django-versioning Documentation

Release 0.5

Stijn Debrouwere

February 10, 2017

Contents

`django-revisions` is a Django app that allows you to keep a version history for model instances. It has a simple API, but is also integrated with the Django admin interface. `django-revisions` doesn't add any tables to your database, nor does it work by serializing old revisions – making this app very natural to work with and migration-friendly, as opposed to other solutions out there. (See The basic design of django-revisions.)

- Access and revert to any previous model save with a convenient and minimally intrusive API.
- An optional trash bin for deleted content.
- Admin integration: restore trash, revert content to an older revision.
- Works flawlessly with migration tools like South

`django-revisions` takes care of model instance version history. If you found this page while looking for asset versioning of media files, like javascript or CSS, take a look at `django-css` and related apps instead.

CHAPTER 1

Getting Started

1.1 Installation

1.1.1 Models

django-revisions works by adding *models.VersionedModel* as a base class to your model as well as — if you prefer — *shortcuts.VersionedModel*. You can enable a trash function by adding *models.TrashableModel* as a base class to your model *and* adding the class decorator *decorators.trash_aware* to any model for which you want to enable trash (that is, soft deletes).

All three classes are independent, so you'll have to add them in separately.

The models and API work with both single-table models and joined tables (that is, concrete inheritance).

django-revisions makes no effort to be a drop-in for existing models. It adds fields to your models, which means you'll have to run a South migration to get you started, and you'll have to run a small script to generate bundle IDs (they can just equal the object PK).

See Caveats when working with VersionedModel for more detail.

1.1.2 Specifying your own primary key

- working with UUIDs; comparators – e.g. a "created" date (but beware of programmatical creation, resulting in multiple objects with the same created date and thus no canonical "latest" revision)

1.1.3 Admin integration

1.1.4 Deleting and trashing versioned content

This application also includes a simple abstract model that will put deleted objects into a **trash bin**, rather than outright deleting them from the database. `TrashableModel` works with any model, versioned or not. It adds a single `is_trash` field to the database table, so make sure to add that in manually or remember to execute a migration.

Note that, for design reasons, you can't trash individual revisions. If you want to undo a revision, `obj.revert_to(obj.get_revisions().prev)` or `obj.get_revisions().prev.make_current_revision()` are the preferred methods. That way, the version history is kept intact.

Hard deleting indidual revisions is possible for administration purposes, using `obj.delete_revision()`, but is highly discouraged.

1.2 API

django-revisions is still sorely lacking in documentation, though early adopters can get started by browsing through the methods available on *models.TrashableModel*, *models.VersionedModel* and the shortcuts in *shortcuts.VersionedModel*.

1.2.1 Some examples

```
# models.py
from django.db import models
import revisions
class Story(revisions.models.VersionedModel, revisions.shortcuts.VersionedModel):
    title = models.CharField(max_length=200)
    date = models.DateTimeField(auto_now=True)
    log = models.CharField(max_length=200)

    def __unicode__(self):
        return self.title

    class Versioning:
        publication_date = 'date'
        clear_each_revision = ['log']

# interactive session
>>> story = Story(title="a story (v1)")
>>> story.save()
>>> story.pk
1
>>> story.title = "a story (v2)"
>>> story.save()
>>> story.pk
2
>>> story.get_revisions()
[<Story: a story (v1)>, <Story: a story (v2)>]
>>> story.get_revisions()[0].make_current_revision()
>>> story.pk
3
>>> story.get_revisions()
[<Story: a story (v1)>, <Story: a story (v2)>, <Story: a story (v1)>]
>>> old_story = story.get_revisions()[0]
>>> # revert to a story
>>> story.revert_to(old_story)
>>> # or a primary key
>>> story.revert_to(2)
>>> # dates work as well
>>> story.revert_to(old_story.date)
>>> story.log = "Changed some stuff"
>>> story.save()
>>> # the Django admin clears out fields that have to be empty on each rev for you:
>>> story.prepare_for_writing()
>>> story.log
''
```

1.2.2 Methods and attributes

Shortcuts

1.2.3 Learn more

The basic design of django-revisions

Ways of versioning

`django-revisions` works by linking different revisions of the same content together using a shared id, while identifying each individual revision using a version id (vid). The version id serves as the primary key. This is the same approach as used by content management systems like Wordpress and Drupal.

There are a couple of different architectures in use that each do versioning in a different way.

- MessageDB is a dedicated versioned data store.
- A couple of systems use a version control system as the back-end for versioning, like django-rcsfield or the Jekyll blog system.
- django-reversion stores history in a serialized format (e.g. pickled, perhaps as a delta) on the model itself. This is the most popular choice right now.
- django-modelhistory (inactive), django-history (inactive), django-history-tables (inactive), AuditTrail (inactive), Audit (inactive), django-versioning all store model history in a different table from the model data itself – often but not always pickled.
- This app stores history is in the same database, in the same table(s), and unpickled: any version including the most recent one is stored in an identical format, plainly viewable in the database.

Advantages to same-table versioning While storing old versions as serialized objects is easiest to implement, there are definite advantages to same-table same-format versioning.

- Any version is equal: old versions are easily queryable both from Django or from any other piece of software in any language, using plain SQL. Serialized data does not provide for easy querying, which makes certain use-cases tough or resource-heavy, like redirecting users who visit old content (perhaps with a different slug) to the latest version, or querying the database from systems other than Django.
- Migrations are a helluvalot easier with same-table versioning than with a versioning system that relies on serializing, as most do. Unless you write custom migration scripts, serialized data is very prone to schema conflicts when you update model definitions.

While using a version control system to keep version history might initially cater to our geek sensibilities, it comes with definite drawbacks.

- Everything in the same database, on the same model means one less thing to worry about: VCS-based systems leave you to figure out a backup strategy for yet another persistence layer, as a good ol' database dump would not contain any history. In `django-revisions`, all history lives in your database.
- While people may work on the same code simultaneously, people never write or edit stories simultaneously (and indeed you may want to prevent this from happening, using e.g. django-locking), rendering all the stuff that we love so much in version control systems (like branching, forking, merging and easily fixing conflicts) pointless. It's overkill.

Reading

http://code.djangoproject.com/wiki/FullHistory http://blog.brunogola.com.br/2009/10/django-model-history-with-django-reversion/ http://lethain.com/entry/2008/oct/15/choosing-between-audittrail-and-django-rcsfield/

Caveats when working with `VersionedModel`

References to bundles or specific versions

A foreign key to a versioned object will always point to a specific version and not the bundle as a whole. Sometimes, however, it's useful to be able to have a reference to the *bundle* and not a specific version — every revision of "client" that's tied to a "project", say.

Provided you're accessing a piece of versioned content through a reference from another model, you can get the latest revision of that reference with the `get_latest_revision` method.

The other way around is easy too. Say you have an Author model that refers to a versioned Story model. On instances, you can simply use `story.related_author_set` (instead of `story.author_set`) to access all authors across versions and regardless of which specific version or versions an author is linked to.

A side note: why you shouldn't use Django's to_field to reference content bundles

In Django 1.0 you used to be able to abuse foreign keys to allow for pseudo-foreign key references to a *bundle* instead of a specific version.

class Instructions(models.Model): # dit om een model te testen gerelateerd aan de bundel story = models.ForeignKey(Story, to_field='id')

Because `VersionedModel.latest` is the default manager for versioned content, such a foreign key attribute would then return the latest revision of the bundle even though the bundle id field is shared among revisions.

However, in Django 1.2, this no longer works, because foreign keys should by their very nature only reference unique fields. See http://groups.google.com/group/django-users/browse_thread/thread/fcd3915a19ae333e and http://code.djangoproject.com/ticket/11702 for more information.

Adding your own managers

If you add your own managers to an object, make sure to add revisions.managers.LatestManager() back in, preferably as the first and thus default manager. You'll probably also want to add django.db.managers.Manager() back in, as *objects*.

1.3 Development: reporting bugs, helping out, running the test suite

Development takes place on GitHub. Feel free to fork, and please report any bugs or feature requests over there. Run the test suite simply by adding `revisions` and `revisions.tests` to your apps, and subsequently running `python manage.py test revisions`. `django-versioning` has been known to work on Django 1.2 but only undergoes frequent testing on Django 1.3. That said, it will probably work on any 1.x installation.

1.4 Roadmap

The first priority is better documentation. After that, there are some features that may or may not get added to the app:

- view version history and do diffs in the admin
- a view wrapper or query shortcut that can handle redirecting to the latest revision when users stumble on outdated content (e.g. when a new revision has a different slug)
- try to follow the *django-reversion* API wherever it makes sense, perhaps creating a `shortcuts.ReversionModel` model.

1.5 Changelog

- 0.5: Added support for `unique` and `unique_together` constraints at the bundle (as opposed to version) level. Improved support for UUID-based models and added tests to ascertain everything works as normal.
- 0.4: Added a `VersionedModelBase` and removed all explicit references to `vid` as the primary key, to be able to support models regardless of whether their AutoField is named `vid` and regardless of whether it works with regular IDs, UUIDs et cetera. Changed the way you add in shortcuts. Use `shortcuts.VersionedModel` to get a versioned model _with_ shortcuts, or use both `models.VersionedModel` and `shortcuts.VersionedModelShortcuts` to stick to the old ways of doing things.
- 0.3: Improved docs and added support for versioning on models with concrete inheritance.
- 0.2: First public release. Added a lot of unit tests.
- 0.1: First release.

1.6 Indices and tables

- genindex
- modindex
- search