached to a base model class in a model-inheritance tree. It allows queries on that base model to return heterogenous results of the actual proper subtypes, without any additional queries.

For instance, if you have a `Place` model with subclasses `Restaurant` and `Bar`, you may want to query all Places:

```
nearby_places = Place.objects.filter(location='here')
```

But when you iterate over `nearby_places`, you'll get only `Place` instances back, even for objects that are "really" `Restaurant` or `Bar`. If you attach an `InheritanceManager` to `Place`, you can just call the `select_subclasses()` method on the `InheritanceManager` or any `QuerySet` from it, and the resulting objects will be instances of `Restaurant` or `Bar`:

```
from model_utils.managers import InheritanceManager

class Place(models.Model):
    # ...
    objects = InheritanceManager()

class Restaurant(Place):
    # ...

class Bar(Place):
    # ...

nearby_places = Place.objects.filter(location='here').select_subclasses()
```

```
for place in nearby_places:
    # "place" will automatically be an instance of Place, Restaurant, or Bar
```

The database query performed will have an extra join for each subclass; if you want to reduce the number of joins and you only need particular subclasses to be returned as their actual type, you can pass subclass names to `select_subclasses()`, much like the built-in `select_related()` method:

```
nearby_places = Place.objects.select_subclasses("restaurant")
# restaurants will be Restaurant instances, bars will still be Place instances

nearby_places = Place.objects.select_subclasses("restaurant", "bar")
# all Places will be converted to Restaurant and Bar instances.
```

It is also possible to use the subclasses themselves as arguments to `select_subclasses`, leaving it to calculate the relationship for you:

```
nearby_places = Place.objects.select_subclasses(Restaurant)
# restaurants will be Restaurant instances, bars will still be Place instances

nearby_places = Place.objects.select_subclasses(Restaurant, Bar)
# all Places will be converted to Restaurant and Bar instances.
```

It is even possible to mix and match the two:

```
nearby_places = Place.objects.select_subclasses(Restaurant, "bar")
# all Places will be converted to Restaurant and Bar instances.
```

`InheritanceManager` also provides a subclass-fetching alternative to the `get()` method:

```
place = Place.objects.get_subclass(id=some_id)
# "place" will automatically be an instance of Place, Restaurant, or Bar
```

If you don't explicitly call `select_subclasses()` or `get_subclass()`, an `InheritanceManager` behaves identically to a normal `Manager`; so it's safe to use as your default manager for the model.

Note: Due to Django bug #16572, on Django versions prior to 1.6 `InheritanceManager` only supports a single level of model inheritance; it won't work for grandchild models.

1.4.2 QueryManager

Many custom model managers do nothing more than return a QuerySet that is filtered in some way. `QueryManager` allows you to express this pattern with a minimum of boilerplate:

```
from django.db import models
from model_utils.managers import QueryManager

class Post(models.Model):
    ...
    published = models.BooleanField()
    pub_date = models.DateField()
    ...

    objects = models.Manager()
    public = QueryManager(published=True).order_by('-pub_date')
```

The kwargs passed to `QueryManager` will be passed as-is to the `QuerySet.filter()` method. You can also pass a `Q` object to `QueryManager` to express more complex conditions. Note that you can set the ordering of the

`QuerySet` returned by the `QueryManager` by chaining a call to `.order_by()` on the `QueryManager` (this is not required).

1.4.3 PassThroughManager

A common "gotcha" when defining methods on a custom manager class is that those same methods are not automatically also available on the QuerySets returned by that manager, so are not "chainable". This can be counterintuitive, as most of the public QuerySet API is mirrored on managers. It is possible to create a custom Manager that returns QuerySets that have the same additional methods, but this requires boilerplate code. The `PassThroughManager` class (contributed by Paul McLanahan) removes this boilerplate.

To use `PassThroughManager`, rather than defining a custom manager with additional methods, define a custom `QuerySet` subclass with the additional methods you want, and pass that `QuerySet` subclass to the `PassThroughManager.for_queryset_class()` class method. The returned `PassThroughManager` subclass will always return instances of your custom `QuerySet`, and you can also call methods of your custom `QuerySet` directly on the manager:

```
from datetime import datetime
from django.db import models
from django.db.models.query import QuerySet
from model_utils.managers import PassThroughManager

class PostQuerySet(QuerySet):
    def by_author(self, user):
        return self.filter(user=user)

    def published(self):
        return self.filter(published__lte=datetime.now())

    def unpublished(self):
        return self.filter(published__gte=datetime.now())


class Post(models.Model):
    user = models.ForeignKey(User)
    published = models.DateTimeField()

    objects = PassThroughManager.for_queryset_class(PostQuerySet)()

Post.objects.published()
Post.objects.by_author(user=request.user).unpublished()
```

1.4.4 Mixins

Each of the above manager classes has a corresponding mixin that can be used to add functionality to any manager. For example, to create a GeoDjango `GeoManager` that includes "pass through" functionality, you can write the following code:

```
from django.contrib.gis.db import models
from django.contrib.gis.db.models.query import GeoQuerySet

from model_utils.managers import PassThroughManagerMixin

class PassThroughGeoManager(PassThroughManagerMixin, models.GeoManager):
    pass
```

```
class LocationQuerySet(GeoQuerySet):
    def within_boundary(self, geom):
        return self.filter(point__within=geom)

    def public(self):
        return self.filter(public=True)

class Location(models.Model):
    point  = models.PointField()
    public = models.BooleanField(default=True)
    objects = PassThroughGeoManager.for_queryset_class(LocationQuerySet)()

Location.objects.public()
Location.objects.within_boundary(geom=geom)
Location.objects.within_boundary(geom=geom).public()
```

Now you have a "pass through manager" that can also take advantage of GeoDjango's spatial lookups. You can similarly add additional functionality to any manager by composing that manager with `InheritanceManagerMixin` or `QueryManagerMixin`.

(Note that any manager class using `InheritanceManagerMixin` must return a `QuerySet` class using `InheritanceQuerySetMixin` from its `get_queryset` method. This means that if composing `InheritanceManagerMixin` and `PassThroughManagerMixin`, the `QuerySet` class passed to `PassThroughManager.for_queryset_class` must inherit `InheritanceQuerySetMixin`.)

1.5 Miscellaneous Utilities

1.5.1 Choices

`Choices` provides some conveniences for setting `choices` on a Django model field:

```
from model_utils import Choices

class Article(models.Model):
    STATUS = Choices('draft', 'published')
    status = models.CharField(choices=STATUS, default=STATUS.draft, max_length=20)
```

A `Choices` object is initialized with any number of choices. In the simplest case, each choice is a string; that string will be used both as the database representation of the choice, and the human-readable representation. Note that you can access options as attributes on the `Choices` object: `STATUS.draft`.

But you may want your human-readable versions translated, in which case you need to separate the human-readable version from the DB representation. In this case you can provide choices as two-tuples:

```
from model_utils import Choices

class Article(models.Model):
    STATUS = Choices(('draft', _('draft')), ('published', _('published')))
    status = models.CharField(choices=STATUS, default=STATUS.draft, max_length=20)
```

But what if your database representation of choices is constrained in a way that would hinder readability of your code? For instance, you may need to use an `IntegerField` rather than a `CharField`, or you may want the database to order the values in your field in some specific way. In this case, you can provide your choices as triples, where the first element is the database representation, the second is a valid Python identifier you will use in your code as a constant, and the third is the human-readable version:

```
from model_utils import Choices

class Article(models.Model):
    STATUS = Choices((0, 'draft', _('draft')), (1, 'published', _('published')))
    status = models.IntegerField(choices=STATUS, default=STATUS.draft)
```

You can index into a `Choices` instance to translate a database representation to its display name:

```
status_display = Article.STATUS[article.status]
```

Option groups can also be used with `Choices`; in that case each argument is a tuple consisting of the option group name and a list of options, where each option in the list is either a string, a two-tuple, or a triple as outlined above. For example:

```
from model_utils import Choices

class Article(models.Model):
STATUS = Choices(('Visible', ['new', 'archived']), ('Invisible', ['draft', 'deleted']))
```

Choices can be concatenated with the + operator, both to other Choices instances and other iterable objects that could be converted into Choices:

```
from model_utils import Choices

GENERIC_CHOICES = Choices((0, 'draft', _('draft')), (1, 'published', _('published')))

class Article(models.Model):
    STATUS = GENERIC_CHOICES + [(2, 'featured', _('featured'))]
    status = models.IntegerField(choices=STATUS, default=STATUS.draft)
```

1.5.2 Field Tracker

A `FieldTracker` can be added to a model to track changes in model fields. A `FieldTracker` allows querying for field changes since a model instance was last saved. An example of applying `FieldTracker` to a model:

```
from django.db import models
from model_utils import FieldTracker

class Post(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()

    tracker = FieldTracker()
```

Note: `django-model-utils` 1.3.0 introduced the `ModelTracker` object for tracking changes to model field values. Unfortunately `ModelTracker` suffered from some serious flaws in its handling of `ForeignKey` fields, potentially resulting in many extra database queries if a `ForeignKey` field was tracked. In order to avoid breaking API backwards-compatibility, `ModelTracker` retains the previous behavior but is deprecated, and `FieldTracker` has been introduced to provide better `ForeignKey` handling. All uses of `ModelTracker` should be replaced by `FieldTracker`.

Summary of differences between `ModelTracker` and `FieldTracker`:

- The previous value returned for a tracked `ForeignKey` field will now be the raw ID rather than the full object (avoiding extra database queries). (GH-43)

- The `changed()` method no longer returns the empty dictionary for all unsaved instances; rather, `None` is considered to be the initial value of all fields if the model has never been saved, thus `changed()` on an unsaved instance will return a dictionary containing all fields whose current value is not `None`.
- The `has_changed()` method no longer crashes after an object's first save. (GH-53).

Accessing a field tracker

There are multiple methods available for checking for changes in model fields.

previous

Returns the value of the given field during the last save:

```
>>> a = Post.objects.create(title='First Post')
>>> a.title = 'Welcome'
>>> a.tracker.previous('title')
u'First Post'
```

Returns `None` when the model instance isn't saved yet.

has_changed

Returns `True` if the given field has changed since the last save:

```
>>> a = Post.objects.create(title='First Post')
>>> a.title = 'Welcome'
>>> a.tracker.has_changed('title')
True
>>> a.tracker.has_changed('body')
False
```

The `has_changed` method relies on `previous` to determine whether a field's values has changed.

changed

Returns a dictionary of all fields that have been changed since the last save and the values of the fields during the last save:

```
>>> a = Post.objects.create(title='First Post')
>>> a.title = 'Welcome'
>>> a.body = 'First post!'
>>> a.tracker.changed()
{'title': 'First Post', 'body': ''}
```

The `changed` method relies on `has_changed` to determine which fields have changed.

Tracking specific fields

A fields parameter can be given to `FieldTracker` to limit tracking to specific fields:

```
from django.db import models
from model_utils import FieldTracker

class Post(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()

    title_tracker = FieldTracker(fields=['title'])
```

An example using the model specified above:

```
>>> a = Post.objects.create(title='First Post')
>>> a.body = 'First post!'
>>> a.title_tracker.changed()
{'title': None}
```

Checking changes using signals

The field tracker methods may also be used in `pre_save` and `post_save` signal handlers to identify field changes on model save.

Note: Due to the implementation of `FieldTracker`, `post_save` signal handlers relying on field tracker methods should only be registered after model creation.

CHAPTER 2

Contributing

Please file bugs and send pull requests to the GitHub repository and issue tracker.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search