django-safedelete Documentation

Release 0.4

palkeo

Jun 19, 2017

Contents

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 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.
>>> 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

Current branch (0.4.x) has been tested with :

- Django 1.8 using python 2.7 and python 3.3 to 3.4.
- Django 1.9 using python 2.7 and python 3.4 to 3.5.
- Django 1.10 using python 2.7 and python 3.4 to 3.5.

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

Documentation

Model

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 _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* that returns the non-deleted models.

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

Attribute deleted_objects The *safedelete.managers.SafeDeleteDeletedManager* that 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(***kwargs*)
Undelete a soft-deleted model.

Args: kwargs: Passed onto *save()*.

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

---

delete(*force_policy=None*, ***kwargs*)
Overrides Django's delete behaviour based on the model's delete policy.

Args: force_policy: Force a specific delete policy. (default: {None}) kwargs: Passed onto *save()* if soft deleted.

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

Deprecated since version 0.4.0: Use *SafeDeleteModel* instead.

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.

Managers

Built-in managers

class safedelete.managers.**SafeDeleteManager**(*queryset_class=None*, **args*, ***kwargs*)

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.

all_with_deleted()

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()

Only show the soft deleted models.

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

---

all(***kwargs*)

Pass kwargs to SafeDeleteQuerySet.all().

Args: show_deleted: Show deleted models. (default: {False})

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

class `safedelete.managers.`**`SafeDeleteAllManager`**(*queryset_class=None*, **args*, ***kwargs*)
SafeDeleteManager with `_safedelete_visibility` set to `DELETED_VISIBLE`.

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

class `safedelete.managers.`**`SafeDeleteDeletedManager`**(*queryset_class=None*, **args*, ***kwargs*)
SafeDeleteManager with `_safedelete_visibility` set to `DELETED_ONLY_VISIBLE`.

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

Visibility

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

class `safedelete.managers.`**`SafeDeleteManager`**(*queryset_class=None*, **args*, ***kwargs*)
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.

QuerySet

Built-in QuerySet

class `safedelete.queryset.`**`SafeDeleteQueryset`**(*model=None*, *query=None*, *using=None*, *hints=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.

`delete`(*force_policy=None*)

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`()

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*)

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})

Signals

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.

Handling administration

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.

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

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

Example

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

Python Module Index

S

Index