# django-simple-history Documentation

***Release 2.7.1***

**Corey Bertram**

**Apr 16, 2019**

# Contents

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

This app supports the following combinations of Django and Python:

| Django | Python |
|---|---|
| 1.11 | 2.7, 3.4, 3.5, 3.6, 3.7 |
| 2.0 | 3.4, 3.5, 3.6, 3.7 |
| 2.1 | 3.5, 3.6, 3.7 |

CHAPTER 1

# Contribute

- Issue Tracker: https://github.com/treyhunner/django-simple-history/issues
- Source Code: https://github.com/treyhunner/django-simple-history

Pull requests are welcome.

CHAPTER 2

# Documentation

## 2.1 Quick Start

### 2.1.1 Install

Install from PyPI with `pip`:

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

### 2.1.2 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 `HistoryRequestMiddleware` 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 *User Tracking*.

#### Track History

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.

#### Track History for a Third-Party Model

To track history for a model you didn't create, use the `simple_history.register` function. 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)
```

If you want to separate the migrations of the historical model into an app other than the third-party model's app, you can set the `app` parameter in `register`. For instance, if you want the migrations to live in the migrations folder of the package you register the model in, you could do:

```
register(User, app=__package__)
```

### 2.1.3 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`.

### 2.1.4 What Now?

By adding `HistoricalRecords` to a model or registering a model using `register`, you automatically start tracking any create, update, or delete that occurs on that model. Now you can *query the history programmatically* and *view the history in Django admin*.

### 2.1.5 What is `django-simple-history` Doing Behind the Scenes?

If you tried the code *above* and ran the migrations on it, you'll see the following tables in your database:

- `app_choice`
- `app_historicalchoice`
- `app_historicalpoll`
- `app_poll`

The two extra tables with `historical` prepended to their names are tables created by `django-simple-history`. These tables store every change that you make to their respective base tables. Every time a create, update, or delete occurs on `Choice` or `Poll` a new row is created in the historical table for that model including all of the fields in the instance of the base model, as well as other metadata:

- `history_user`: the user that made the create/update/delete
- `history_date`: the datetime at which the create/update/delete occurred
- `history_change_reason`: the reason the create/update/delete occurred (null by default)
- `history_id`: the primary key for the historical table (note the the base table's primary key is not unique on the historical table since there are multiple versions of it on the historical table)
- `history_type`: + for create, ~ for update, and − for delete

Now try saving an instance of `Choice` or `Poll`. Check the historical table to see that the history is being tracked.

## 2.2 Querying History

### 2.2.1 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>]
```

### 2.2.2 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!

### 2.2.3 Getting previous and next historical record

If you have a historical record for an instance and would like to retrieve the previous historical record (older) or next historical record (newer), *prev_record* and *next_record* read-only attributes can be used, respectively.

```
>>> from polls.models import Poll, Choice
>>> from datetime import datetime
>>> poll = Poll.objects.create(question="what's up?", pub_date=datetime.now())
>>>
>>> record = poll.history.first()
>>> record.prev_record
None
>>> record.next_record
None
>>> poll.question = "what is up?"
>>> poll.save()
>>> record.next_record
<HistoricalPoll: Poll object as of 2010-10-25 18:04:13.814128>
```

If a historical record is the first record, *prev_record* will be *None*. Similarly, if it is the latest record, *next_record* will be *None*

### 2.2.4 Reverting the Model

`SimpleHistoryAdmin` allows users to revert back to an old version of the model through the admin interface. You can also do this programmatically. To do so, you can take any historical object, and save the associated instance. For example, if we want to access the earliest `HistoricalPoll`, for an instance of `Poll`, we can do:

```
>>> poll.history.earliest()
<HistoricalPoll: Poll object as of 2010-10-25 18:04:13.814128>
```

And to revert to that `HistoricalPoll` instance, we can do:

```
>>> earliest_poll = poll.history.earliest()
>>> earliest_poll.instance.save()
```

This will change the `poll` instance to have the data from the `HistoricalPoll` object and it will create a new row in the `HistoricalPoll` table indicating that a new change has been made.

### 2.2.5 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>
```

### 2.2.6 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>
```

### 2.2.7 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
        return ret


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

### 2.2.8 Filtering data using a relationship to the model

To filter changes to the data, a relationship to the history can be established. For example, all data records in which a particular user was involved.

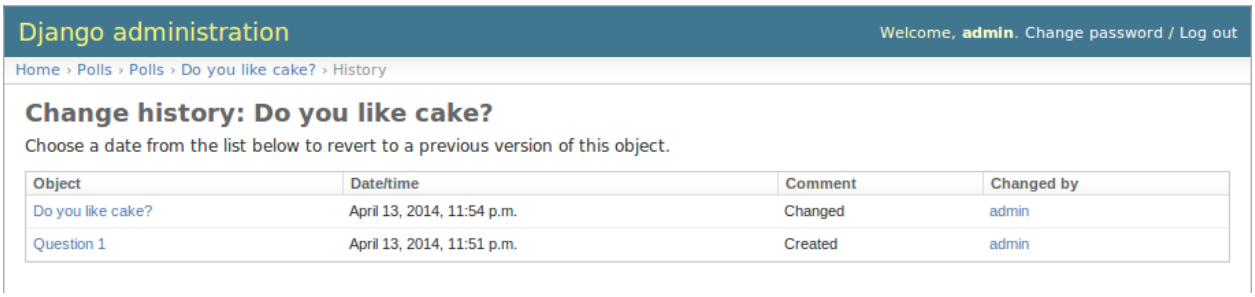
```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    log = HistoricalRecords(related_name='history')


Poll.objects.filter(history__history_user=4)
```

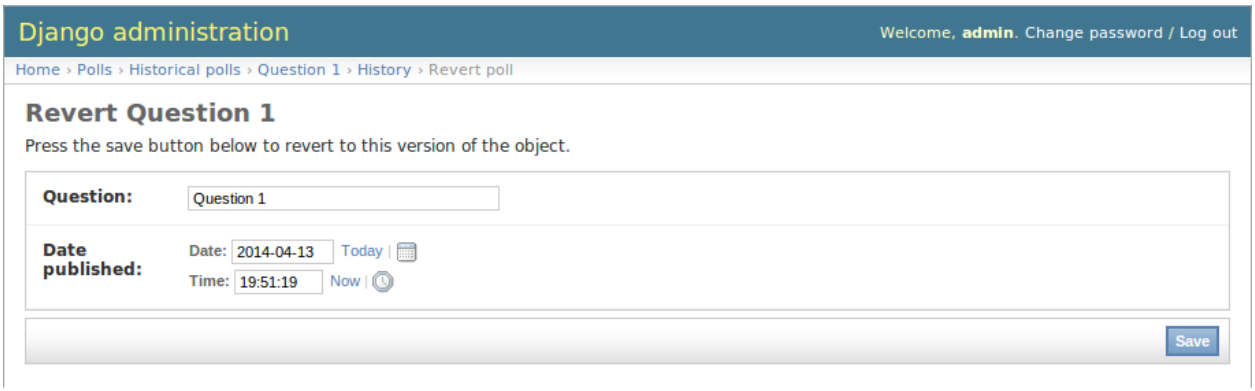
## 2.3 Admin Integration

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.

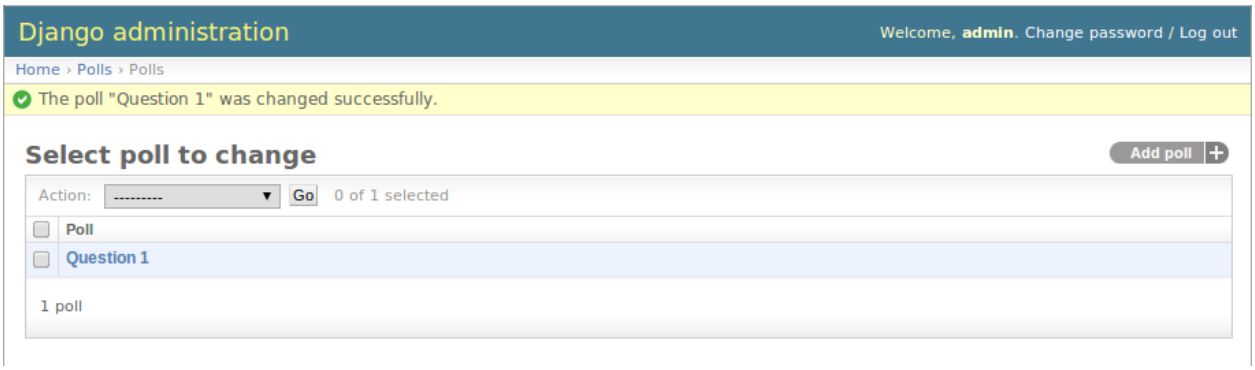
This will replace the history object page on the admin site and allow viewing and reverting to previous model versions. Changes made in admin change forms will also accurately note the user who made the change.



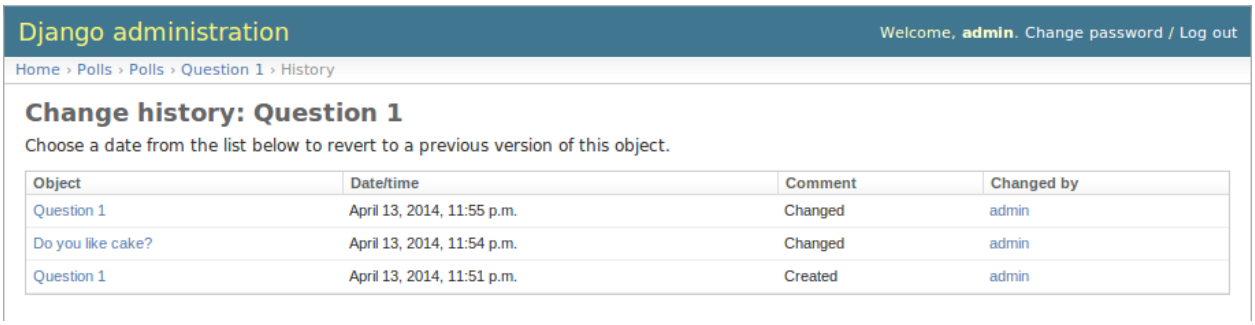

Clicking on an object presents the option to revert to that version of the object.




(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
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 *User Tracking*).

### 2.3.1 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

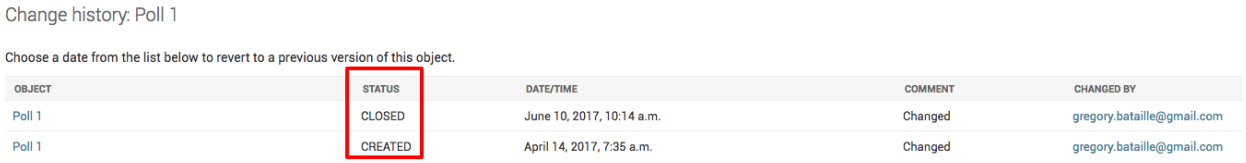
```
from django.contrib import admin
from simple_history.admin import SimpleHistoryAdmin
from .models import Poll, Choice
```

(continues on next page)

(continued from previous page)

```
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)
```




## 2.4 Historical Model Customizations

### 2.4.1 Custom `history_id`

By default, the historical table of a model will use an `AutoField` for the table's `history_id` (the history table's primary key). However, you can specify a different type of field for `history_id` by passing a different field to `history_id_field` parameter.

The example below uses a `UUIDField` instead of an `AutoField`:

```
import uuid
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(
        history_id_field=models.UUIDField(default=uuid.uuid4)
    )
```

Since using a `UUIDField` for the `history_id` is a common use case, there is a `SIMPLE_HISTORY_HISTORY_ID_USE_UUID` setting that will set all `history_id``s to UUIDs. Set this with the following line in your ``settings.py` file:

```
SIMPLE_HISTORY_HISTORY_ID_USE_UUID = True
```

This setting can still be overridden using the `history_id_field` parameter on a per model basis.

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

Note: regardless of what field type you specify as your history_id field, that field will automatically set `primary_key=True` and `editable=False`.

### 2.4.2 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

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

### 2.4.3 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')
```

### 2.4.4 Custom model name

By default, historical model is named as 'Historical' + model name. For example, historical records for `Choice` is called `HistoricalChoice`. Users can specify a custom model name via the constructor on

`HistoricalRecords`. The common use case for this is avoiding naming conflict if the user already defined a model named as 'Historical' + model name.

This feature provides the ability to override the default model name used for the generated history model.

To configure history models to use a different name for the history model class, use an option named `custom_model_name`. The value for this option can be a *string* or a *callable*. A simple string replaces the default name of *'Historical' + model name* with the defined string. The most simple use case is illustrated below using a simple string:

```
class ModelNameExample(models.Model):
    history = HistoricalRecords(
        custom_model_name='SimpleHistoricalModelNameExample'
    )
```

If you are using a base class for your models and want to apply a name change for the historical model for all models using the base class then a callable can be used. The callable is passed the name of the model for which the history model will be created. As an example using the callable mechanism, the below changes the default prefix *Historical* to *Audit*:

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    history = HistoricalRecords(custom_model_name=lambda x:f'Audit{x}')

class Opinion(models.Model):
    opinion = models.CharField(max_length=2000)

register(Opinion, custom_model_name=lambda x:f'Audit{x}')
```

The resulting history class names would be *AuditPoll* and *AuditOpinion*. If the app the models are defined in is *yoda* then the corresponding history table names would be *yoda_auditpoll* and *yoda_auditopinion*

**IMPORTANT: Setting *custom_model_name* to *lambda x:f'{x}'* is not permitted.** An error will be generated and no history model created if they are the same.

### 2.4.5 TextField as *history_change_reason*

The `HistoricalRecords` object can be customized to accept a `TextField` model field for saving the *history_change_reason* either through settings or via the constructor on the model. The common use case for this is for supporting larger model change histories to support changelog-like features.

```
SIMPLE_HISTORY_HISTORY_CHANGE_REASON_USE_TEXT_FIELD=True
```

or

```
class TextFieldExample(models.Model):
    greeting = models.CharField(max_length=100)
    history = HistoricalRecords(
        history_change_reason_field=models.TextField(null=True)
    )
```

### 2.4.6 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])
```

### 2.4.7 Excluded Fields

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.

### 2.4.8 Adding additional fields to historical models

Sometimes it is useful to be able to add additional fields to historical models that do not exist on the source model. This is possible by combining the `bases` functionality with the `pre_create_historical_record` signal.

```
# in models.py
class IPAddressHistoricalModel(models.Model):
    """
    Abstract model for history models tracking the IP address.
    """
    ip_address = models.GenericIPAddressField(_('IP address'))

    class Meta:
        abstract = True


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

    history = HistoricalRecords(bases=[IPAddressHistoricalModel,]
```

```
# define your signal handler/callback anywhere outside of models.py
def add_history_ip_address(sender, **kwargs):
    history_instance = kwargs['history_instance']
    # thread.request for use only when the simple_history middleware is on and enabled
    history_instance.ip_address = HistoricalRecords.thread.request.META['REMOTE_ADDR']
```

```
# in apps.py
class TestsConfig(AppConfig):
    def ready(self):
        from simple_history.tests.models \
            import HistoricalPollWithExtraFields

        pre_create_historical_record.connect(
            add_history_ip_address,
            sender=HistoricalPollWithExtraFields
        )
```

More information on signals in `django-simple-history` is available in *Signals*.

### 2.4.9 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 second 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 an instance with an implicit 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')
```

### 2.4.10 Deleting historical record

In some circumstances you may want to delete all the historical records when the master record is deleted. This can be accomplished by setting `cascade_delete_history=True`.

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

### 2.4.11 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.

### 2.4.12 History Model In Different App

By default the app_label for the history model is the same as the base model. In some circumstances you may want to have the history models belong in a different app. This will support creating history models in a different database to the base model using database routing functionality based on app_label. To configure history models in a different app, add this to the HistoricalRecords instantiation or the record invocation: `app="SomeAppName"`.

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

class Opinion(models.Model):
    opinion = models.CharField(max_length=2000)

register(Opinion, app="SomeAppName")
```

## 2.5 User Tracking

### 2.5.1 Recording Which User Changed a Model

There are four documented ways to attach users to a tracked change:

1. Use the `HistoryRequestMiddleware`. The middleware sets the User instance that made the request as the `history_user` on the history table.

2. Use `simple_history.admin.SimpleHistoryAdmin`. Under the hood, `SimpleHistoryAdmin` actually sets the `_history_user` on the object to attach the user to the tracked change by overriding the *save_model* method.

3. Assign a user to the `_history_user` attribute of the object as described in the *_history_user section*.

4. Track the user using an explicit `history_user_id`, which is described in *Manually Track User Model*. This method is particularly useful when using multiple databases (where your user model lives in a separate database to your historical model), or when using a user that doesn't live within the Django app (i.e. a user model retrieved from an API).

### Using `_history_user` to Record 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 *Admin Integration*).

Another option for identifying the change user is by providing a function via `get_user`. If provided it will be called everytime that the `history_user` needs to be identified with the following key word arguments:

- `instance`: The current instance being modified
- `request`: If using the middleware the current request object will be provided if they are authenticated.

This is very helpful when using `register`:

```
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')
```

(continues on next page)

(continued from previous page)

```
def get_poll_user(instance, **kwargs):
    return instance.changed_by

register(Poll, get_user=get_poll_user)
```

#### Manually Track User Model

Although `django-simple-history` tracks the `history_user` (the user who changed the model) using a django foreign key, there are instances where we might want to track this user but cannot use a Django foreign key.

**Note:** If you want to track a custom user model that is still accessible through a Django foreign key, refer to *Change User Model*.

The two most common cases where this feature will be helpful are:

1. You are working on a Django app with multiple databases, and your history table is in a separate database from the user table.
2. The user model that you want to use for `history_user` does not live within the Django app, but is only accessible elsewhere (i.e. through an API call).

There are three parameters to `HistoricalRecords` or `register` that facilitate the ability to manually track a `history_user`.

**history_user_id_field** An instance of field (i.e. `IntegerField(null=True)` or `UUIDField(default=uuid.uuid4, null=True)` that will uniquely identify your user object. This is generally the field type of the primary key on your user object.

**history_user_getter** *optional*. A callable that takes the historical instance of the model and returns the `history_user` object. The default getter is shown below:

```
def _history_user_getter(historical_instance):
    if historical_instance.history_user_id is None:
        return None
    User = get_user_model()
    try:
        return User.objects.get(pk=historical_instance.history_user_id)
    except User.DoesNotExist:
        return None
```

**history_user_setter** *optional*. A callable that takes the historical instance and the user instance, and sets `history_user_id` on the historical instance. The default setter is shown below:

```
def _history_user_setter(historical_instance, user):
    if user is not None:
        historical_instance.history_user_id = user.pk
```

### 2.5.2 Change User Model

If you need to use a different user model then `settings.AUTH_USER_MODEL`, pass in the required model to `user_model`. Doing this requires `_history_user` or `get_user` is provided as detailed above.

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

(continues on next page)

(continued from previous page)

```
class PollUser(models.Model):
    user_id = models.ForeignKey('auth.User')


# Only PollUsers should be modifying a Poll
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    changed_by = models.ForeignKey(PollUser)
    history = HistoricalRecords(user_model=PollUser)

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

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

## 2.6 Signals

*django-simple-history* includes signals that help you provide custom behavior when saving a historical record. Arguments passed to the signals include the following:

**instance** The source model instance being saved

**history_instance** The corresponding history record

**history_date** Datetime of the history record's creation

**history_change_reason** Freetext description of the reason for the change

**history_user** The user that instigated the change

**using** The database alias being used

To connect the signals to your callbacks, you can use the `@receiver` decorator:

```
from django.dispatch import receiver
from simple_history.signals import (
    pre_create_historical_record,
    post_create_historical_record
)

@receiver(pre_create_historical_record)
def pre_create_historical_record_callback(sender, **kwargs):
    print("Sent before saving historical record")

@receiver(post_create_historical_record)
def post_create_historical_record_callback(sender, **kwargs):
    print("Sent after saving historical record")
```

## 2.7 History Diffing

When you have two instances of the same `HistoricalRecord` (such as the `HistoricalPoll` example above), you can perform diffs to see what changed. This will result in a `ModelDelta` containing the following properties:

1. A list with each field changed between the two historical records
2. A list with the names of all fields that incurred changes from one record to the other
3. the old and new records.

This may be useful when you want to construct timelines and need to get only the model modifications.

```
p = Poll.objects.create(question="what's up?")
p.question = "what's up, man?"
p.save()

new_record, old_record = p.history.all()
delta = new_record.diff_against(old_record)
for change in delta.changes:
    print("{} changed from {} to {}".format(change.field, change.old, change.new))
```

## 2.8 Multiple databases

### 2.8.1 Interacting with Multiple Databases

*django-simple-history* follows the Django conventions for interacting with multiple databases.

```
>>> # This will create a new historical record on the 'other' database.
>>> poll = Poll.objects.using('other').create(question='Question 1')

>>> # This will also create a new historical record on the 'other' database.
>>> poll.save(using='other')
```

When interacting with `QuerySets`, use `using()`:

```
>>> # This will return a QuerySet from the 'other' database.
Poll.history.using('other').all()
```

When interacting with manager methods, use `db_manager()`:

```
>>> # This will call a manager method on the 'other' database.
>>> poll.history.db_manager('other').as_of(datetime(2010, 10, 25, 18, 4, 0))
```

See the Django documentation for more information on how to interact with multiple databases.

### 2.8.2 Tracking User in a Separate Database

When using `django-simple-history` in app with multiple database, you may run into an issue where you want to track the history on a table that lives in a separate database to your user model. Since Django does not support cross-database relations, you will have to manually track the `history_user` using an explicit ID. The full documentation on this feature is in *Manually Track User Model*.

### 2.8.3 Tracking History Separate from the Base Model

You can choose whether or not to track models' history in the same database by setting the flag *use_base_model_db*.

‘‘‘ class MyModel(models.Model):

    ... history = HistoricalRecords(use_base_model_db=False)

‘‘‘

If set to *True*, migrations and audit events will be sent to the same database as the base model. If *False*, they will be sent to the place specified by the database router. The default value is *False*.

## 2.9 Utils

### 2.9.1 clean_duplicate_history

For performance reasons, `django-simple-history` always creates an `HistoricalRecord` when `Model.save()` is called regardless of data having actually changed. If you find yourself with a lot of history duplicates you can schedule the `clean_duplicate_history` command

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

You can use `--auto` to clean up duplicates for every model with `HistoricalRecords` or enumerate specific models as args. There is also `-m/--minutes` to specify how many minutes to go back in history while searching (default checks whole history), so you can schedule, for instance, an hourly cronjob such as

```
$ python manage.py clean_duplicate_history -m 60 --auto
```

## 2.10 Common Issues

### 2.10.1 Bulk Creating and Queryset Updating

`django-simple-history` functions by saving history using a `post_save` signal every time that an object with history is saved. However, for certain bulk operations, such as bulk_create and queryset updates, signals are not sent, and the history is not saved automatically. However, `django-simple-history` provides utility functions to work around this.

#### Bulk Creating a Model with History

As of `django-simple-history` 2.2.0, we can use the utility function `bulk_create_with_history` in order to bulk create objects while saving their history:

```
>>> from simple_history.utils import bulk_create_with_history
>>> from simple_history.tests.models import Poll
>>> from django.utils.timezone import now
>>>
>>> data = [Poll(id=x, question='Question ' + str(x), pub_date=now()) for x in␣
↪range(1000)]
>>> objs = bulk_create_with_history(data, Poll, batch_size=500)
>>> Poll.objects.count()
```

(continues on next page)

(continued from previous page)

```
1000
>>> Poll.history.count()
1000
```

If you want to specify a change reason for each record in the bulk create, you can add *changeReason* on each instance:

```
>>> for poll in data:
        poll.changeReason = 'reason'
>>> objs = bulk_create_with_history(data, Poll, batch_size=500)
>>> Poll.history.get(id=data[0].id).history_change_reason
'reason'
```

#### QuerySet Updates with History

Unlike with `bulk_create`, queryset updates perform an SQL update query on the queryset, and never return the actual updated objects (which would be necessary for the inserts into the historical table). Thus, we tell you that queryset updates will not save history (since no `post_save` signal is sent). As the Django documentation says:

```
If you want to update a bunch of records for a model that has a custom
``save()`` method, loop over them and call ``save()``, like this:
```

```
for e in Entry.objects.filter(pub_date__year=2010):
    e.comments_on = False
    e.save()
```

### 2.10.2 Tracking Custom Users

- `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 *Save without a historical record*.

  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.

### 2.10.3 Using django-webtest with Middleware

When using django-webtest to test your Django project with the django-simple-history middleware, you may run into an error similar to the following:

```
django.db.utils.IntegrityError: (1452, 'Cannot add or update a child row: a foreign␣
↪key constraint fails (`test_env`.`core_historicaladdress`, CONSTRAINT `core_
↪historicaladdress_history_user_id_0f2bed02_fk_user_user_id` FOREIGN KEY (`history_
↪user_id`) REFERENCES `user_user` (`id`))')
```

This error occurs because `django-webtest` sets `DEBUG_PROPAGATE_EXCEPTIONS` to true preventing the middleware from cleaning up the request. To solve this issue, add the following code to any `clean_environment` or `tearDown` method that you use:

```
from simple_history.middleware import HistoricalRecords
if hasattr(HistoricalRecords.thread, 'request'):
    del HistoricalRecords.thread.request
```

### 2.10.4 Using F() expressions

`F()` expressions, as described here, do not work on models that have history. Simple history inserts a new record in the historical table for any model being updated. However, `F()` expressions are only functional on updates. Thus, when an `F()` expression is used on a model with a history table, the historical model tries to insert using the `F()` expression, and raises a `ValueError`.

### 2.10.5 Reserved Field Names

For each base model that has its history tracked using `django-simple-history`, an associated historical model is created. Thus, if we have:

```
class BaseModel(models.Model):
    history = HistoricalRecords()
```

a Django model called `HistoricalBaseModel` is also created with all of the fields from `BaseModel`, plus a few extra fields and methods that are on all historical models.

Since these fields and methods are on all historical models, any field or method names on a base model that clash with those names will not be on the historical model (and, thus, won't be tracked). The reserved historical field and method names are below:

- `history_id`
- `history_date`
- `history_change_reason`
- `history_type`
- `history_object`
- `history_user`
- `history_user_id`
- `instance`
- `instance_type`
- `next_record`
- `prev_record`
- `revert_url`
- `__str__`

So if we have:

```
class BaseModel(models.Model):
    instance = models.CharField(max_length=255)
    history = HistoricalRecords()
```

the `instance` field will not actually be tracked on the history table because it's in the reserved set of terms.

CHAPTER 3

# Changes

## 3.1 2.7.1 (2019-04-16)

- Added the possibility to create a relation to the original model (gh-536)
- Fix router backward-compatibility issue with 2.7.0 (gh-539, gh-547)
- Fix hardcoded history manager (gh-542)
- Replace deprecated *django.utils.six* with *six* (gh-526)
- Allow *custom_model_name* parameter to be a callable (gh-489)

## 3.2 2.7.0 (2019-01-16)

- * Add support for `using` chained manager method and save/delete keyword argument (gh-507)
- Added management command `clean_duplicate_history` to remove duplicate history entries (gh-483)
- Updated most_recent to work with excluded_fields (gh-477)
- Fixed bug that prevented self-referential foreign key from using `'self'` (gh-513)
- Added ability to track custom user with explicit custom `history_user_id_field` (gh-511)
- Don't resolve relationships for history objects (gh-479)
- Reorganization of docs (gh-510)

* NOTE: This change was not backward compatible for users using routers to write history tables to a separate database from their base tables. This issue is fixed in 2.7.1.

## 3.3 2.6.0 (2018-12-12)

- Add `app` parameter to the constructor of `HistoricalRecords` (gh-486)
- Add `custom_model_name` parameter to the constructor of `HistoricalRecords` (gh-451)
- Fix header on history pages when custom site_header is used (gh-448)
- Modify `pre_create_historical_record` to pass `history_instance` for ease of customization (gh-421)
- Raise warning if `HistoricalRecords(inherit=False)` is in an abstract model (gh-341)
- Ensure custom arguments for fields are included in historical models' fields (gh-431)
- Add german translations (gh-484)
- Add `extra_context` parameter to history_form_view (gh-467)
- Fixed bug that prevented `next_record` and `prev_record` to work with custom manager names (gh-501)

## 3.4 2.5.1 (2018-10-19)

- Add `'+'` as the `history_type` for each instance in `bulk_history_create` (gh-449)
- Add support for `history_change_reason` for each instance in `bulk_history_create` (gh-449)
- Add `history_change_reason` in the history list view under the `Change reason` display name (gh-458)
- Fix bug that caused failures when using a custom user model (gh-459)

## 3.5 2.5.0 (2018-10-18)

- Add ability to cascade delete historical records when master record is deleted (gh-440)
- Added Russian localization (gh-441)

## 3.6 2.4.0 (2018-09-20)

- Add pre and post create_historical_record signals (gh-426)
- Remove support for `django_mongodb_engine` when converting AutoFields (gh-432)
- Add support for Django 2.1 (gh-418)

## 3.7 2.3.0 (2018-07-19)

- Add ability to diff `HistoricalRecords` (gh-244)

## 3.8 2.2.0 (2018-07-02)

- Add ability to specify alternative user_model for tracking (gh-371)
- Add util function `bulk_create_with_history` to allow bulk_create with history saved (gh-412)

## 3.9 2.1.1 (2018-06-15)

- Fixed out-of-memory exception when running populate_history management command (gh-408)
- Fix TypeError on populate_history if excluded_fields are specified (gh-410)

## 3.10 2.1.0 (2018-06-04)

- Add ability to specify custom `history_reason` field (gh-379)
- Add ability to specify custom `history_id` field (gh-368)
- Add HistoricalRecord instance properties `prev_record` and `next_record` (gh-365)
- Can set admin methods as attributes on object history change list template (gh-390)
- Fixed compatibility of >= 2.0 versions with old-style middleware (gh-369)

## 3.11 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)

## 3.12 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)

## 3.13 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)

## 3.14 1.8.2 (2017-01-19)

- Add Polish locale.
- Add Django 1.10 support.

## 3.15 1.8.1 (2016-03-19)

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

## 3.16 1.8.0 (2016-02-02)

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

## 3.17 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)

## 3.18 1.6.3 (2015-07-30)

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

## 3.19 1.6.2 (2015-07-04)

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

## 3.20 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)

## 3.21 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)

## 3.22 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]

## 3.23 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()`

## 3.24 1.5.2 (2014-10-15)

- Additional fix for migrations (gh-128)

## 3.25 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)

## 3.26 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

## 3.27 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)

## 3.28 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+)

## 3.29 1.2.3 (2013-04-22)

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

## 3.30 1.2.1 (2013-04-22)

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

## 3.31 Oct 22, 2010

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

## 3.32 Feb 21, 2010

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

# Index