django-model-utils Documentation

Release 4.1.1

Carl Meyer

Apr 08, 2021

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 2.1+ and 3.0+ (latest bugfix release in each series only) on Python 3.6, 3.7 and 3.8.

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

```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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.

1.2.4 UUIDField

A `UUIDField` subclass that provides an UUID field. You can add this field to any model definition.

With the param `primary_key` you can set if this field is the primary key for the model, default is True.

Param `version` is an integer that set default UUID version. Versions 1,3,4 and 5 are supported, default is 4.

If `editable` is set to false the field will not be displayed in the admin or any other ModelForm, default is False.

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

class MyAppModel(models.Model):
    uuid = UUIDField(primary_key=True, version=4, editable=False)
```

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 provides a new `timeframed` manager on the subclass whose queryset pre-filters results to only include those which have a `start` which is not in the future, and an `end` which is not in the past. If either `start` or `end` is `null`, the manager will include it.

```
from model_utils.models import TimeFramedModel
from datetime import datetime, timedelta
class Post(TimeFramedModel):
    pass

p = Post()
p.start = datetime.utcnow() - timedelta(days=1)
p.end = datetime.utcnow() + timedelta(days=7)
p.save()

# this query will return the above Post instance:
Post.timeframed.all()

p.start = None
p.end = None
p.save()

# this query will also return the above Post instance, because
# the `start` and/or `end` are NULL.
Post.timeframed.all()

p.start = datetime.utcnow() + timedelta(days=7)
p.save()

# this query will NOT return our Post instance, because
# the start date is in the future.
Post.timeframed.all()
```

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()
```

1.3.4 SoftDeletableModel

This abstract base class just provides a field `is_removed` which is set to True instead of removing the instance. Entities returned in manager `available_objects` are limited to not-deleted instances.

Note that relying on the default `objects` manager to filter out not-deleted instances is deprecated. `objects` will include deleted objects in a future release.

1.3.5 UUIDModel

This abstract base class provides `id` field on any model that inherits from it which will be the primary key.

If you dont want to set `id` as primary key or change the field name, you can be override it with our UUIDField

Also you can override the default uuid version. Versions 1,3,4 and 5 are now supported.

```
from model_utils.models import UUIDModel

class MyAppModel(UUIDModel):
    pass
```

1.3.6 SaveSignalHandlingModel

An abstract base class model to pass a parameter `signals_to_disable` to `save` method in order to disable signals

```
from model_utils.models import SaveSignalHandlingModel

class SaveSignalTestModel(SaveSignalHandlingModel):
    name = models.CharField(max_length=20)

obj = SaveSignalTestModel(name='Test')
# Note: If you use `Model.objects.create`, the signals can't be disabled
obj.save(signals_to_disable=['pre_save'] # disable `pre_save` signal
```

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.

1.4.2 JoinManager

The `JoinManager` will create a temporary table of your current queryset and join that temporary table with the model of your current queryset. This can be advantageous if you have to page through your entire DB and using django's slice mechanism to do that. `LIMIT .. OFFSET ..` becomes slower the bigger offset you use.

```
sliced_qs = Place.objects.all()[2000:2010]
qs = sliced_qs.join()
# qs contains 10 objects, and there will be a much smaller performance hit
# for paging through all of first 2000 objects.
```

Alternatively, you can give it a queryset and the manager will create a temporary table and join that to your current queryset. This can work as a more performant alternative to using django's `__in` as described in the following (StackExchange answer).

```
big_qs = Restaurant.objects.filter(menu='vegetarian')
qs = Country.objects.filter(country_code='SE').join(big_qs)
```

1.4.3 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).

1.4.4 SoftDeletableManager

Returns only model instances that have the `is_removed` field set to False. Uses `SoftDeletableQuerySet`, which ensures model instances won't be removed in bulk, but they will be marked as removed instead.

1.4.5 Mixins

Each of the above manager classes has a corresponding mixin that can be used to add functionality to any manager.

Note that any manager class using `InheritanceManagerMixin` must return a `QuerySet` class using `InheritanceQuerySetMixin` from its `get_queryset` method.

1.5 Miscellaneous Utilities

1.5.1 Choices

Note: Django 3.0 adds enumeration types. These provide most of the same features as `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)
```

Should you wish to provide a subset of choices for a field, for instance, you have a form class to set some model instance to a failed state, and only wish to show the user the failed outcomes from which to select, you can use the `subset` method:

```
from model_utils import Choices

OUTCOMES = Choices(
    (0, 'success', _('Successful')),
    (1, 'user_cancelled', _('Cancelled by the user')),
    (2, 'admin_cancelled', _('Cancelled by an admin')),
)
FAILED_OUTCOMES = OUTCOMES.subset('user_cancelled', 'admin_cancelled')
```

The `choices` attribute on the model field can then be set to `FAILED_OUTCOMES`, thus allowing the subset to be defined in close proximity to the definition of all the choices, and reused elsewhere as required.

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.

If a field is deferred, calling `previous()` will load the previous value from the database.

has_changed

Returns `True` if the given field has changed since the last save. The `has_changed` method expects a single field:

```
>>> 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.

If a field is deferred and has been assigned locally, calling `has_changed()` will load the previous value from the database to perform the comparison.

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}
```

Tracking Foreign Key Fields

It should be noted that a generic FieldTracker tracks Foreign Keys by db_column name, rather than model field name, and would be accessed as follows:

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

class Parent(models.Model):
    name = models.CharField(max_length=64)

class Child(models.Model):
    name = models.CharField(max_length=64)
    parent = models.ForeignKey(Parent)
    tracker = FieldTracker()
```

```
>>> p = Parent.objects.create(name='P')
>>> c = Child.objects.create(name='C', parent=p)
>>> c.tracker.has_changed('parent_id')
```

To find the db_column names of your model (using the above example):

```
>>> for field in Child._meta.fields:
        field.get_attname_column()
('id', 'id')
('name', 'name')
('parent_id', 'parent_id')
```

The model field name *may* be used when tracking with a specific tracker:

```
specific_tracker = FieldTracker(fields=['parent'])
```

But according to issue #195 this is not recommended for accessing Foreign Key Fields.

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.

FieldTracker implementation details

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

class MyModel(TimeStampedModel):
    name = models.CharField(max_length=64)
    tracker = FieldTracker()

    def save(self, *args, **kwargs):
        """ Automatically add "modified" to update_fields."""
        update_fields = kwargs.get('update_fields')
        if update_fields is not None:
            kwargs['update_fields'] = set(update_fields) | {'modified'}
        super().save(*args, **kwargs)

# [...]

instance = MyModel.objects.first()
instance.name = 'new'
instance.save(update_fields={'name'})
```

This is how `FieldTracker` tracks field changes on `instance.save` call.

1. In `class_prepared` handler `FieldTracker` patches `save_base` and `refresh_from_db` methods to reset initial state for tracked fields.
2. In `post_init` handler `FieldTracker` saves initial values for tracked fields.
3. `MyModel.save` changes `update_fields` in order to store auto updated `modified` timestamp. Complete list of saved fields is now known.
4. `Model.save` does nothing interesting except calling `save_base`.
5. Decorated `save_base()` method calls `super().save_base` and all fields that have values different to initial are considered as changed.
6. `Model.save_base` sends `pre_save` signal, saves instance to database and sends `post_save` signal. All `pre_save`/`post_save` receivers can query `instance.tracker` for a set of changed fields etc.
7. After `Model.save_base` return `FieldTracker` resets initial state for updated fields (if no `update_fields` passed - whole initial state is reset).
8. `instance.refresh_from_db()` call causes initial state reset like for `save_base()`.

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