django-safedelete Documentation

Release 0.4

palkeo

Apr 05, 2023

Contents


coverage
97%

CHAPTER 1

What is it ?

For various reasons, you may want to avoid deleting objects from your database.

This Django application provides an abstract model, that allows you to transparently retrieve or delete your objects, without having them deleted from your database.

You can choose what happens when you delete an object :

- it can be masked from your database (soft delete, the default behavior)
- it can be masked from your database and mask any dependent models. (cascading soft delete)
- it can be normally deleted (hard delete)
- it can be hard-deleted, but if its deletion would delete other objects, it will only be masked
- it can be never deleted or masked from your database (no delete, use with caution)

CHAPTER 2

Example

```
# imports
from safedelete.models import SafeDeleteModel
from safedelete.models import HARD_DELETE_NOCASCADE

# Models

# We create a new model, with the given policy : Objects will be hard-deleted, or␣
↪soft deleted if other objects would have been deleted too.
class Article(SafeDeleteModel):
    _safedelete_policy = HARD_DELETE_NOCASCADE

    name = models.CharField(max_length=100)

class Order(SafeDeleteModel):
    _safedelete_policy = HARD_DELETE_NOCASCADE

    name = models.CharField(max_length=100)
    articles = models.ManyToManyField(Article)


# Example of use

>>> article1 = Article(name='article1')
>>> article1.save()

>>> article2 = Article(name='article2')
>>> article2.save()

>>> order = Order(name='order')
>>> order.save()
>>> order.articles.add(article1)

# This article will be masked, but not deleted from the database as it is still␣
↪referenced in an order.
```

(continues on next page)

(continued from previous page)

```
>>> article1.delete()

# This article will be deleted from the database.
>>> article2.delete()
```

CHAPTER 3

Compatibilities

- Branch 0.2.x is compatible with django >= 1.2
- Branch 0.3.x is compatible with django >= 1.4
- Branch 0.4.x is compatible with django >= 1.8
- Branch 0.5.x is compatible with django >= 1.11
- Branch 1.0.x, 1.1.x and 1.2.x are compatible with django >= 2.2
- Branch 1.3.x is compatible with django >= 3.2 and Python >= 3.7

Current branch (1.3.x) is tested with :

- Django 3.2 using python 3.7 to 3.10.
- Django 4.0 using python 3.8 to 3.10.
- Django 4.1 using python 3.8 to 3.10.
- Django 4.2 using python 3.8 to 3.11.

CHAPTER 4

Installation

Installing from pypi (using pip).

```
pip install django-safedelete
```

Installing from github.

```
pip install -e git://github.com/makinacorpus/django-safedelete.git#egg=django-
↪safedelete
```

Add `safedelete` in your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'safedelete',
    [...]
]
```

The application doesn't have any special requirement.

CHAPTER 5

Configuration

In the main django settings you can activate the boolean variable `SAFE_DELETE_INTERPRET_UNDELETED_OBJECTS_AS_CREAT`
If you do this the `update_or_create()` function from django's standard manager class will return `True` for the `created` variable if the object was soft-deleted and is now "revived".

By default, the field that indicates a database entry is soft-deleted is `deleted`, however, you can override the field name using the `SAFE_DELETE_FIELD_NAME` setting.

CHAPTER 6

Documentation

6.1 Model

6.1.1 Built-in model

class safedelete.models.**SafeDeleteModel**(**args*, ***kwargs*)

Abstract safedelete-ready model.

Note: To create your safedelete-ready models, you have to make them inherit from this model.

Attribute deleted DateTimeField set to the moment the object was deleted. Is set to `None` if the object has not been deleted.

Attribute deleted_by_cascade BooleanField set True whenever the object is deleted due cascade operation called by delete method of any parent Model. Default value is False. Later if its parent model calls for cascading undelete, it will restore only child classes that were also deleted by a cascading operation (deleted_by_cascade equals to True), i.e. all objects that were deleted before their parent deletion, should keep deleted if the same parent object is restored by undelete method.

If this behavior isn't desired, class that inherits from SafeDeleteModel can override this attribute by setting it as None: overriding model class won't have its `deleted_by_cascade` field and won't be restored by cascading undelete even if it was deleted by a cascade operation.

```
>>> class MyModel(SafeDeleteModel):
...     deleted_by_cascade = None
...     my_field = models.TextField()
```

Attribute _safedelete_policy define what happens when you delete an object. It can be one of `HARD_DELETE`, `SOFT_DELETE`, `SOFT_DELETE_CASCADE`, `NO_DELETE` and `HARD_DELETE_NOCASCADE`. Defaults to `SOFT_DELETE`.

```
>>> class MyModel(SafeDeleteModel):
...     _safedelete_policy = SOFT_DELETE
...     my_field = models.TextField()
...
>>> # Now you have your model (with its ``deleted`` field, and custom
↪manager and delete method)
```

Attribute objects The *safedelete.managers.SafeDeleteManager* returns the non-deleted models.

Attribute all_objects The *safedelete.managers.SafeDeleteAllManager* returns all the models (non-deleted and soft-deleted).

Attribute deleted_objects The *safedelete.managers.SafeDeleteDeletedManager* returns the soft-deleted models.

save(*keep_deleted=False*, ***kwargs*)
Save an object, un-deleting it if it was deleted.

Args: keep_deleted: Do not undelete the model if soft-deleted. (default: {False}) kwargs: Passed onto *save()*.

Note: Undeletes soft-deleted models by default.

---

undelete(*force_policy: Optional[int] = None*, ***kwargs*) → Tuple[int, Dict[str, int]]
Undelete a soft-deleted model.

Args: force_policy: Force a specific undelete policy. (default: {None}) kwargs: Passed onto *save()*.

Note: Will raise a `AssertionError` if the model was not soft-deleted.

---

classmethod has_unique_fields() → bool
Checks if one of the fields of this model has a unique constraint set (unique=True).

It also checks if the model has sets of field names that, taken together, must be unique.

Args: model: Model instance to check

class safedelete.models.**SafeDeleteMixin**(**args*, ***kwargs*)
`SafeDeleteModel` was previously named `SafeDeleteMixin`.

Deprecated since version 0.4.0: Use *SafeDeleteModel* instead.

6.1.2 Policies

You can change the policy of your model by setting its `_safedelete_policy` attribute. The different policies are:

safedelete.models.**HARD_DELETE**

This policy will:

- Hard delete objects from the database if you call the `delete()` method.

  There is no difference with « normal » models, but you can still manually mask them from the database, for example by using `obj.delete(force_policy=SOFT_DELETE)`.

`safedelete.models.`**`SOFT_DELETE`**
: This policy will:

  This will make the objects be automatically masked (and not deleted), when you call the delete() method. They will NOT be masked in cascade.

`safedelete.models.`**`SOFT_DELETE_CASCADE`**
: This policy will:

  This will make the objects be automatically masked (and not deleted) and all related objects, when you call the delete() method. They will be masked in cascade.

`safedelete.models.`**`HARD_DELETE_NOCASCADE`**

  This policy will:

  - Delete the object from database if no objects depends on it (e.g. no objects would have been deleted in cascade).
  - Mask the object if it would have deleted other objects with it.

`safedelete.models.`**`NO_DELETE`**

  This policy will:

  - Keep the objects from being masked or deleted from your database. The only way of removing objects will be by using raw SQL.

6.1.3 Policies Delete Logic Customization

Each of the policies has an overwritable function in case you need to customize a particular policy delete logic. The function per policy are as follows:

Policy	Overwritable Function
SOFT_DELETE	soft_delete_policy_action
HARD_DELETE	hard_delete_policy_action
HARD_DELETE_NOCASCADE	hard_delete_cascade_policy_action
SOFT_DELETE_CASCADE	soft_delete_cascade_policy_action

Example:

To add custom logic before or after the execution of the original delete logic of a model with the policy SOFT_DELETE you can overwrite the `soft_delete_policy_action` function as such:

```
def soft_delete_policy_action(self, **kwargs):
    # Insert here custom pre delete logic
    delete_response = super().soft_delete_policy_action(**kwargs)
    # Insert here custom post delete logic
    return delete_response
```

6.1.4 Fields uniqueness

Because unique constraints are set at the database level, set *unique=True* on a field will also check uniqueness against soft deleted objects. This can lead to confusion as the soft deleted objects are not visible by the user. This can be solved by setting a partial unique constraint that will only check uniqueness on non-deleted objects:

```
class Post(SafeDeleteModel):
    name = models.CharField(max_length=100)

    class Meta:
        constraints = [
            UniqueConstraint(
                fields=['name'],
                condition=Q(deleted__isnull=True),
                name='unique_active_name'
            ),
        ]
```

6.2 Managers

6.2.1 Built-in managers

class safedelete.managers.**SafeDeleteManager**(*queryset_class: Optional[Type[safedelete.queryset.SafeDeleteQueryset]] = None*)

Default manager for the SafeDeleteModel.

If _safedelete_visibility == DELETED_VISIBLE_BY_PK, the manager can returns deleted objects if they are accessed by primary key.

Attribute _safedelete_visibility define what happens when you query masked objects. It can be one of `DELETED_INVISIBLE` and `DELETED_VISIBLE_BY_PK`. Defaults to `DELETED_INVISIBLE`.

```
>>> from safedelete.models import SafeDeleteModel
>>> from safedelete.managers import SafeDeleteManager
>>> class MyModelManager(SafeDeleteManager):
...     _safedelete_visibility = DELETED_VISIBLE_BY_PK
...
>>> class MyModel(SafeDeleteModel):
...     _safedelete_policy = SOFT_DELETE
...     my_field = models.TextField()
...     objects = MyModelManager()
...
>>>
```

Attribute _queryset_class define which class for queryset should be used This attribute allows to add custom filters for both deleted and not deleted objects. It is `SafeDeleteQueryset` by default. Custom queryset classes should be inherited from `SafeDeleteQueryset`.

get_queryset()
Return a new QuerySet object. Subclasses can override this method to customize the behavior of the Manager.

all_with_deleted() → django.db.models.query.QuerySet
Show all models including the soft deleted models.

Note: This is useful for related managers as those don't have access to `all_objects`.

deleted_only() → django.db.models.query.QuerySet
Only show the soft deleted models.

Note: This is useful for related managers as those don't have access to `deleted_objects`.

all(***kwargs*) → django.db.models.query.QuerySet
Pass kwargs to `SafeDeleteQuerySet.all()`.

Args: force_visibility: Show deleted models. (default: {None})

Note: The `force_visibility` argument is meant for related managers when no other managers like `all_objects` or `deleted_objects` are available.

update_or_create(*defaults=None*, ***kwargs*) → Tuple[django.db.models.base.Model, bool]
See `()`.

Change to regular djangoesk function: Regular update_or_create() fails on soft-deleted, existing record with unique constraint on non-id field If object is soft-deleted we don't update-or-create it but reset the deleted field to None. So the object is visible again like a create in any other case.

Attention: If the object is "revived" from a soft-deleted state the created return value will still be false because the object is technically not created unless you set SAFE_DELETE_INTERPRET_UNDELETED_OBJECTS_AS_CREATED = True in the django settings.

Args: defaults: Dict with defaults to update/create model instance with kwargs: Attributes to lookup model instance with

static get_soft_delete_policies()
Returns all states which stand for some kind of soft-delete

class safedelete.managers.**SafeDeleteAllManager**(*queryset_class: Optional[Type[safedelete.queryset.SafeDeleteQueryset]] = None*)
SafeDeleteManager with `_safedelete_visibility` set to `DELETED_VISIBLE`.

Note: This is used in `safedelete.models.SafeDeleteModel.all_objects`.

class safedelete.managers.**SafeDeleteDeletedManager**(*queryset_class: Optional[Type[safedelete.queryset.SafeDeleteQueryset]] = None*)
SafeDeleteManager with `_safedelete_visibility` set to `DELETED_ONLY_VISIBLE`.

Note: This is used in `safedelete.models.SafeDeleteModel.deleted_objects`.

6.2.2 Visibility

A custom manager is used to determine which objects should be included in the querysets.

class safedelete.managers.**SafeDeleteManager**(*queryset_class: Optional[Type[safedelete.queryset.SafeDeleteQueryset]] = None*)
Default manager for the SafeDeleteModel.

If _safedelete_visibility == DELETED_VISIBLE_BY_PK, the manager can returns deleted objects if they are accessed by primary key.

Attribute _safedelete_visibility define what happens when you query masked objects. It can be one of `DELETED_INVISIBLE` and `DELETED_VISIBLE_BY_PK`. Defaults to `DELETED_INVISIBLE`.

```
>>> from safedelete.models import SafeDeleteModel
>>> from safedelete.managers import SafeDeleteManager
>>> class MyModelManager(SafeDeleteManager):
...     _safedelete_visibility = DELETED_VISIBLE_BY_PK
...
>>> class MyModel(SafeDeleteModel):
...     _safedelete_policy = SOFT_DELETE
...     my_field = models.TextField()
...     objects = MyModelManager()
...
>>>
```

Attribute _queryset_class define which class for queryset should be used This attribute allows to add custom filters for both deleted and not deleted objects. It is `SafeDeleteQueryset` by default. Custom queryset classes should be inherited from `SafeDeleteQueryset`.

If you want to change which objects are "masked", you can set the `_safedelete_visibility` attribute of the manager to one of the following:

`safedelete.managers.`**`DELETED_INVISIBLE`**

This is the default visibility.

The objects marked as deleted will be visible in one case : If you access them directly using a OneToOne or a ForeignKey relation.

For example, if you have an article with a masked author, you can still access the author using `article.author`. If the article is masked, you are not able to access it using reverse relationship : `author.article_set` will not contain the masked article.

`safedelete.managers.`**`DELETED_VISIBLE_BY_FIELD`**

This policy is like *`DELETED_INVISIBLE`*, except that you can still access a deleted object if you call the `get()` or `filter()` function, passing it the default field `pk` parameter. Configurable through the *_safedelete_visibility_field* attribute of the manager.

So, deleted objects are still available if you access them directly by this field.

6.3 QuerySet

6.3.1 Built-in QuerySet

class safedelete.queryset.**SafeDeleteQueryset**(*model: Optional[Type[django.db.models.base.Model]] = None, query: Optional[safedelete.query.SafeDeleteQuery] = None, using: Optional[str] = None, hints: Optional[Dict[str, django.db.models.base.Model]] = None*)

Default queryset for the SafeDeleteManager.

Takes care of "lazily evaluating" safedelete QuerySets. QuerySets passed within the `SafeDeleteQueryset` will have all of the models available. The deleted policy is evaluated at the very end of the chain when the QuerySet itself is evaluated.

classmethod as_manager()
Override as_manager behavior to ensure we create a SafeDeleteManager.

delete(*force_policy: Optional[int] = None*) → Tuple[int, Dict[str, int]]
Overrides bulk delete behaviour.

Note: The current implementation loses performance on bulk deletes in order to safely delete objects according to the deletion policies set.

See also:

`safedelete.models.SafeDeleteModel.delete()`

undelete(*force_policy: Optional[int] = None*) → Tuple[int, Dict[str, int]]
Undelete all soft deleted models.

Note: The current implementation loses performance on bulk undeletes in order to call the pre/post-save signals.

See also:

`safedelete.models.SafeDeleteModel.undelete()`

all(*force_visibility=None*) → _QS
Override so related managers can also see the deleted models.

A model's m2m field does not easily have access to *all_objects* and so setting *force_visibility* to True is a way of getting all of the models. It is not recommended to use *force_visibility* outside of related models because it will create a new queryset.

Args: force_visibility: Force a deletion visibility. (default: {None})

filter(**args*, ***kwargs*)
Return a new QuerySet instance with the args ANDed to the existing set.

6.4 Signals

6.4.1 Signals

There are two signals available. Please refer to the Django signals documentation on how to use them.

`safedelete.signals.`**`pre_softdelete`**

Sent before an object is soft deleted.

`safedelete.signals.`**`post_softdelete`**

Sent after an object has been soft deleted.

`safedelete.signals.`**`post_undelete`**

Sent after a deleted object is restored.

6.5 Handling administration

6.5.1 Model admin

Deleted objects will also be hidden in the admin site by default. A `ModelAdmin` abstract class is provided to give access to deleted objects.

An undelete action is provided to undelete objects in bulk. The `deleted` attribute is also excluded from editing by default.

You can use the `highlight_deleted` method to show deleted objects in red in the admin listing.

You also have the option of using `highlight_deleted_field` which is similar to `highlight_deleted`, but allows you to specify a field for sorting and representation. Whereas `highlight_deleted` uses your object's `__str__` function to represent the object, `highlight_deleted_field` uses the value from your object's specified field.

To use `highlight_deleted_field`, add "highlight_deleted_field" to your list filters (as a string, seen in the example below), and set *field_to_highlight = "desired_field_name"* (also seen below). Then you should also set its short description (again, see below).

class `safedelete.admin.`**SafeDeleteAdmin**(*model*, *admin_site*)

An abstract ModelAdmin which will include deleted objects in its listing.

Example

```
>>> from safedelete.admin import SafeDeleteAdmin,␣
↪SafeDeleteAdminFilter, highlight_deleted
>>> class ContactAdmin(SafeDeleteAdmin):
...     list_display = (highlight_deleted, "highlight_deleted_field",
↪"first_name", "last_name", "email") + SafeDeleteAdmin.list_display
...     list_filter = ("last_name", SafeDeleteAdminFilter,) +␣
↪SafeDeleteAdmin.list_filter
...
...     field_to_highlight = "id"
...
... ContactAdmin.highlight_deleted_field.short_description =␣
↪ContactAdmin.field_to_highlight
```

Python Module Index

S

Index