django-changesets-doc Documentation

Release 0.0.1

David Townshend

Mar 29, 2017

Contents

1 Contents **3**
- 1.1 Models . . . 3

2 Installation **7**

3 Configuration **9**

4 Usage **11**
- 4.1 Model Overview . . . 11
- 4.2 Tracking Changes . . . 11
- 4.3 Querying History . . . 12

Python Module Index **13**

Warning: This app is very much a work in progress, and far from complete or stable. This documentation is currently just a roadmap and sections of it have not yet been implemented. Once the documentation matches the code this notice will be removed.

If anyone would like to contribute to the code, fork it and create a pull request.

django-changesets is a history tracking app for Django.

CHAPTER 1

Contents

Models

Recording Functions

record_changeset(*[**kwargs]*)
Return a context manager which records all changes made into a *ChangeSet*. If any keyword arguments are give, they are passed to *ChangeSet* when it is created as the context manager exits.

start_recording(*[**kwargs]*)
This, coupled with *stop_recording*, work the same as *record_changeset*. It is important to ensure that *stop_recording* is called (e.g. in a `finally` clause) or the changeset will never be created.

stop_recording()
See *start_recording*

scan(*objects*[, *delete_only=False*])
This must be called during recording (e.g. within *record_changeset*).

It scans a group of objects for changes after they have been made, and is ideal for situations where recorder cannot track the changes as they are made, or for populating initial data.

objects can be the name of a django app, a django `Model`, a `Model` instance, or an iterable of any of these. Deleted instances can be included and will be identified as such by their lack of primary key. If *delete_only* is `True`, then the *objects* will only be scanned for deleted objects. This is useful when, for example, `bulk_delete` has just been used.

Object wrapper

One of the fundamental concepts of this app is that the original models should always be an accurate representation of the current data state. The main implication of this is that deleted records should actually be deleted, so a query of `MyModel.objects.all()` doesn't return an deleted or inaccurate data. However, it is useful to maintain a history of deleted records and historical foreign keys may refer to deleted records, so every reference to a record is wrapped in by the *ObjectWrapper* model.

class ObjectWrapper
A `GenericForeignKey` is used to reference the instance, so the usual caveats regarding these apply.

instance
Return the model instance being wrapped by this object. If it does not exist (because it has been deleted), then `None` is returned.

is_deleted()
Return `True` if the instance does not exist.

A custom manager provides a *wrap* method, to easily create new *ObjectWrappers*.

`ObjectWrapperManager.`**wrap**(*instance*)
Return an *ObjectWrapper* wrapping *instance*. This is functionally similar to `get_or_create`.

Field types

Individual changes to an instance are stored by field. To properly manage this, a *FieldType* model is used. It is similar in concept to `ContentType`, but operates at field level instead.

class FieldType
A *FieldType* is identified by the model (via a foreign key to `ContentType`) and the *field_name*. Additionally, for relation fields, the related model is automatically captured in *rel_content_type*.

model
Return the model containing the field.

field
Return the actual field object (e.g. a `CharField` instance).

is_fk()
Return `True` if the field is a foreign key.

is_m2m()
Return `True` if the field is a `ManyToManyField`.

clean_value(*value*)
Return a cleaned value using `field.clean`. If the value is invalid, `ValidationError` will be raise.

A custom manager is used to provide additional features.

class FieldTypeManager

for_field(*model*, *field_name*)
Get a *FieldType* for the specified model and field name. If the field does not exist on the model, `FieldDoesNotExist` is raised.

get_by_model(*model*[, *related=True*])
Filter all *FieldTypes* by model. If *related* is `True`, the fields on other models with a foreign or many-to-many key pointing to *model* will be included.

Changesets

class ChangeSet

user
The user who made the change. This is a `User` object from the `django.contrib.auth` framework.

comment
Optional comments related to the changeset. This is stored in a `TextField`.

Internals

CHANGE_STATE
`(('+', 'add'), ('-', 'delete'))`

The constant defines the choices for *State.state* and *M2MChange.change_type*.

class State
When a record is added or deleted, the change in state is recorded in this model.

changeset
Return the containing changeset

state
This indicates if the objects was added or deleted. This is a field defined with choices *CHANGE_STATE*, so django provides a way of getting the description via `get_state_display()`.

wrapper
The *ObjectWrapper* wrapping the instance.

instance
This property is a shortcut to return `state.wrapper.instance`.

See also:

ObjectWrapper.instance

class ValueChange
Individual changes are stored in this model.

changeset
Return the containing changeset

wrapper
The *ObjectWrapper* wrapping the instance.

instance
This property is a shortcut to return `state.wrapper.instance`.

See also:

ObjectWrapper.instance

field
A *FieldType* instance referring to the field changed.

value
Todo

class M2MChange
This model inherits directly from *ValueChange* and adds a single field.

change_type
This indicates if the object referenced in *value* was added or deleted. This is a field defined with choices *CHANGE_STATE*, so django provides a way of getting the description via `get_change_type_display()`.

Middleware

class ChangeSetMiddleware
Todo

CHAPTER 2

Installation

The easiest is to install with pip:

```
pip install django-changesets
```

CHAPTER 3

Configuration

django-changesets requires the *auth* module, so the first configuration step is to add the necessary apps to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    'django.contrib.auth',
    'changesets',
    ...
)
```

Optionally, install the middleware after the *auth* middleware (see *Middleware* for details):

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'changesets.middleware.ChangeSetMiddleware',
    ...
)
```

If the models already contain data, then bring create an initial changeset by running:

```
with record_changeset(comment='Initial data'):
    scan('myapp')
```

CHAPTER 4

Usage

This app aims be provide a flexible and stable method of recording changes to data by encapsulating a group of changes into a *ChangeSet*, stamped with the time and user who made the changes.

One of the more powerful features provided is the ability of record a changeset after the fact (this also makes it very easy to add the app to existing data). Since every change is simply a database record, the history is also mutable, although changes to the history need to be done with care since they could result in invalid data (e.g. broken relationships).

This is designed to work with all relationship fields, including `ForeignKey`, `ManyToManyField` and `GenericForeignKey`.

Model Overview

The primary model in this app is *ChangeSet*. This represents a collection of individual changes made at a specific point in time by a single user. Individual changes are recorded by the object, field, and value changed.

To ensure stability of the changesets, changed model instances are not referenced directly, instead they are wrapped in an *ObjectWrapper* model. The reason for this is that records in this model are never deleted, meaning that changes relating to deleted objects can be kept and still be sensibly queried.

Fields are represented by *FieldType*, which is a bit like a `ContentType`, but for fields instead of models.

Changed values are converted to text and stored in a `TextField`.

Tracking Changes

The most basic method of recording a changeset is through *record_changeset*. For example:

```
with record_changeset(user=my_user, comment='Some changes'):
    # Change a value
    obj.field = 2
    obj.save()
    # Add something
    MyModel.objects.create(value='new object')
    # Delete something
    old_obj.delete()
```

```
    # For a bulk operation we need to find the changes are they are made
    queryset.update(my_value='new value')
    scan(queryset)
```

Middleware

Often, changesets will be wanted for all changes made by a user through a view, and this can be implemented simply by adding *ChangeSetMiddleware* below `AuthenticationMiddleware` in `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'changesets.middleware.ChangeSetMiddleware',
    ...
)
```

This has the effect of wrapping every view in a ChangeSet and automatically assigning the current user. Note that bulk operations must still be dealt with explicitly.

Introspection

If changes were made and not recorded (often because of a bulk operation, or after setting up changesets on an existing project), they can be pulled into a changeset afterwards using *scan*. For example:

```
with record_changeset(comment='Bulk operations just happened'):
    # We know which objects were only added and changed
    scan(changed_queryset)
    # We know that there were only deletions here
    scan(OtherModel, delete_only=True)
    # Lot of stuff happened in this model
    scan(MessyModel)
    # This entire app has changes
    scan('myapp')
```

Querying History

ChangeSets are just models, so they can be queried just the same as any other model.

Python Module Index

C

changesets, 1

Index

C

CHANGE_STATE (in module changesets), 5
change_type (M2MChange attribute), 5
ChangeSet (class in changesets), 4
changeset (State attribute), 5
changeset (ValueChange attribute), 5
ChangeSetMiddleware (class in changesets), 5
changesets (module), 1, 3
clean_value() (FieldType method), 4
comment (ChangeSet attribute), 4

F

field (FieldType attribute), 4
field (ValueChange attribute), 5
FieldType (class in changesets), 4
FieldTypeManager (class in changesets), 4
for_field() (FieldTypeManager method), 4

G

get_by_model() (FieldTypeManager method), 4

I

instance (ObjectWrapper attribute), 3
instance (State attribute), 5
instance (ValueChange attribute), 5
is_deleted() (ObjectWrapper method), 4
is_fk() (FieldType method), 4
is_m2m() (FieldType method), 4

M

M2MChange (class in changesets), 5
model (FieldType attribute), 4

O

ObjectWrapper (class in changesets), 3

R

record_changeset() (in module changesets), 3

S

scan() (in module changesets), 3
start_recording() (in module changesets), 3
State (class in changesets), 5
state (State attribute), 5
stop_recording() (in module changesets), 3

U

user (ChangeSet attribute), 4

V

value (ValueChange attribute), 5
ValueChange (class in changesets), 5

W

wrap() (ObjectWrapperManager method), 4
wrapper (State attribute), 5
wrapper (ValueChange attribute), 5