django-simple-history Documentation

Release 2.0

Corey Bertram

Dec 10, 2018

Contents

django-simple-history stores Django model state on every create/update/delete.

CHAPTER 1

Documentation

1.1 Quick Start

1.1.1 Install

Install from PyPI with `pip`:

```
$ pip install django-simple-history
```

1.1.2 Configure

Settings

Add `simple_history` to your `INSTALLED_APPS`

```
INSTALLED_APPS = [
    # ...
    'simple_history',
]
```

The historical models can track who made each change. To populate the history user automatically you can add middleware to your Django settings:

```
MIDDLEWARE = [
    # ...
    'simple_history.middleware.HistoryRequestMiddleware',
]
```

If you do not want to use the middleware, you can explicitly indicate the user making the change as documented in *Advanced Usage*.

Models

To track history for a model, create an instance of `simple_history.models.HistoricalRecords` on the model.

An example for tracking changes on the `Poll` and `Choice` models in the Django tutorial:

```
from django.db import models
from simple_history.models import HistoricalRecords

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    history = HistoricalRecords()

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
    history = HistoricalRecords()
```

Now all changes to `Poll` and `Choice` model instances will be tracked in the database.

Run Migrations

With your model changes in place, create and apply the database migrations:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Existing Projects

For existing projects, you can call the populate command to generate an initial change for preexisting model instances:

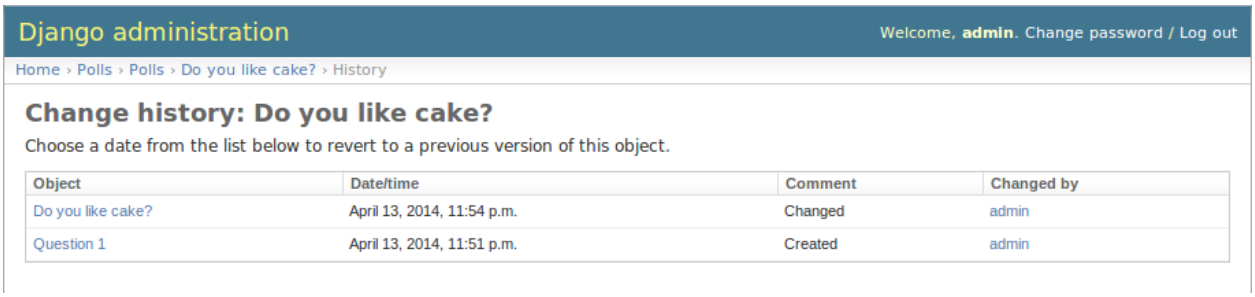
```
$ python manage.py populate_history --auto
```

By default, history rows are inserted in batches of 200. This can be changed if needed for large tables by using the `--batchsize` option, for example `--batchsize 500`.

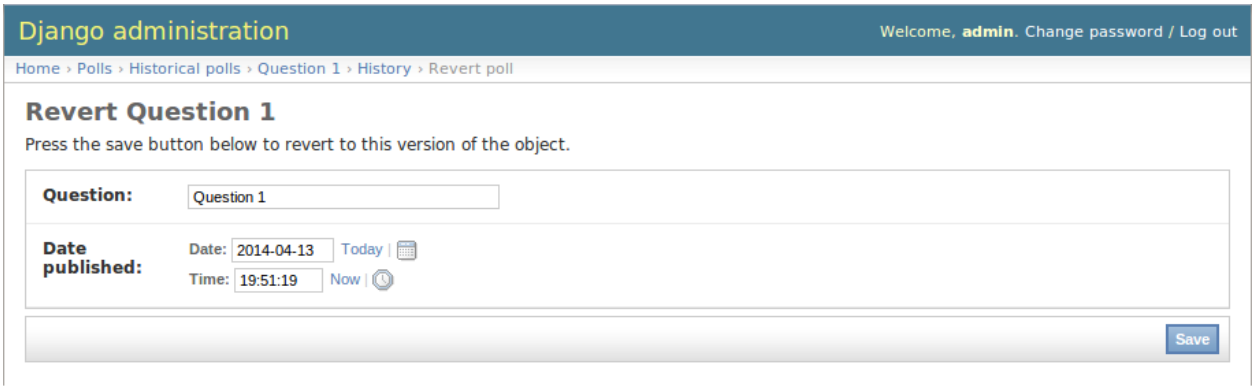
1.1.3 Integration with Django Admin

To allow viewing previous model versions on the Django admin site, inherit from the `simple_history.admin.SimpleHistoryAdmin` class when registering your model with the admin site.

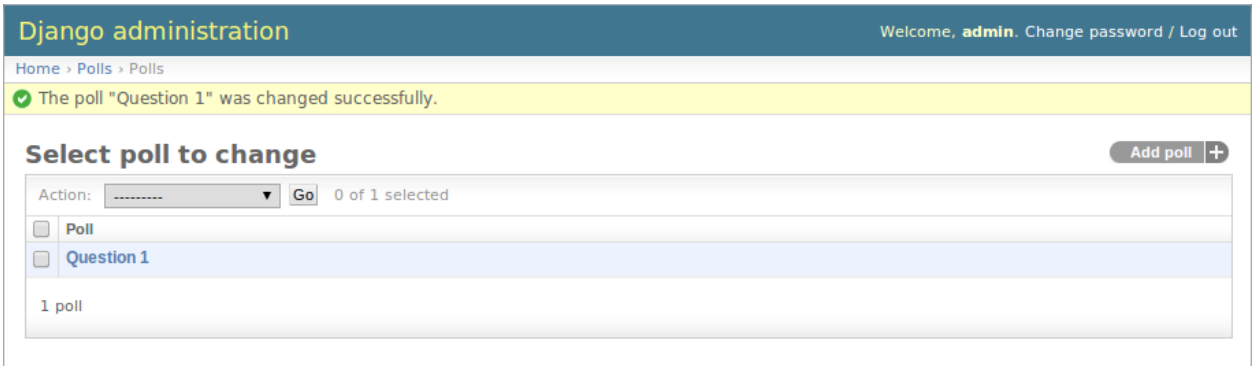
This will replace the history object page on the admin site and allow viewing and reverting to previous model versions. Changes made in admin change forms will also accurately note the user who made the change.



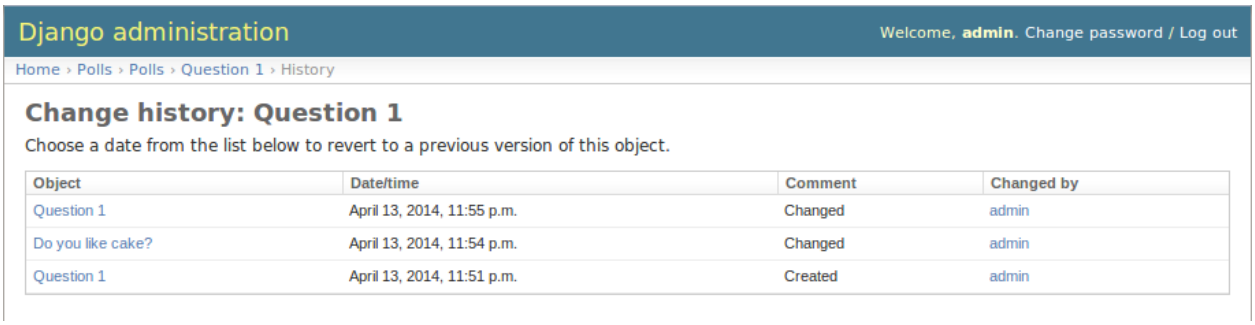
Clicking on an object presents the option to revert to that version of the object.



(The object is reverted to the selected state)



Reversions like this are added to the history.



An example of admin integration for the `Poll` and `Choice` models:

```
from django.contrib import admin
from simple_history.admin import SimpleHistoryAdmin
```

(continues on next page)

(continued from previous page)

```
from .models import Poll, Choice

admin.site.register(Poll, SimpleHistoryAdmin)
admin.site.register(Choice, SimpleHistoryAdmin)
```

Changing a history-tracked model from the admin interface will automatically record the user who made the change (see *Advanced Usage*).

Displaying custom columns in the admin history list view

By default, the history log displays one line per change containing

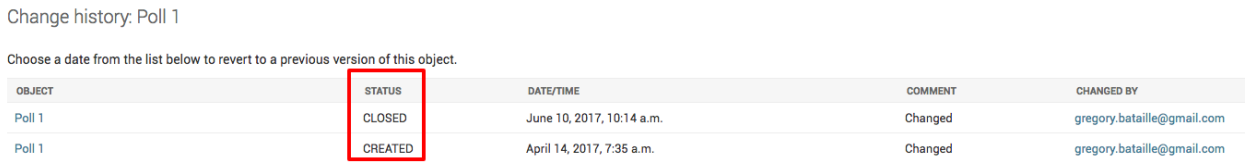
- a link to the detail of the object at that point in time
- the date and time the object was changed
- a comment corresponding to the change
- the author of the change

You can add other columns (for example the object's status to see how it evolved) by adding a `history_list_display` array of fields to the admin class

```
from django.contrib import admin
from simple_history.admin import SimpleHistoryAdmin
from .models import Poll, Choice


class PollHistoryAdmin(SimpleHistoryAdmin):
    list_display = ["id", "name", "status"]
    history_list_display = ["status"]
    search_fields = ['name', 'user__username']

admin.site.register(Poll, PollHistoryAdmin)
admin.site.register(Choice, SimpleHistoryAdmin)
```



1.1.4 Querying history

Querying history on a model instance

The `HistoricalRecords` object on a model instance can be used in the same way as a model manager:

```
>>> from polls.models import Poll, Choice
>>> from datetime import datetime
>>> poll = Poll.objects.create(question="what's up?", pub_date=datetime.now())
>>>
>>> poll.history.all()
[<HistoricalPoll: Poll object as of 2010-10-25 18:03:29.855689>]
```

Whenever a model instance is saved a new historical record is created:

```
>>> poll.pub_date = datetime(2007, 4, 1, 0, 0)
>>> poll.save()
>>> poll.history.all()
[<HistoricalPoll: Poll object as of 2010-10-25 18:04:13.814128>, <HistoricalPoll:␣
↪Poll object as of 2010-10-25 18:03:29.855689>]
```

Querying history on a model class

Historical records for all instances of a model can be queried by using the `HistoricalRecords` manager on the model class. For example historical records for all `Choice` instances can be queried by using the manager on the `Choice` model class:

```
>>> choice1 = poll.choice_set.create(choice_text='Not Much', votes=0)
>>> choice2 = poll.choice_set.create(choice_text='The sky', votes=0)
>>>
>>> Choice.history
<simple_history.manager.HistoryManager object at 0x1cc4290>
>>> Choice.history.all()
[<HistoricalChoice: Choice object as of 2010-10-25 18:05:12.183340>,
↪<HistoricalChoice: Choice object as of 2010-10-25 18:04:59.047351>]
```

Because the history is model, you can also filter it like regularly QuerySets, a.k. Choice.history.filter(choice_text='Not Much') will work!

1.2 Common Issues

- `fields.E300`:

  ```
  ERRORS:
  custom_user.HistoricalCustomUser.history_user: (fields.E300) Field defines a␣
  ↪relation with model 'custom_user.CustomUser', which is either not installed, or␣
  ↪is abstract.
  ```

  Use `register()` to track changes to the custom user model instead of setting `HistoricalRecords` on the model directly. See *History for a Third-Party Model*.

  The reason for this, is that unfortunately `HistoricalRecords` cannot be set directly on a swapped user model because of the user foreign key to track the user making changes.

1.3 Advanced Usage

1.3.1 Database Migrations

By default, Historical models live in the same app as the model they track. Historical models are tracked by migrations in the same way as any other model. Whenever the original model changes, the historical model will change also.

Therefore tracking historical models with migrations should work automatically.

1.3.2 Locating past model instance

Two extra methods are provided for locating previous models instances on historical record model managers.

as_of

This method will return an instance of the model as it would have existed at the provided date and time.

```
>>> from datetime import datetime
>>> poll.history.as_of(datetime(2010, 10, 25, 18, 4, 0))
<Poll: Poll object as of 2010-10-25 18:03:29.855689>
>>> poll.history.as_of(datetime(2010, 10, 25, 18, 5, 0))
<Poll: Poll object as of 2010-10-25 18:04:13.814128>
```

most_recent

This method will return the most recent copy of the model available in the model history.

```
>>> from datetime import datetime
>>> poll.history.most_recent()
<Poll: Poll object as of 2010-10-25 18:04:13.814128>
```

1.3.3 History for a Third-Party Model

To track history for a model you didn't create, use the `simple_history.register` utility. You can use this to track models from third-party apps you don't have control over. Here's an example of using `simple_history.register` to history-track the `User` model from the `django.contrib.auth` app:

```
from simple_history import register
from django.contrib.auth.models import User

register(User)
```

1.3.4 Allow tracking to be inherited

By default history tracking is only added for the model that is passed to `register()` or has the `HistoricalRecords` descriptor. By passing `inherit=True` to either way of registering you can change that behavior so that any child model inheriting from it will have historical tracking as well. Be careful though, in cases where a model can be tracked more than once, `MultipleRegistrationsError` will be raised.

```
from django.contrib.auth.models import User
from django.db import models
from simple_history import register
from simple_history.models import HistoricalRecords

# register() example
register(User, inherit=True)

# HistoricalRecords example
class Poll(models.Model):
    history = HistoricalRecords(inherit=True)
```

Both `User` and `Poll` in the example above will cause any model inheriting from them to have historical tracking as well.

1.3.5 Recording Which User Changed a Model

To denote which user changed a model, assign a `_history_user` attribute on your model.

For example if you have a `changed_by` field on your model that records which user last changed the model, you could create a `_history_user` property referencing the `changed_by` field:

```
from django.db import models
from simple_history.models import HistoricalRecords

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    changed_by = models.ForeignKey('auth.User')
    history = HistoricalRecords()

    @property
    def _history_user(self):
        return self.changed_by

    @_history_user.setter
    def _history_user(self, value):
        self.changed_by = value
```

Admin integration requires that you use a `_history_user.setter` attribute with your custom `_history_user` property (see *Integration with Django Admin*).

1.3.6 Custom `history_date`

You're able to set a custom `history_date` attribute for the historical record, by defining the property `_history_date` in your model. That's helpful if you want to add versions to your model, which happened before the current model version, e.g. when batch importing historical data. The content of the property `_history_date` has to be a datetime-object, but setting the value of the property to a `DateTimeField`, which is already defined in the model, will work too.

```
from django.db import models
from simple_history.models import HistoricalRecords

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    changed_by = models.ForeignKey('auth.User')
    history = HistoricalRecords()
    __history_date = None

    @property
    def _history_date(self):
        return self.__history_date

    @_history_date.setter
    def _history_date(self, value):
        self.__history_date = value
```

```
from datetime import datetime
from models import Poll

```

(continues on next page)

(continued from previous page)

```
my_poll = Poll(question="what's up?")
my_poll._history_date = datetime.now()
my_poll.save()
```

1.3.7 Change Base Class of HistoricalRecord Models

To change the auto-generated HistoricalRecord models base class from `models.Model`, pass in the abstract class in a list to `bases`.

```
class RoutableModel(models.Model):
    class Meta:
        abstract = True


class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    changed_by = models.ForeignKey('auth.User')
    history = HistoricalRecords(bases=[RoutableModel])
```

1.3.8 Custom history table name

By default, the table name for historical models follow the Django convention and just add `historical` before model name. For instance, if your application name is `polls` and your model name `Question`, then the table name will be `polls_historicalquestion`.

You can use the `table_name` parameter with both `HistoricalRecords()` or `register()` to change this behavior.

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    history = HistoricalRecords(table_name='polls_question_history')
```

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

register(Question, table_name='polls_question_history')
```

1.3.9 Choosing fields to not be stored

It is possible to use the parameter `excluded_fields` to choose which fields will be stored on every create/update/delete.

For example, if you have the model:

```
class PollWithExcludeFields(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
```

And you don't want to store the changes for the field `pub_date`, it is necessary to update the model to:

```
class PollWithExcludeFields(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    history = HistoricalRecords(excluded_fields=['pub_date'])
```

By default, django-simple-history stores the changes for all fields in the model.

1.3.10 Change Reason

Change reason is a message to explain why the change was made in the instance. It is stored in the field `history_change_reason` and its default value is `None`.

By default, the django-simple-history gets the change reason in the field `changeReason` of the instance. Also, is possible to pass the `changeReason` explicitly. For this, after a save or delete in an instance, is necessary call the function `utils.update_change_reason`. The first argument of this function is the instance and the seccond is the message that represents the change reason.

For instance, for the model:

```
from django.db import models
from simple_history.models import HistoricalRecords

class Poll(models.Model):
    question = models.CharField(max_length=200)
    history = HistoricalRecords()
```

You can create a instance with a implicity change reason.

```
poll = Poll(question='Question 1')
poll.changeReason = 'Add a question'
poll.save()
```

Or you can pass the change reason explicitly:

```
from simple_history.utils import update_change_reason

poll = Poll(question='Question 1')
poll.save()
update_change_reason(poll, 'Add a question')
```

1.3.11 Save without a historical record

If you want to save a model without a historical record, you can use the following:

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    history = HistoricalRecords()

    def save_without_historical_record(self, *args, **kwargs):
        self.skip_history_when_saving = True
        try:
            ret = self.save(*args, **kwargs)
        finally:
            del self.skip_history_when_saving
```

(continues on next page)

(continued from previous page)

```
        return ret


poll = Poll(question='something')
poll.save_without_historical_record()
```

CHAPTER 2

Code

Code and issue tracker: https://github.com/treyhunner/django-simple-history

Pull requests are welcome.

CHAPTER 3

Changes

3.1 2.0 (2018-04-05)

- Added Django 2.0 support (gh-330)
- Dropped support for Django<=1.10 (gh-356)
- Fix bug where history_view ignored user permissions (gh-361)
- Fixed HistoryRequestMiddleware which hadn't been working for Django>1.9 (gh-364)

3.2 1.9.1 (2018-03-30)

- Use get_queryset rather than model.objects in history_view. (gh-303)
- Change ugettext calls in models.py to ugettext_lazy
- Resolve issue where model references itself (gh-278)
- Fix issue with tracking an inherited model (abstract class) (gh-269)
- Fix history detail view on django-admin for abstract models (gh-308)
- Dropped support for Django<=1.6 and Python 3.3 (gh-292)

3.3 1.9.0 (2017-06-11)

- Add –batchsize option to the populate_history management command. (gh-231)
- Add ability to show specific attributes in admin history list view. (gh-256)
- Add Brazilian Portuguese translation file. (gh-279)
- Fix locale file packaging issue. (gh-280)

- Add ability to specify reason for history change. (gh-275)
- Test against Django 1.11 and Python 3.6. (gh-276)
- Add *excluded_fields* option to exclude fields from history. (gh-274)

3.4 1.8.2 (2017-01-19)

- Add Polish locale.
- Add Django 1.10 support.

3.5 1.8.1 (2016-03-19)

- Clear the threadlocal request object when processing the response to prevent test interactions. (gh-213)

3.6 1.8.0 (2016-02-02)

- History tracking can be inherited by passing *inherit=True*. (gh-63)

3.7 1.7.0 (2015-12-02)

- Add ability to list history in admin when the object instance is deleted. (gh-72)
- Add ability to change history through the admin. (Enabled with the *SIMPLE_HISTORY_EDIT* setting.)
- Add Django 1.9 support.
- Support for custom tables names. (gh-196)

3.8 1.6.3 (2015-07-30)

- Respect *to_field* and *db_column* parameters (gh-182)

3.9 1.6.2 (2015-07-04)

- Use app loading system and fix deprecation warnings on Django 1.8 (gh-172)
- Update Landscape configuration

3.10 1.6.1 (2015-04-21)

- Fix OneToOneField transformation for historical models (gh-166)
- Disable cascading deletes from related models to historical models
- Fix restoring historical instances with missing one-to-one relations (gh-162)

3.11 1.6.0 (2015-04-16)

- Add support for Django 1.8+
- Deprecated use of `CustomForeignKeyField` (to be removed)
- Remove default reverse accessor to *auth.User* for historical models (gh-121)

3.12 1.5.4 (2015-01-03)

- Fix a bug when models have a `ForeignKey` with `primary_key=True`
- Do NOT delete the history elements when a user is deleted.
- Add support for `latest`
- Allow setting a reason for change. [using option changeReason]

3.13 1.5.3 (2014-11-18)

- Fix migrations while using `order_with_respsect_to` (gh-140)
- Fix migrations using south
- Allow history accessor class to be overridden in `register()`

3.14 1.5.2 (2014-10-15)

- Additional fix for migrations (gh-128)

3.15 1.5.1 (2014-10-13)

- Removed some incompatibilities with non-default admin sites (gh-92)
- Fixed error caused by `HistoryRequestMiddleware` during anonymous requests (gh-115 fixes gh-114)
- Added workaround for clashing related historical accessors on User (gh-121)
- Added support for MongoDB AutoField (gh-125)
- Fixed CustomForeignKeyField errors with 1.7 migrations (gh-126 fixes gh-124)

3.16 1.5.0 (2014-08-17)

- Extended availability of the `as_of` method to models as well as instances.
- Allow `history_user` on historical objects to be set by middleware.
- Fixed error that occurs when a foreign key is designated using just the name of the model.
- Drop Django 1.3 support

3.17 1.4.0 (2014-06-29)

- Fixed error that occurs when models have a foreign key pointing to a one to one field.
- Fix bug when model verbose_name uses unicode (gh-76)
- Allow non-integer foreign keys
- Allow foreign keys referencing the name of the model as a string
- Added the ability to specify a custom `history_date`
- Note that `simple_history` should be added to `INSTALLED_APPS` (gh-94 fixes gh-69)
- Properly handle primary key escaping in admin URLs (gh-96 fixes gh-81)
- Add support for new app loading (Django 1.7+)
- Allow specifying custom base classes for historical models (gh-98)

3.18 1.3.0 (2013-05-17)

- Fixed bug when using `django-simple-history` on nested models package
- Allow history table to be formatted correctly with `django-admin-bootstrap`
- Disallow calling `simple_history.register` twice on the same model
- Added Python 3 support
- Added support for custom user model (Django 1.5+)

3.19 1.2.3 (2013-04-22)

- Fixed packaging bug: added admin template files to PyPI package

3.20 1.2.1 (2013-04-22)

- Added tests
- Added history view/revert feature in admin interface
- Various fixes and improvements

3.21 Oct 22, 2010

- Merged setup.py from Klaas van Schelven - Thanks!

3.22 Feb 21, 2010

- Initial project creation, with changes to support ForeignKey relations.