

---

# **django-model-utils Documentation**

***Release 4.3.1***

**Carl Meyer**

**Nov 16, 2022**



<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Setup	3
1.1.1	Installation	3
1.1.2	Dependencies	3
1.2	Fields	3
1.2.1	StatusField	3
1.2.2	MonitorField	4
1.2.3	SplitField	4
1.2.4	UUIDField	5
1.2.5	UrlsafeTokenField	6
1.3	Models	6
1.3.1	TimeFramedModel	6
1.3.2	TimeStampedModel	7
1.3.3	StatusModel	7
1.3.4	SoftDeletableModel	8
1.3.5	UUIDModel	8
1.3.6	SaveSignalHandlingModel	8
1.4	Model Managers	8
1.4.1	InheritanceManager	8
1.4.2	JoinManager	9
1.4.3	QueryManager	10
1.4.4	SoftDeletableManager	10
1.4.5	Mixins	10
1.5	Miscellaneous Utilities	10
1.5.1	Choices	10
1.5.2	Field Tracker	12
1.6	Changelog	17
1.6.1	4.3.1 (2022-11-15)	17
1.6.2	4.3.0	17
1.6.3	4.2.0 (2021-10-11)	17
1.6.4	4.1.1 (2020-12-01)	17
1.6.5	4.1.0 (2020-11-29)	17
1.6.6	4.0.0 (2019-12-11)	18
1.6.7	3.2.0 (2019.06.21)	18
1.6.8	3.1.2 (2018.05.09)	19
1.6.9	3.1.1 (2017.12.17)	19

1.6.10	3.1.0 (2017.12.11)	19
1.6.11	3.0.0 (2017.04.13)	19
1.6.12	2.6.1 (2017.01.11)	19
1.6.13	2.6 (2016.09.19)	20
1.6.14	2.5.2 (2016.08.09)	20
1.6.15	2.5.1 (2016.08.03)	20
1.6.16	2.5 (2016.04.18)	20
1.6.17	2.4 (2015.12.03)	20
1.6.18	2.3.1 (2015-07-20)	20
1.6.19	2.3 (2015.07.17)	20
1.6.20	2.2 (2014.07.31)	21
1.6.21	2.1.1 (2014.07.28)	21
1.6.22	2.1.0 (2014.07.25)	21
1.6.23	2.0.3 (2014.03.19)	21
1.6.24	2.0.2 (2014.02.19)	21
1.6.25	2.0.1 (2014.02.11)	21
1.6.26	2.0 (2014.01.06)	22
1.6.27	1.5.0 (2013.08.29)	22
1.6.28	1.4.0 (2013.06.03)	22
1.6.29	1.3.1 (2013.04.11)	23
1.6.30	1.3.0 (2013.03.27)	23
1.6.31	1.2.0 (2013.01.27)	23
1.6.32	1.1.0 (2012.04.13)	23
1.6.33	1.0.0 (2011.06.16)	24
1.6.34	0.6.0 (2011.02.18)	24
1.6.35	0.5.0 (2010.09.24)	24
1.6.36	0.4.0 (2010.03.16)	24
1.6.37	0.3.0	24
<b>2</b>	<b>Contributing</b>	<b>25</b>
<b>3</b>	<b>Indices and tables</b>	<b>27</b>

Django model mixins and utilities.



## 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 3.2+](#) (latest bugfix release in each series only) on Python 3.7+.

## 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):
```

(continues on next page)

(continued from previous page)

```
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)
```

## 1.2.5 UrIsafeTokenField

A CharField subclass that provides random token generating using python's `secrets.token_urlsafe` as default value.

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

`max_length` specifies the maximum length of the token. The default value is 128.

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

class MyAppModel(models.Model):
    uuid = UrIsafeTokenField(editable=False, max_length=128)
```

You can provide your custom token generator using the `factory` argument. `factory` should be callable. It will raise `TypeError` if it is not callable. `factory` is called with `max_length` argument to generate the token, and should return a string of specified maximum length.

```
import uuid

from django.db import models
from model_utils.fields import UrIsafeTokenField

def _token_factory(max_length):
    return uuid.uuid4().hex

class MyAppModel(models.Model):
    uuid = UrIsafeTokenField(max_length=32, factory=_token_factory)
```

## 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
```

(continues on next page)

(continued from previous page)

```

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 don't want to set `id` as primary key or change the field name, you can 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 heterogeneous 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')))
```

(continues on next page)

(continued from previous page)

```
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()`.

## When FieldTracker resets fields state

By the definition:

---

### Note:

- Field value *is changed* if it differs from current database value.

- Field value *was changed* if value has changed in database and field state didn't reset.

```
instance = Tracked.objects.get(pk=1)
# name not changed
instance.name += '_changed'
# name is changed
instance.save()
# name is not changed again
```

Current implementation resets fields state after `post_save` signals emitting. This is convenient for “outer” code like in example above, but does not help when model save method is overridden.

```
class MyModel(models.Model):
    name = models.CharField(max_length=64)
    tracker = FieldsTracker()

    def save(self): # erroneous implementation
        self.name = self.name.replace(' ', '_')
        name_changed = self.tracker.has_changed('name')
        super().save()
        # changed state has been reset here, so we need to store previous state_
↪ somewhere else
        if name_changed:
            do_something_about_it()
```

FieldTracker provides a context manager interface to postpone fields state reset in complicate situations.

- Fields state resets after exiting from outer-most context
- By default, all fields are reset, but field list can be provided
- Fields are counted separately depending on field list passed to context managers
- Tracker can be used as decorator
- Different instances have their own context state
- Different trackers in same instance have separate context state

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

    def save(self): # correct implementation
        self.name = self.name.replace(' ', '_')

        with self.tracker:
            super().save()
            # changed state reset is postponed
            if self.tracker.has_changed('name'):
                do_something_about_it()

    # Decorator example
    @tracker
    def save(self): ...

    # Restrict a set of fields to reset here
    @tracker(fields=('name'))
    def save(self): ...
```

(continues on next page)

(continued from previous page)

```
# Context manager with field list
def save(self):
    with self.tracker('name'):
        ...
```

## 1.6 Changelog

### 1.6.1 4.3.1 (2022-11-15)

- Confirm support for *Django* 4.0
- Add Spanish translation
- Add French translation
- Drop Django 1.7 workaround from *select\_subclasses()*
- Drop support for *Django* < 3.2
- Drop support for *Python* 3.6
- Confirm support for *Django* 4.1

### 1.6.2 4.3.0

- Never released due to packaging issues.

### 1.6.3 4.2.0 (2021-10-11)

- Add support for *Django* 3.2
- Drop support for *Django* 3.0
- Add support for *Python* 3.10
- Added urlsafe token field.
- Introduce context manager for FieldTracker state reset (GH-#491)
- Fix performance regression of FieldTracker on FileField subclasses on Django 3.1+ (GH-#498)

### 1.6.4 4.1.1 (2020-12-01)

- Applied *isort* to codebase (Refs GH-#402)
- Fix *TypeError* in save when model inherits from both *TimeStampModel* and *StatusModel*. (Fixes GH-465)

### 1.6.5 4.1.0 (2020-11-29)

#### Breaking changes:

- *FieldTracker* now marks fields as not changed after *refresh\_from\_db* respecting *fields* argument (GH-#404)

- *FieldTracker* now respects *update\_fields* changed in overridden *save()* method (GH-#404)
- *FieldTracker* now resets states after *pre\_save()* and not anymore *save()* signals, possibly altering the behaviour of overridden *save()* methods (GH-#404)

**Other changes:**

- Update *InheritanceQuerySetMixin* to avoid querying too much tables
- *TimeStampedModel* now automatically adds 'modified' field as an *update\_fields* parameter even if it is forgotten while using *save()*
- Replace *ugettext\_lazy* with *gettext\_lazy* to satisfy Django deprecation warning
- Add *available\_objects* manager to *SoftDeletableModel* and add deprecation warning to *objects* manager.
- *StatusModel* now automatically adds 'status\_changed' field during save as an *update\_fields* parameter when 'status' is present in it to make sure it is not forgotten.
- Update test requirements
- Move tests to GitHub Actions: <https://github.com/jazzband/django-model-utils/actions>
- Drop support for *Django 2.1*
- Add support for *Python 3.9*
- Add support for *Django 3.1*

## 1.6.6 4.0.0 (2019-12-11)

- Added *Choices.subset*.
- Remove hacks for previously supported Django versions. (Fixes GH-390)
- Dropped support for Python 2.7. (Fixes GH-393)
- Dropped usage of *six*
- Drop support for *Django 1.11*
- Add support for *Python 3.8*
- Add support for *Django 3.0*

## 1.6.7 3.2.0 (2019.06.21)

- Catch *AttributeError* for deferred abstract fields, fixes GH-331.
- Update documentation to explain usage of *timeframed* model manager, fixes GH-118
- Honor *OneToOneField.parent\_link=False*.
- Fix handling of deferred attributes on Django 1.10+, fixes GH-278
- Fix *FieldTracker.has\_changed()* and *FieldTracker.previous()* to return correct responses for deferred fields.
- Add Simplified Chinese translations.
- Update *AutoLastModifiedField* so that at instance creation it will always be set equal to created to make querying easier. Fixes GH-254
- Support *reversed* for all kinds of *Choices* objects, fixes GH-309
- Fix Model instance non picklable GH-330

- Fix patched *save* in *FieldTracker*
- Upgrades test requirements (pytest, pytest-django, pytest-cov) and skips tox test with Python 3.5 and Django (trunk)
- Add *UUIDModel* and *UUIDField* support.

### 1.6.8 3.1.2 (2018.05.09)

- Update *InheritanceIterable* to inherit from *ModelIterable* instead of *BaseIterable*, fixes GH-277.
- Add *all\_objects* Manager for '*SoftDeletableModel*' to include soft deleted objects on queries as per issue GH-255

### 1.6.9 3.1.1 (2017.12.17)

- Update classifiers and README via GH-306, fixes GH-305

### 1.6.10 3.1.0 (2017.12.11)

- Support for Django 2.0 via GH-298, fixes GH-297
- Remove old travis script via GH-300
- Fix codecov and switch to py.test #301

### 1.6.11 3.0.0 (2017.04.13)

- Drop support for Python 2.6.
- Drop support for Django 1.4, 1.5, 1.6, 1.7.
- Exclude tests from the distribution, fixes GH-258.
- Add support for Django 1.11 GH-269
- Add a new model to disable *pre\_save/post\_save* signals

### 1.6.12 2.6.1 (2017.01.11)

- Fix infinite recursion with multiple *MonitorField* and *defer()* or *only()* on Django 1.10+. Thanks Romain Garrigues. Merge of GH-242, fixes GH-241.
- Fix *InheritanceManager* and *SoftDeletableManager* to respect *self.\_queryset\_class* instead of hardcoding the queryset class. Merge of GH-250, fixes GH-249.
- Add mixins for *SoftDeletableQuerySet* and *SoftDeletableManager*, as stated in the the documentation.
- Fix *SoftDeletableModel.delete()* to use the correct database connection. Merge of GH-239.
- Added boolean keyword argument *soft* to *SoftDeletableModel.delete()* that revert to default behavior when set to *False*. Merge of GH-240.
- Enforced default manager in *StatusModel* to avoid manager order issues when using abstract models that redefine *objects* manager. Merge of GH-253, fixes GH-251.

### 1.6.13 2.6 (2016.09.19)

- Added *SoftDeletableModel* abstract class, its manager *SoftDeletableManager* and queryset *SoftDeletableQuerySet*.
- Fix issue with field tracker and deferred FileField for Django 1.10.

### 1.6.14 2.5.2 (2016.08.09)

- Include *runtests.py* in sdist.

### 1.6.15 2.5.1 (2016.08.03)

- Fix *InheritanceQuerySet* raising an *AttributeError* exception under Django 1.9.
- Django 1.10 support regressed with changes between pre-alpha and final release; 1.10 currently not supported.

### 1.6.16 2.5 (2016.04.18)

- Drop support for Python 3.2.
- Add support for Django 1.10 pre-alpha.
- Track foreign keys on parent models properly when a tracker is defined on a child model. Fixes GH-214.

### 1.6.17 2.4 (2015.12.03)

- Remove *PassThroughManager*. Use Django's built-in *QuerySet.as\_manager()* and/or *Manager.from\_queryset()* utilities instead.
- Add support for Django 1.9.

### 1.6.18 2.3.1 (2015-07-20)

- Remove all translation-related automation in *setup.py*. Fixes GH-178 and GH-179. Thanks Joe Weiss, Matt Molyneaux, and others for the reports.

### 1.6.19 2.3 (2015.07.17)

- Keep track of deferred fields on model instance instead of on FieldInstanceTracker instance. Fixes accessing deferred fields for multiple instances of a model from the same queryset. Thanks Bram Boogaard. Merge of GH-151.
- Fix Django 1.7 migrations compatibility for SplitField. Thanks ad-m. Merge of GH-157; fixes GH-156.
- Add German translations.
- Django 1.8 compatibility.



### 1.6.20 2.2 (2014.07.31)

- Revert GH-130, restoring ability to access `FieldTracker` changes in overridden `save` methods or `post_save` handlers. This reopens GH-83 (inability to pickle models with `FieldTracker`) until a solution can be found that doesn't break behavior otherwise. Thanks Brian May for the report. Fixes GH-143.

### 1.6.21 2.1.1 (2014.07.28)

- ASCII-fold all non-ASCII characters in changelog; again. Argh. Apologies to those whose names are mangled by this change. It seems that `distutils` makes it impossible to handle non-ASCII content reliably under Python 3 in a `setup.py` `long_description`, when the system encoding may be ASCII. Thanks Brian May for the report. Fixes GH-141.

### 1.6.22 2.1.0 (2014.07.25)

- Add support for Django's built-in migrations to `MonitorField` and `StatusField`.
- `PassThroughManager` now has support for seeing exposed methods via `dir`, allowing `IPython` tab completion to be useful. Merge of GH-104, fixes GH-55.
- Add pickle support for models using `FieldTracker`. Thanks Ondrej Slintak for the report. Thanks Matthew Schinckel for the fix. Merge of GH-130, fixes GH-83.

### 1.6.23 2.0.3 (2014.03.19)

- Fix `get_query_set` vs `get_queryset` in `PassThroughManager` for Django <1.6. Fixes issues with related managers not filtering by relation properly. Thanks whop, Bojan Mihelac, Daniel Shapiro, and Matthew Schinckel for the report; Matthew for the fix. Merge of GH-121.
- Fix `FieldTracker` with deferred model attributes. Thanks Michael van Tellingen. Merge of GH-115.
- Fix `InheritanceManager` with self-referential FK; avoid infinite recursion. Thanks rsenkbeil. Merge of GH-114.

### 1.6.24 2.0.2 (2014.02.19)

- ASCII-fold all non-ASCII characters in changelog. Apologies to those whose names are mangled by this change. It seems that `distutils` makes it impossible to handle non-ASCII content reliably under Python 3 in a `setup.py` `long_description`, when the system encoding may be ASCII. Thanks Simone Dalla for the report. Fixes GH-113.

### 1.6.25 2.0.1 (2014.02.11)

- Fix dependency to be on "Django" rather than "django", which plays better with static PyPI mirrors. Thanks Travis Swicegood.
- Fix issue with attempt to access `__slots__` when copying `PassThroughManager`. Thanks Patryk Zawadzki. Merge of GH-105.
- Improve `InheritanceManager` so any attributes added by using `extra(select)` will be propagated onto children. Thanks Curtis Maloney. Merge of GH-101, fixes GH-34.

- Added `InheritanceManagerMixin`, `InheritanceQuerySetMixin`, `PassThroughManagerMixin`, and `QueryManagerMixin` to allow composing their functionality with other custom manager/queryset subclasses (e.g. those in GeoDjango). Thanks Douglas Meehan!

### 1.6.26 2.0 (2014.01.06)

- **BACKWARDS-INCOMPATIBLE:** Indexing into a `Choices` instance now translates database representations to human-readable choice names, rather than simply indexing into an array of choice tuples. (Indexing into `Choices` was previously not documented.) If you have code that is relying on indexing or slicing `Choices`, the simplest workaround is to change e.g. `STATUS[1:]` to `list(STATUS)[1:]`.
- Fixed bug with checking for field name conflicts for added query managers on *StatusModel*.
- Can pass *choices\_name* to *StatusField* to use a different name for choices class attribute. `STATUS` is used by default.
- Can pass model subclasses, rather than strings, into *select\_subclasses()*. Thanks Keryn Knight. Merge of GH-79.
- Deepcopying a *Choices* instance no longer fails with infinite recursion in *getattr*. Thanks Leden. Merge of GH-75.
- *get\_subclass()* method is now available on both managers and querysets. Thanks Travis Swicegood. Merge of GH-82.
- Fix bug in *InheritanceManager* with grandchild classes on Django 1.6+; *select\_subclasses('child', 'child\_\_grandchild')* would only ever get to the child class. Thanks Keryn Knight for report and proposed fix.
- *MonitorField* now accepts a 'when' parameter. It will update only when the field changes to one of the values specified.

### 1.6.27 1.5.0 (2013.08.29)

- *Choices* now accepts option-groupings. Fixes GH-14.
- *Choices* can now be added to other *Choices* or to any iterable, and can be compared for equality with itself. Thanks Tony Aldridge. (Merge of GH-76.)
- *Choices* now `__contains__` its Python identifier values. Thanks Keryn Knight. (Merge of GH-69).
- Fixed a bug causing `KeyError` when saving with the parameter `update_fields` in which there are untracked fields. Thanks Mikhail Sionov. (Merge of GH-70, fixes GH-71).
- Fixed *FieldTracker* usage on inherited models. Fixes GH-57.
- Added mutable field support to *FieldTracker* (Merge of GH-73, fixes GH-74)

### 1.6.28 1.4.0 (2013.06.03)

- Introduced *FieldTracker* as replacement for *ModelTracker*, which is now deprecated.
- *PassThroughManager.for\_queryset\_class()* no longer ignores superclass `get_query_set`. Thanks Andy Freeland.
- Fixed *InheritanceManager* bug with grandchildren in Django 1.6. Thanks CrazyCasta.
- Fixed lack of `get_FOO_display` method for *StatusField*. Fixes GH-41.

### 1.6.29 1.3.1 (2013.04.11)

- Added explicit default to `BooleanField` in tests, for Django trunk compatibility.
- Fixed intermittent `StatusField` bug. Fixes GH-29.
- Added Python 3 support.
- Dropped support for Django 1.2 and 1.3. Django 1.4.2+ required.

### 1.6.30 1.3.0 (2013.03.27)

- Allow specifying default value for a `StatusField`. Thanks Felipe Prenholato.
- Fix calling `create()` on a `RelatedManager` that subclasses a dynamic `PassThroughManager`. Thanks SeiryuZ for the report. Fixes GH-24.
- Add workaround for <https://code.djangoproject.com/ticket/16855> in `InheritanceQuerySet` to avoid overriding prior calls to `select_related()`. Thanks ivirabyan.
- Added support for arbitrary levels of model inheritance in `InheritanceManager`. Thanks ivirabyan. (This feature only works in Django 1.6+ due to <https://code.djangoproject.com/ticket/16572>).
- Added `ModelTracker` for tracking field changes between model saves. Thanks Trey Hunner.

### 1.6.31 1.2.0 (2013.01.27)

- Moved primary development from [Bitbucket](#) to [GitHub](#). Bitbucket mirror will continue to receive updates; Bitbucket issue tracker will be closed once all issues tracked in it are resolved.
- Removed deprecated `ChoiceEnum`, `InheritanceCastModel`, `InheritanceCastManager`, and `manager_from`.
- Fixed pickling of `PassThroughManager`. Thanks Rinat Shigapov.
- Set `use_for_related_fields = True` on `QueryManager`.
- Added `__len__` method to `Choices`. Thanks Ryan Kaskel and James Oakley.
- Fixed `InheritanceQuerySet` on Django 1.5. Thanks Javier Garcia Sogo.

### 1.6.32 1.1.0 (2012.04.13)

- Updated `AutoCreatedField`, `AutoLastModifiedField`, `MonitorField`, and `TimeFramedModel` to use `django.utils.timezone.now` on Django 1.4. Thanks Donald Stufft.
- Fixed annotation of `InheritanceQuerysets`. Thanks Jeff Elmore and Facundo Gaich.
- Dropped support for Python 2.5 and Django 1.1. Both are no longer supported even for security fixes, and should not be used.
- Added `PassThroughManager.for_queryset_class()`, which fixes use of `PassThroughManager` with related fields. Thanks Ryan Kaskel for report and fix.
- Added `InheritanceManager.get_subclass()`. Thanks smacker.

### 1.6.33 1.0.0 (2011.06.16)

- Fixed using SplitField on an abstract base model.
- Fixed issue #8, adding `use_for_related_fields = True` to `InheritanceManager`.
- Added `PassThroughManager`. Thanks Paul McLanahan.
- Added pending-deprecation warnings for `InheritanceCastModel`, `manager_from`, and Django 1.1 support. Removed documentation for the deprecated utilities. Bumped `ChoiceEnum` from pending-deprecation to deprecation.
- Fixed issue #6, bug with `InheritanceManager` and descriptor fields (e.g. `FileField`). Thanks zygfried for the fix and sayane for tests.

### 1.6.34 0.6.0 (2011.02.18)

- updated SplitField to define `get_prep_value` rather than `get_db_prep_value`. This avoids deprecation warnings on Django trunk/1.3, but makes SplitField incompatible with Django versions prior to 1.2.
- added `InheritanceManager`, a better approach to selecting subclass instances for Django 1.2+. Thanks Jeff Elmore.
- added `InheritanceCastManager` and `InheritanceCastQuerySet`, to allow bulk casting of a queryset to child types. Thanks Gregor Muellegger.

### 1.6.35 0.5.0 (2010.09.24)

- added `manager_from` (thanks George Sakkis)
- added `StatusField`, `MonitorField`, `TimeFramedModel`, and `StatusModel` (thanks Jannis Leidel)
- deprecated `ChoiceEnum` and replaced with `Choices`

### 1.6.36 0.4.0 (2010.03.16)

- added `SplitField`
- added `ChoiceEnum`
- added South support for custom model fields

### 1.6.37 0.3.0

- Added `QueryManager`

## 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`