Save The Change

Release 1.1.0

February 16, 2017

Contents

1 Developer Interface **1**

2 Internals **3**

3 History **5**
- 3.1 1.1.0 (05/16/2014) . . . 5
- 3.2 1.0.0 (09/08/2013) . . . 5

4 Save The Change **7**
- 4.1 Installation . . . 7
- 4.2 Usage . . . 8
- 4.3 How It Works . . . 8
- 4.4 Caveats . . . 8
- 4.5 Goodies . . . 8

CHAPTER 1

Developer Interface

save_the_change.decorators.**SaveTheChange**(*cls*)
Decorator that wraps models with a save hook to save only what's changed.

save_the_change.decorators.**UpdateTogether**(**groups*)
Decorator for specifying groups of fields to be updated together.

Usage:

```
>>> from django.db import models
>>> from save_the_change.decorators import SaveTheChange, UpdateTogether
>>>
>>> @SaveTheChange
>>> @UpdateTogether(
...     ('height_feet', 'height_inches'),
...     ('weight_pounds', 'weight_kilos')
... )
>>> class Knight(models.model):
>>>     ...
```

save_the_change.decorators.**TrackChanges**(*cls*)
Decorator that adds some methods and properties to models for working with changed fields.

has_changed `True` if any fields on the model have changed from its last known database representation.

changed_fields A `set` of the names of all changed fields on the model.

old_values The model's fields in their last known database representation as a read-only mapping (*OldValues*).

revert_fields() Reverts the given fields back to their last known database representation.

class save_the_change.mappings.**OldValues**(*instance*)
A read-only `Mapping` of the original values for its model.

Attributes can be accessed with either dot or bracket notation.

CHAPTER 2

Internals

class save_the_change.decorators.**STCMixin**(**args*, ***kwargs*)

Hooks into `__init__()`, `save()`, and `refresh_from_db()`, and adds some new, private attributes to the model:

_mutable_fields A `dict` storing a copy of potentially mutable values on first access.

_changed_fields A `dict` storing a copy of immutable fields' original values when they're changed.

save_the_change.decorators.**_inject_stc**(*cls*)

Wraps model attributes in descriptors to track their changes.

Injects a mixin into the model's __bases__ as well to handle the {create,load}/change/save lifecycle, and adds some attributes to the model's `_meta`:

_stc_injected `True` if we've already wrapped fields on this model.

_stc_save_hooks A `list` of hooks to run during `save()`.

save_the_change.decorators.**_save_the_change_save_hook**(*instance*, **args*, ***kwargs*)

Sets `update_fields` on `save()` to only what's changed.

`update_fields` is only set if it doesn't already exist and when doing so is safe. This means its not set if the instance is new and yet to be saved to the database, if the instance is being saved with a new primary key, or if `save()` has been called with `force_insert`.

Returns (continue_saving, args, kwargs)

Return type `tuple`

save_the_change.decorators.**_update_together_save_hook**(*instance*, **args*, ***kwargs*)

Sets `update_fields` on `save()` to include any fields that have been marked as needing to be updated together with fields already in `update_fields`.

Returns (continue_saving, args, kwargs)

Return type `tuple`

class save_the_change.descriptors.**ChangeTrackingDescriptor**(*name*, *django_descriptor=None*)

Descriptor that wraps model attributes to detect changes.

Not all fields in older versions of Django are represented by descriptors themselves, so we handle both getting/setting bare attributes on the model and calling out to descriptors if they exist.

save_the_change.descriptors.**_inject_descriptors**(*cls*)

Iterates over concrete fields in a model and wraps them in a descriptor to track their changes.

save_the_change.util.**is_mutable**(*obj*)

Checks if given object is likely mutable.

Parameters **obj** – object to check.

We check that the object is itself a known immutable type, and then attempt to recursively check any objects within it. Strings are special cased to prevent us getting stuck in an infinite loop.

Returns `True` if the object is likely mutable, `False` if it definitely is not.

Return type `bool`

CHAPTER 3

History

1.1.0 (05/16/2014)

- Add proper support for ForeignKeys (thanks to Brandon Konkle and Brian Wilson).
- Add update_together field to model Meta, via UpdateTogetherModel.

1.0.0 (09/08/2013)

- Initial release.

CHAPTER 4

Save The Change

Save The Change takes this:

```
>>> lancelot = Knight.objects.get(name="Sir Lancelot")
>>> lancelot.favorite_color = "Blue"
>>> lancelot.save()
```

And does this:

```
UPDATE "roundtable_knight"
SET "favorite_color" = 'Blue'
```

Instead of this:

```
UPDATE "roundtable_knight"
SET "name" = 'Sir Lancelot',
    "from" = 'Camelot',
    "quest" = 'To seek the Holy Grail.',
    "favorite_color" = 'Blue',
    "epithet" = 'The brave',
    "actor" = 'John Cleese',
    "full_name" = 'John Marwood Cleese',
    "height" = '6''11"',
    "birth_date" = '1939-10-27',
    "birth_union" = 'UK',
    "birth_country" = 'England',
    "birth_county" = 'Somerset',
    "birth_town" = 'Weston-Super-Mare',
    "facial_hair" = 'mustache',
    "graduated" = true,
    "university" = 'Cambridge University',
    "degree" = 'LL.B.',
```

Installation

Install Save The Change just like everything else:

```
$ pip install django-save-the-change
```

Usage

Just add the *SaveTheChange* decorator to your model:

```
from django.db import models
from save_the_change.decorators import SaveTheChange

@SaveTheChange
class Knight(models.model):
        ...
```

And that's it! Keep using Django like you always have, Save The Change will take care of you.

How It Works

Save The Change encapsulates the fields of your model with its own descriptors that track their values for any changes. When you call `save()`, Save The Change passes the names of your changed fields through Django's `update_fields` argument, and Django does the rest, sending only those fields back to the database.

Caveats

Save The Change can't help you with `ManyToManyField`s nor reverse relations, as those aren't handled through `save()`. But everything else should work.

Goodies

Save The Change also comes with two additional decorators, *TrackChanges* and *UpdateTogether*.

TrackChanges provides some additional properties and methods to keep interact with changes made to your model, including comparing the old and new values and reverting any changes to your model before you save it. It can be used independently of *SaveTheChange*.

UpdateTogether is an additional decorator which allows you to specify groups of fields that are dependent on each other in your model, ensuring that if any of them change they'll all be saved together. For example:

```
from django.db import models
from save_the_change.decorators import SaveTheChange, UpdateTogether

@SaveTheChange
@UpdateTogether(('height_feet', 'height_inches'))
class Knight(models.model):
        ...
```

Now if you ever make a change to either part of our Knight's height, *both* the feet and the inches will be sent to the database together, so that they can't accidentally fall out of sync.

Symbols

_inject_descriptors() (in module save_the_change.descriptors), 3
_inject_stc() (in module save_the_change.decorators), 3
_save_the_change_save_hook() (in module save_the_change.decorators), 3
_update_together_save_hook() (in module save_the_change.decorators), 3

C

ChangeTrackingDescriptor (class in save_the_change.descriptors), 3

I

is_mutable() (in module save_the_change.util), 3

O

OldValues (class in save_the_change.mappings), 1

S

SaveTheChange() (in module save_the_change.decorators), 1
STCMixin (class in save_the_change.decorators), 3

T

TrackChanges() (in module save_the_change.decorators), 1

U

UpdateTogether() (in module save_the_change.decorators), 1