django-reversion Documentation

Release 1.8.0

Dave Hall

December 11, 2015

Contents

CHAPTER 1

Getting started with django-reversion

To install django-reversion, follow these steps:

1. Install with pip: `pip install django-reversion`.
2. Add `'reversion'` to `INSTALLED_APPS`.
3. Run `manage.py syncdb`.

The latest release (1.8) of django-reversion is designed to work with Django 1.6. If you have installed anything other than the latest version of Django, please check the *compatible Django versions* page before installing django-reversion.

There are a number of alternative methods you can use when installing django-reversion. Please check the *installation methods* page for more information.

CHAPTER 2

Admin integration

django-reversion can be used to add a powerful rollback and recovery facility to your admin site. To enable this, simply register your models with a subclass of `reversion.VersionAdmin`:

```
import reversion

class YourModelAdmin(reversion.VersionAdmin):

    pass

admin.site.register(YourModel, YourModelAdmin)
```

Whenever you register a model with the `VersionAdmin` class, be sure to run the `./manage.py createinitialrevisions` command to populate the version database with an initial set of model data. Depending on the number of rows in your database, this command could take a while to execute.

For more information about admin integration, please read the *admin integration* documentation.

CHAPTER 3

Low Level API

You can use django-reversion's API to build powerful version-controlled views. For more information, please read the *low level API* documentation.

CHAPTER 4

More information

4.1 Installation

4.1.1 Installation methods

Note: It is recommended that you always use the latest release of django-reversion with the latest release of Django. If you are using an older version of Django, then please check out the *Compatible Django Versions* page for more information.

pip

You can install django-reversion into your system, or virtual environment, by running the following command in a terminal:

```
$ pip install django-reversion
```

easy_install

The popular easy_install utility can be used to install the latest django-reversion release from the Python Package Index. Simply run the following command in a terminal:

```
$ sudo easy_install django-reversion
```

Git

Using Git to install django-reversion provides an easy way of upgrading your installation at a later date. Simply clone the public git repository and symlink the `src/reversion` directory into your `PYTHONPATH`:

```
$ git clone git://github.com/etianen/django-reversion.git
$ cd django-reversion.git
$ git checkout release-1.8
$ ln -s src/reversion /your/pythonpath/location/reversion
```

4.1.2 Compatible Django Versions

django-reversion is an actively-maintained project, and aims to stay compatible with the latest version of Django. Unfortunately, this means that the latest release of django-reversion might not work with older versions of Django.

If you are using anything other than the latest release of Django, it is important that you check the table below to ensure that your django-reversion download will be compatible.

Django version	Reversion release
1.6+	1.8
1.5.1+	1.7.1
1.5	1.7
1.4.4+	1.6.6
1.4.3	1.6.5
1.4.2	1.6.4
1.4.1	1.6.3
1.4	1.6.1
1.3.6	1.5.7
1.3.5	1.5.6
1.3.4	1.5.5
1.3.3	1.5.4
1.3.2	1.5.3
1.3.1	1.5.2
1.3	1.5
1.2.5	1.3.3
1.2.4	1.3.3
1.2.3	1.3.2
1.2	1.3
1.1.1	1.2.1
1.1	1.2
1.0.4	1.1.2
1.0.3	1.1.2
1.0.2	1.1.1

Getting the code

All django-reversion releases are available from the project downloads area. You can also use Git to checkout tags from the public git repository.

There are a number of alternative methods you can use when installing django-reversion. Please check the *installation methods* page for more information.

4.1.3 Schema migrations

This page describes the schema migrations that have taken place over the lifetime of django-reversion, along with a how-to guide for updating your schema using South.

django-reversion 1.8

The current working version removes `type` column from `reversion_version` table.

In order to apply this migration using south, simply run:

```
./manage.py migrate reversion
```

django-reversion 1.5

This version adds in significant speedups for models with integer primary keys.

In order to apply this migration using south, simply run:

```
./manage.py migrate reversion
```

If you have a large amount of existing version data, then this command might take a little while to run while the database tables are updated.

django-reversion 1.4

This version added a much-requested 'type' field to Version models, allows statistic to be gathered about the number of additions, changes and deletions that have been applied to a model.

In order to apply this migration, it is first necessary to install South.

1. Add 'south' to your `INSTALLED_APPS` setting.
2. Run `./manage.py syncdb`

You then need to run the following two commands to complete the migration:

```
./manage.py migrate reversion 0001 --fake
./manage.py migrate reversion
```

django-reversion 1.3.3

No migration needed.

4.1.4 Admin integration

django-reversion can be used to add a powerful rollback and recovery facility to your admin site. To enable this, simply register your models with a subclass of `reversion.VersionAdmin`.

```
import reversion

class YourModelAdmin(reversion.VersionAdmin):

    pass

admin.site.register(YourModel, YourModelAdmin)
```

You can also use `reversion.VersionAdmin` as a mixin with another specialized admin class.

```
class YourModelAdmin(reversion.VersionAdmin, YourBaseModelAdmin):

    pass
```

If you're using an existing third party app, then you can add patch django-reversion into its admin class by using the `reversion.helpers.patch_admin()` method. For example, to add version control to the built-in User model:

```
from reversion.helpers import patch_admin

patch_admin(User)
```

Admin customizations

It's possible to customize the way django-reversion integrates with your admin site by specifying options on the subclass of `reversion.VersionAdmin` as follows:

```
class YourModelAdmin(reversion.VersionAdmin):

    option_name = option_value
```

The available admin options are:

- **history_latest_first:** Whether to display the available versions in reverse chronological order on the revert and recover views (default `False`)
- **ignore_duplicate_revisions:** Whether to ignore duplicate revisions when storing version data (default `False`)
- **recover_form_template:** The name of the template to use when rendering the recover form (default `'reversion/recover_form.html'`)
- **reversion_format:** The name of a serialization format to use when storing version data (default `'json'`)
- **revision_form_template:** The name of the template to use when rendering the revert form (default `'reversion/revision_form.html'`)
- **recover_list_template:** The name of the template to use when rendering the recover list view (default `'reversion/recover_list.html'`)

Customizing admin templates

In addition to specifying custom templates using the options above, you can also place specially named templates on your template root to override the default templates on a per-model or per-app basis.

For example, to override the recover_list template for the user model, the auth app, or all registered models, you could create a template with one of the following names:

- `'reversion/auth/user/recover_list.html'`
- `'reversion/auth/recover_list.html'`
- `'reversion/recover_list.html'`

4.2 Further reading

4.2.1 Low-level API

You can use django-reversion's API to build powerful version-controlled views outside of the built-in admin site.

Registering models with django-reversion

If you're already using the *admin integration* for a model, then there's no need to register it. However, if you want to register a model without using the admin integration, then you need to use the `reversion.register()` method.

```
import reversion

reversion.register(YourModel)
```

Warning: If you're using django-reversion in an management command, and are using the automatic `VersionAdmin` registration method, then you'll need to import the relevant `admin.py` file at the top of your management command file.

Warning: When Django starts up, some python scripts get loaded twice, which can cause 'already registered' errors to be thrown. If you place your calls to `reversion.register()` in the `models.py` file, immediately after the model definition, this problem will go away.

Creating revisions

A revision represents one or more changes made to your models, grouped together as a single unit. You create a revision by marking up a section of code to represent a revision. Whenever you call `save()` on a model within the scope of a revision, it will be added to that revision.

Note: If you call `save()` outside of the scope of a revision, a revision is NOT created. This means that you are in control of when to create revisions.

There are several ways to create revisions, as explained below. Although there is nothing stopping you from mixing and matching these approaches, it is recommended that you pick one of the methods and stick with it throughout your project.

reversion.create_revision() decorator

You can decorate any function with the `reversion.create_revision()` decorator. Any changes to your models that occur during this function will be grouped together into a revision.

```
@transaction.atomic()
@reversion.create_revision()
def you_view_func(request):
    your_model.save()
```

reversion.create_revision() context manager

You can use a context manager to mark up a block of code. Once the block terminates, any changes made to your models will be grouped together into a revision.

```
with transaction.atomic(), reversion.create_revision():
    your_model.save()
```

RevisionMiddleware

The simplest way to create revisions is to use `reversion.middleware.RevisionMiddleware`. This will automatically wrap every request in a revision, ensuring that all changes to your models will be added to their version history.

To enable the revision middleware, simply add it to your `MIDDLEWARE_CLASSES` setting as follows:

```
MIDDLEWARE_CLASSES = (
    'reversion.middleware.RevisionMiddleware',
    # Other middleware goes here...
)
```

Warning: Due to changes in the Django 1.6 transaction handling, revision data will be saved in a separate database transaction to the one used to save your models, even if you set `ATOMIC_REQUESTS = True`. If you need to ensure that your models and revisions are saved in the save transaction, please use the `reversion.create_revision()` context manager or decorator in combination with `transaction.atomic()`.

Version meta data

It is possible to attach a comment and a user reference to an active revision using the following method:

```
with transaction.atomic(), reversion.create_revision():
    your_model.save()
    reversion.set_user(user)
    reversion.set_comment("Comment text...")
```

If you use `RevisionMiddleware`, then the user will automatically be added to the revision from the incoming request.

Custom meta data

You can attach custom meta data to a revision by creating a separate django model to hold the additional fields. For example:

```
from reversion.models import Revision

class VersionRating(models.Model):
    revision = models.OneToOneField(Revision)  # This is required
    rating = models.PositiveIntegerField()
```

You can then attach this meta class to a revision using the following method:

```
reversion.add_meta(VersionRating, rating=5)
```

Reverting to previous revisions

To revert a model to a previous version, use the following method:

```
your_model = YourModel.objects.get(pk=1)

# Build a list of all previous versions, latest versions first:
version_list = reversion.get_for_object(your_model)

# Build a list of all previous versions, latest versions first, duplicates removed:
version_list = reversion.get_unique_for_object(your_model)

# Find the most recent version for a given date:
version = reversion.get_for_date(your_model, datetime.datetime(2008, 7, 10))

# Access the model data stored within the version:
version_data = version.field_dict

# Revert all objects in this revision:
version.revision.revert()

# Revert all objects in this revision, deleting related objects that have been created since the rev
```

```
version.revision.revert(delete=True)

# Just revert this object, leaving the rest of the revision unchanged:
version.revert()
```

Recovering Deleted Objects

To recover a deleted object, use the following method:

```
# Built a list of all deleted objects, latest deletions first.
deleted_list = reversion.get_deleted(YourModel)

# Access a specific deleted object.
delete_version = deleted_list.get(id=5)

# Recover all objects in this revision:
deleted_version.revision.revert()

# Just recover this object, leaving the rest of the revision unchanged:
deleted_version.revert()
```

Advanced model registration

Following foreign key relationships

Normally, when you save a model it will only save the primary key of any ForeignKey or ManyToMany fields. If you also wish to include the data of the foreign key in your revisions, pass a list of relationship names to the `reversion.register()` method.

```
reversion.register(YourModel, follow=["your_foreign_key_field"])
```

Please note: If you use the follow parameter, you must also ensure that the related model has been registered with django-reversion.

In addition to ForeignKey and ManyToMany relationships, you can also specify related names of one-to-many relationships in the follow clause. For example, given the following database models:

```
class Person(models.Model):
    pass

class Pet(models.Model):
    person = models.ForeignKey(Person)

reversion.register(Person, follow=["pet_set"])
reversion.register(Pet)
```

Now whenever you save a revision containing a `Person`, all related `Pet` instances will be automatically saved to the same revision.

Multi-table inheritance

By default, django-reversion will not save data in any parent classes of a model that uses multi-table inheritance. If you wish to also add parent models to your revision, you must explicitly add them to the follow clause when you register the model.

For example:

```
class Place(models.Model):
    pass

class Restaurant(Place):
    pass

reversion.register(Place)
reversion.register(Restaurant, follow=["place_ptr"])
```

Saving a subset of fields

If you only want a subset of fields to be saved to a revision, you can specify a `fields` or `exclude` argument to the `reversion.register()` method.

```
reversion.register(YourModel, fields=["pk", "foo", "bar"])
reversion.register(YourModel, exclude=["foo"])
```

Please note: If you are not careful, then it is possible to specify a combination of fields that will make the model impossible to recover. As such, approach this option with caution.

Custom serialization format

By default, django-reversion will serialize model data using the `'json'` serialization format. You can override this on a per-model basis using the format argument to the register method.

```
reversion.register(YourModel, format="yaml")
```

Please note: The named serializer must serialize model data to a utf-8 encoded character string. Please verify that your serializer is compatible before using it with django-reversion.

Really advanced registration

It's possible to customize almost every aspect of model registration by registering your model with a subclass of `reversion.VersionAdapter`. Behind the scenes, `reversion.register()` does this anyway, but you can explicitly provide your own VersionAdapter if you need to perform really advanced customization.

```
class MyVersionAdapter(reversion.VersionAdapter):
    pass  # Please see the reversion source code for available methods to override.

reversion.register(MyModel, adapter_cls=MyVersionAdapter)
```

Automatic Registration by the Admin Interface

As mentioned at the start of this page, the admin interface will automatically register any models that use the `VersionAdmin` class. The admin interface will automatically follow any InlineAdmin relationships, as well as any parent links for models that use multi-table inheritance.

For example:

```
# models.py

class Place(models.Model):
    pass

class Restaurant(Place):
    pass

class Meal(models.Model):
    restaurant = models.ForeignKey(Restaurant)

# admin.py

class MealInlineAdmin(admin.StackedInline):
    model = Meal

class RestaurantAdmin(VersionAdmin):
    inlines = MealInlineAdmin,

admin.site.register(Restaurant, RestaurantAdmin)
```

Since `Restaurant` has been registered with a subclass of `VersionAdmin`, the following registration calls will be made automatically:

```
reversion.register(Place)
reversion.register(Restaurant, follow=("place_ptr", "meal_set"))
reversion.register(Meal)
```

It is only necessary to manually register these models if you wish to override the default registration parameters. In most cases, however, the defaults will suit just fine.

4.2.2 Management commands

django-reversion comes with a number of additional django-admin.py management commands, detailed below.

createinitialrevisions

This command is used to create a single, base revision for all registered models in your project. It should be run after installing django-reversion. If your project contains a lot of version-controlled data, then this might take a while to complete.

```
django-admin.py createinitialrevisions
django-admin.py createinitialrevisions someapp
django-admin.py createinitialrevisions someapp.SomeModel
```

4.2.3 Signals sent by django-reversion

django-reversion provides a number of custom signals that can be used to tie-in additional functionality to the version creation mechanism.

Important: Don't connect to the pre_save or post_save signals of the Version or Revision models directly, use the signals outlined below instead. The pre_save and post_save signals are no longer sent by the Version or Revision models since django-reversion 1.7.

reversion.pre_revision_commit

This signal is triggered just before a revision is saved to the database. It receives the following keyword arguments:

- **instances** - A list of the model instances in the revision.
- **revision** - The unsaved Revision model.
- **versions** - The unsaved Version models in the revision.

reversion.post_revision_commit

This signal is triggered just after a revision is saved to the database. It receives the following keyword arguments:

- **instances** - A list of the model instances in the revision.
- **revision** - The saved Revision model.
- **versions** - The saved Version models in the revision.

Connecting to signals

The signals listed above are sent only once *per revision*, rather than once *per model in the revision*. In practice, this means that you should connect to the signals without specifying a *sender*, as below:

```
def on_revision_commit(**kwargs):
    pass  # Your signal handler code here.
reversion.post_revision_commit.connect(on_revision_commit)
```

To execute code only when a revision has been saved for a particular Model, you should inspect the contents of the *instances* parameter, as below:

```
def on_revision_commit(instances, **kwargs):
    for instance in instances:
        if isinstance(instance, MyModel):
            pass  # Your signal handler code here.
reversion.post_revision_commit.connect(on_revision_commit)
```

4.2.4 How it works

Saving Revisions

Enabling version control for a model is achieved using the `reversion.register` method. This registers the version control machinery with the `post_save` signal for that model, allowing new changes to the model to be caught.

```
import reversion

reversion.register(YourModel)
```

Any models that use subclasses of `VersionAdmin` in the admin interface will be automatically registered with django-reversion. As such, it is only necessary to manually register these models if you wish to override the default registration settings.

Whenever you save changes to a model, it is serialized using the Django serialization framework into a JSON string. This is saved to the database as a `reversion.models.Version` model. Each `Version` model is linked to a model instance using a `GenericForeignKey`.

Foreign keys and many-to-many relationships are normally saved as their primary keys only. However, the `reversion.register` method takes an optional follow clause allowing these relationships to be automatically added to revisions. Please see *Low Level API* for more information.

Reverting Versions

Reverting a version is simply a matter of loading the appropriate `Version` model from the database, deserializing the model data, and re-saving the old data.

There are a number of utility methods present on the `Version` object manager to assist this process. Please see *Low Level API* for more information.

Revision Management

Related changes to models are grouped together in revisions. This allows for atomic rollback from one revision to another. You can automate revision management using either `reversion.middleware.RevisionMiddleware`, or the `reversion.revision.create_on_success decorator`.

For more information on creating revisions, please see *Low Level API*.

Admin Integration

Full admin integration is achieved using the `reversion.admin.VersionAdmin` class. This will create a new revision whenever a model is edited using the admin interface. Any models registered for version control, including inline models, will be included in this revision.

The `object_history` view is extended to make each `LogEntry` a link that can be used to revert the model back to the most recent version at the time the `LogEntry` was created.

Choosing to revert a model will display the standard model change form. The fields in this form are populated using the data contained in the revision corresponding to the chosen `LogEntry`. Saving this form will result in a new revision being created containing the new model data.

For most projects, simply registering a model with a subclass of `VersionAdmin` is enough to satisfy all its version-control needs.

4.2.5 Generating Diffs

A common problem when dealing with version-controlled text is generating diffs to highlight changes between different versions.

django-reversion comes with a number of helper functions that make generating diffs easy. They all rely on the google-diff-match-patch library, so make sure you have this installed before trying to use the functions.

Low-Level API

It is possible to generate two types of diff using the diff helper functions. For the purpose of these examples, it is assumed that you have created a model called `Page`, which contains a text field called `content`.

First of all, you need to use the *low level API* to retrieve the versions you want to compare.

```
from reversion.helpers import generate_patch

# Get the page object to generate diffs for.
page = Page.objects.all()[0]

# Get the two versions to compare.
available_versions = reversion.get_for_object(page)

old_version = available_versions[0]
new_version = available_versions[1]
```

Now, in order to generate a text patch:

```
from reversion.helpers import generate_patch

patch = generate_patch(old_version, new_version, "content")
```

Or, to generate a pretty HTML patch:

```
from reversion.helpers import generate_patch_html

patch_html = generate_patch_html(old_version, new_version, "content")
```

Because text diffs can often be fragmented and hard to read, an optional `cleanup` parameter may be passed to generate friendlier diffs.

```
patch_html = generate_patch_html(old_version, new_version, "content", cleanup="semantic")
patch_html = generate_patch_html(old_version, new_version, "content", cleanup="efficiency")
```

Of the two cleanup styles, the one that generally produces the best result is 'semantic'.

Admin Integration

The admin integration for django-reversion does not currently support diff generation. This is a deliberate design decision, as it would make the framework a lot more heavyweight, as well as carrying the risk of confusing non-technical end users.

While future versions may support a more advanced admin class, for the time being it is left up to your own imagination for ways in which to integrate diffs with your project.