CleanerVersion Documentation

Release 2.1.0

Jean-Christophe Zulian, Brian King, Andrea Marcacci, Manuel Jec

Mar 13, 2018

Contents

Contents:

CHAPTER 1

Historization with CleanerVersion

Disclaimer: This documentation as well as the CleanerVersion application code have been written to work against Django 1.9.x through 1.11.x. The documentation may not be accurate anymore when using more recent versions of Django.

1.1 Quick Start

1.1.1 Installation

If you don't like to work with the sources directly, you can also install the CleanerVersion package from PyPI by doing so (you may need superuser privileges, as for every other pip-installation):

```
pip install cleanerversion
```

If you want to check whether your components are compatible with CleanerVersion, you can run the unit tests coming with CleanerVersion. To do so, register the CleanerVersion test app to the `INSTALLED_APPS` variable of your Django project by adding the `versions_tests` keyword as follows:

```
INSTALLED_APPS = (
    ...
    'versions_tests',
    ...
)
```

Now, whether things work out correctly, run CleanerVersion's unit tests from within your Django project root:

```
python manage.py test versions_tests
```

If this terminates with a `OK`, you're all set. Go on and create your models as follows. Keep in mind that you are not required to keep `versions_tests` in your `INSTALLED_APPS` settings, it will only create unnecessary tables everytime you sync with your DB. So, you can safely remove it after having run the test suite.

1.1.2 A simple versionable model

First, import all the necessary modules. In this example, all the imports are done in the beginning, such that this would be a working example, if place in the same source file. Here's how:

```
from datetime import datetime
from django.db.models.fields import CharField
from django.utils.timezone import utc
from versions.models import Versionable

class Person(Versionable):
    name = CharField(max_length=200)
    address = CharField(max_length=200)
    phone = CharField(max_length=200)
```

Assuming you know how to deal with Django Models (you will need to migrate your DB before your code gets usable; Or you're only testing, then that step is done by Django), the next step is using your model to create some entries:

```
p = Person.objects.create(name='Donald Fauntleroy Duck', address='Duckburg', phone=
↪'123456')
t1 = datetime.utcnow().replace(tzinfo=utc)

p = p.clone() # Important! Fetch the returned object, it's the current one! Continue␣
↪work with this one.
p.address = 'Entenhausen'
p.save()
t2 = datetime.utcnow().replace(tzinfo=utc)

p = p.clone()
p.phone = '987654'
p.save()
t3 = datetime.utcnow().replace(tzinfo=utc)
```

Now, let's query the entries:

```
donald_current = Person.objects.as_of().get(name__startswith='Donald')  # Get the␣
↪current entry
print str(donald_current.address)  # Prints 'Entenhausen'
print str(donald_current.phone)  # Prints '987654'

donald_t1 = Person.objects.as_of(t1).get(name__startswith='Donald')  # Get a historic␣
↪entry
print str(donald_t1.address)  # Prints 'Duckburg'
print str(donald_t1.phone)  # Prints '123456'
```

1.1.3 A related versionable model

Here comes the less simple approach. What we are going to set up is both, a Many-to-One- and a Many-to-Many-relationship. Keep in mind, that this is just an example and we try to focus on the relationship part, rather than the semantical correctness of the entries' fields:

```
from datetime import datetime
from django.db.models.fields import CharField
from django.utils.timezone import utc
from versions.models import Versionable, VersionedManyToManyField, VersionedForeignKey
```

(continues on next page)

(continued from previous page)

```
class Discipline(Versionable):
    """A sports discipline"""
    name = CharField(max_length=200)
    rules = CharField(max_length=200)

class SportsClub(Versionable):
    """Sort of an association for practicing sports"""
    name = CharField(max_length=200)
    practice_periodicity = CharField(max_length=200)
    discipline = VersionedForeignKey('Discipline')

class Person(Versionable):
    name = CharField(max_length=200)
    phone = CharField(max_length=200)
    sportsclubs = VersionedManyToManyField('SportsClub', related_name='members')
```

Here comes the data loading for demo:

```
running = Discipline.objects.create(name='Running', rules='There are none (almost)')
icehockey = Discipline.objects.create(name='Ice Hockey', rules='There\'s a ton of them
↪')

stb = SportsClub.objects.create(name='STB', practice_periodicity='tuesday and␣
↪thursday night', discipline=running)
hcfg = SportsClub.objects.create(name='HCFG', practice_periodicity='monday, wednesday␣
↪and friday night', discipline=icehockey)

peter = Person.objects.create(name='Peter', phone='123456')
mary = Person.objects.create(name='Mary', phone='987654')

# Bringing things together
# Peter wants to run
peter.sportsclubs.add(stb)

t1 = datetime.utcnow().replace(tzinfo=utc)

# Peter later joins HCFG for ice hockey
hcfg.members.add(peter)

# Mary joins STB for running
stb.members.add(mary)

t2 = datetime.utcnow().replace(tzinfo=utc)

# HCFG changes the paractice times
hcfg = hcfg.clone()
hcfg.practice_periodicity = 'monday, wednesday and thursday'
hcfg.save()

# Too bad, new practice times don't work out for Peter anymore, he leaves HCFG
hcfg.members.remove(peter)
t3 = datetime.utcnow().replace(tzinfo=utc)
```

Let's continue with the queries, to check, whether all that story can be reconstructed:

```
### Querying for timestamp t1
sportsclub = SportsClub.objects.as_of(t1).get(name='HCFG')
```

(continues on next page)

(continued from previous page)

```
print "Number of " + sportsclub.name + " (" + sportsclub.discipline.name + ")␣
↪members: " + str(sportsclub.members.count())
for member in list(sportsclub.members.all()):
    print "- " + str(member.name)  # prints ""

sportsclub = SportsClub.objects.as_of(t1).get(name='STB')
print "Number of " + sportsclub.name + " (" + sportsclub.discipline.name + ")␣
↪members: " + str(sportsclub.members.count())
for member in list(sportsclub.members.all()):
    print "- " + str(member.name)  # prints "- Peter"


### Querying for timestamp t2
sportsclub = SportsClub.objects.as_of(t2).get(name='HCFG')
print "Number of " + sportsclub.name + " (" + sportsclub.discipline.name + ")␣
↪members: " + str(sportsclub.members.count())
for member in list(sportsclub.members.all()):
    print "- " + str(member.name)  # prints "- Peter"


### Querying for timestamp t3
sportsclub = SportsClub.objects.as_of(t3).get(name='HCFG')
print "Number of " + sportsclub.name + " (" + sportsclub.discipline.name + ")␣
↪members: " + str(sportsclub.members.count())
for member in list(sportsclub.members.all()):
    print "- " + str(member.name)  # prints ""

sportsclub = SportsClub.objects.as_of(t3).get(name='STB')
print "Number of " + sportsclub.name + " (" + sportsclub.discipline.name + ")␣
↪members: " + str(sportsclub.members.count())
for member in list(sportsclub.members.all()):
    print "- " + str(member.name)  # prints "- Peter\n- Mary"
```

Pretty easy, isn't it? ;)

1.2 Slowly Changing Dimensions - Type 2

Find the basics of slowly changing dimensions - type 2 and other types at Wikipedia. These concepts were taken over and extended to cover different types of relationships.

The technical details and assumptions are documented in the following sections.

1.3 Historization of a single entity

The definition of `Versionable` fields is as follows:

id The virtual ID of an entry. This field figures also as the primary key (pk) and is randomly created

identity Identifies an object over all its versions, i.e. identity does not change from one version to another

version_birth_date The timestamp at which an object was created. All versions of an object will have the same creation date.

version_start_date The timestamp at which a version was created.

version_end_date The timestamp at which a version was cloned. If a version has not been cloned yet, `version_end_date` will be set to `None` (or NULL) and the entry is considered the most recent entry of an object (i.e. it is the object's current version)

Let's assume the following class definition for this hands-on:

```
class Item(Versionable):
    name = CharField(max_length="200")  # referred to as the payload data
    version = CharField(max_length="200")  # part of the payload data as well; added␣
↪for more transparency
```

Having the class, let's create an instance of it:

```
item = Item.objects.create(name="Peter Muster", version="1")
```

This sequence of commands generated the following DB entry in the table associated to `Item` (inheriting from `Versionable`):

id (pk)	identity	version_birth_date	version_start_date	version_end_date	name	version
123	123	2014-08-14 14:43:00	2014-08-14 14:43:00	None	Peter Muster	1

Once you wish to change some value on your object, do it as follows:

```
item = item.clone()
item.name = "Peter Mauser"
item.version = "2"
item.save()
```

In the first line, we create the new version of the item entry and assign it immediately to the same variable we used to work with.

On the new version, we can now change the payload data at will and `save()` the object, once we're done.

On a DB level, things will look as follows:

id (pk)	identity	version_birth_date	version_start_date	version_end_date	name	version
123	123	2014-08-14 14:43:00	2014-08-14 15:09:00	None	Peter Mauser	2
124	123	2014-08-14 14:43:00	2014-08-14 14:43:00	2014-08-14 15:09:00	Peter Muster	1

Notice the primary key of the current entry did not change. The original `id` will always point the current version of an object.

Revisions of an object (i.e. historic versions) are copies of the current entry at the time pointed by the version's `version_end_date`.

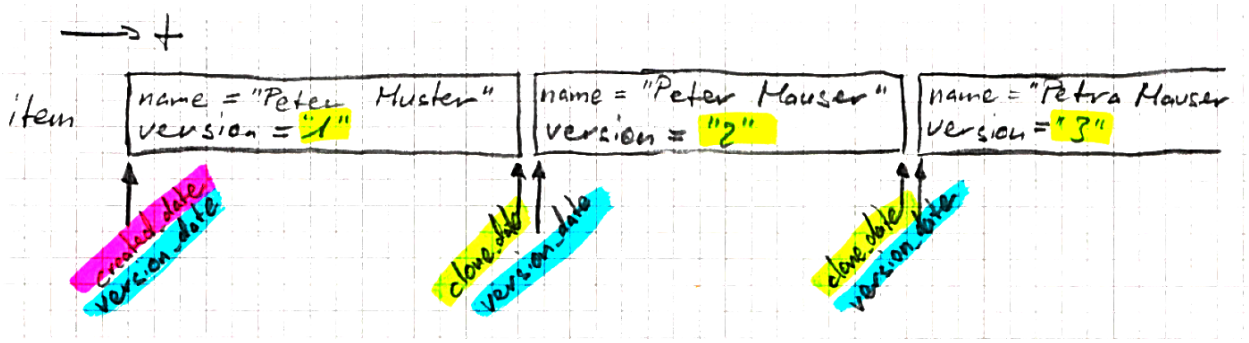
For making things clearer, we create another version:

```
item = item.clone()
item.name = "Petra Mauser"
item.version = "3"
item.save()
```

Once again, the situation on DB level will present itself as follows:

id (pk)	identity	version_birth_date	version_start_date	version_end_date	name	version
123	123	2014-08-14 14:43:00	2014-08-14 15:21:00	None	Petra Mauser	3
124	123	2014-08-14 14:43:00	2014-08-14 14:43:00	2014-08-14 15:09:00	Peter Muster	1
125	123	2014-08-14 14:43:00	2014-08-14 15:09:00	2014-08-14 15:21:00	Peter Mauser	2

On a timeline, the state can be represented as follows:



1.4 Many-to-One relationships

1.4.1 Declaring versioned M2O relationship

Here's an example with a sportsclub that can practice at most one sporty discipline:

```
class SportsClub(Versionable):
    """Sort of an association for practicing sports"""
    name = CharField(max_length=200)
    practice_periodicity = CharField(max_length=200)
    discipline = VersionedForeignKey('Discipline')

class Discipline(Versionable):
    """A sports discipline"""
    name = CharField(max_length=200)
    rules = CharField(max_length=200)
```

If a many-to-one (M2O) relationship can also be unset, don't forget to set the nullable flag (null=true) as an argument of the `VersionedForeignKey` field.

1.4.2 Adding objects to a versioned M2O relationship

Let's create two disciplines and some sportsclubs practicing these disciplines:

```
running = Discipline.objects.create(name='Running', rules='There are none (almost)')
icehockey = Discipline.objects.create(name='Ice Hockey', rules='There\'s a ton of them
↪')
```

(continues on next page)

(continued from previous page)

```
stb = SportsClub.objects.create(name='STB', practice_periodicity='tuesday and␣
↪thursday night',
                                                discipline=running)
hcfg = SportsClub.objects.create(name='HCFG',
                                                 practice_periodicity='monday, wednesday␣
↪and friday night',
                                                 discipline=icehockey)
lca = SportsClub.objects.create(name='LCA', practice_periodicity='individual',
                                                discipline=running)

t1 = datetime.utcnow().replace(tzinfo=utc)
```

1.4.3 Reading objects from a M2O relationship

Now, let's read some stuff previously loaded:

```
sportsclubs = SportsClub.objects.as_of(t1)  # This returns all SportsClubs existing␣
↪at time t1 [returned within a QuerySet]
```

You can also use select_related() to reduce the number of database queries made, if you know that you'll need the ForeignKey-related objects:

```
# Only one database query is made for this set of statements:
hcfg = SportsClub.objects.current.select_related('discipline').get(name='HCFG')
print hcfg.discipline.name
```

Note that select_related only works for models containing foreign keys. It does not work for reverse relationships:

```
# This does not save any database queries!  select_related() has no effect here:
icehockey = Discipline.objects.current.select_related('sportsclub_set').get(name='Ice␣
↪Hockey')
print icehockey.sportsclub_set.first().name
```

This is not a CleanerVersion limitation; it's just the way that Django's select_related() works. Use prefetch_related() instead if you want to prefetch reverse or many-to-many relationships. Note that prefetch_related() will use at least two queries to prefetch the related objects. See also the *Notes about using prefetch_related*.

Filtering using objects

Following on the above example, let's create a new version of the running Discipline. First, though, let's take a look at the id, identities and foreign keys as they are now:

```
>> (running.id, running.identity)
(1, 1)
>> (stb.discipline_id, stb.id, stb.identity)
(1, 10, 10)
>> (lca.discipline_id, lca.id, lca.identity)
(1, 20, 20)
```

OK, so now we create a new version:

```
running = running.clone()
running.rules = "Don't run on other's feet"
running.save()

# Fetch the old version from the database:
running_at_t1 = Discipline.objects.as_of(t1).get(name='Running')
```

How do the id, identities, and foreign keys look at this point?

```
>> (running.id, running.identity)
(1, 1)

>> (running_at_t1.id, running_at_t1.identity)
(2, 1)

>> (stb.discipline_id, stb.id, stb.identity)
(1, 10, 10)

>> (lca.discipline_id, lca.id, lca.identity)
(1, 20, 20)
```

The objects `running` and `running_at_t1` have different ids, but the same identity; they are different versions of the same object. The id of the old version has changed; the new version has the original id value.

Notice that `stb` and `lca` still refer to Discipline with id `1`. When they were created, at t1, they were actually pointing to a different version than the current version. Their discipline_id column was not updated to point to the old version when `running` was cloned. This is an important implementation detail - foreign keys point to the latest version of the foreign object, which always has it's id equal to it's identity. If this was not the case, it would be necessary to clone all of the objects that have a foreign key pointing to object X when object X is cloned; this would result in a very quickly growing database.

When searching for an object at a given time t1, foreign key values are matched against the related records identity column, and the related record are further restricted to those records that are valid at t1.

All of this should help you understand that when you filter a query for a certain point in time using an object, it's actually the identity of the object that will be used for the filtering, and not the id. You are effectively saying, "I want to limit to records that were associated *with some version of* this object".

```
>> stb1 = SportsClub.objects.as_of(t1).filter(discipline=running, name='STB').first()
>> stb2 = SportsClub.objects.as_of(t1).filter(discipline=running_at_t1, name='STB').
↪first()
>> (stb1.discipline.id, stb2.discipline.id)
(2, 2)

>> stb3 = SportsClub.objects.current.filter(discipline=running, name='STB').first()
>> stb4 = SportsClub.objects.current.filter(discipline=running_at_t1, name='STB').
↪first()
>> (stb3.discipline.id, stb4.discipline.id)
(1, 1)
```

If you really want to filter using the id of the object, you need to explicitly use the id instead of passing the object itself:

```
>> stb5 = SportsClub.objects.as_of(t1).filter(discipline_id=running.id, name='STB').
↪first()
>> stb6 = SportsClub.objects.as_of(t1).filter(discipline_id=running_at_t1.id, name=
↪'STB').first()
```

(continues on next page)

(continued from previous page)

```
>> (stb5.discipline.id, stb6 is None)
(True, 2)

>> stb7 = SportsClub.objects.current.filter(discipline_id=running.id, name='STB').
↪first()
>> stb8 = SportsClub.objects.current.filter(discipline_id=running_at_t1.id, name='STB
↪').first()
>> (stb7.discipline.id, stb8 is None)
(1, True)
```

1.5 Many-to-Many relationships

1.5.1 Declaring versioned M2M relationships

Assume a Person can be part of multiple SportsClubs:

```
class Person(Versionable):
    name = CharField(max_length=200)
    phone = CharField(max_length=200)
    sportsclubs = VersionedManyToManyField('SportsClub', related_name='members')

class SportsClub(Versionable):
    """Sort of an association for practicing sports"""
    name = CharField(max_length=200)
    practice_periodicity = CharField(max_length=200)
```

1.5.2 Adding objects to a versioned M2M relationship

Adding objects to a many-to-many relationship works just like in standard Django:

```
person1 = Person.objects.create(name="Hanover Fiste", phone="555-1234")
person2 = Person.objects.create(name="Gloria", phone="555-6777")
club = SportsClub.objects.create(name="Sweatshop", practice_periodicity="daily")


# This is one way to do it:
club.members.add(person1, person2)

# Another way to do it to assign a list.  This will remove any existing
# members that are not in the list, and add any members that are in the
# list, but not yet associated in the database.
club.members = [person1, person2]
```

Changing many-to-many relationships is only allowed when using the current version of the object:

```
# This would raise an Exception:
old_club = SportsClub.objects.previous_version(club)
old_club.members.add(person3)
```

1.5.3 Reading objects from a versioned M2M relationship

This works just like in standard Django, with the exception that you specify either that you are using the current state, or the state at a specific point in time:

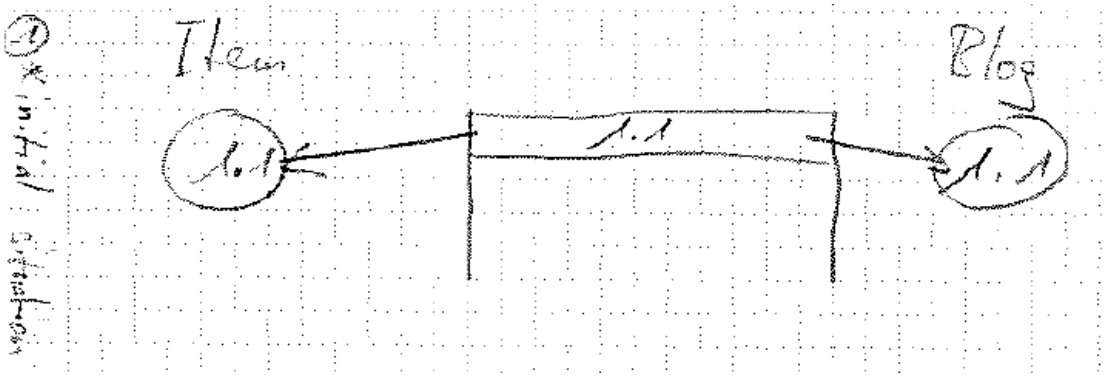
```
# Working with the current state:
club = Club.objects.current.get(name='Sweatshop')
local_members = club.members.filter(phone__startswith='555').all()

# Working with a specific point in time:
november1 = datetime(2014, 11, 1).replace(tzinfo=utc)
club = Club.objects.as_of(november1).get(name='Sweatshop')
# The related objects that are retrieved were existing and related as of november1,␣
↪too.
local_members = club.members.filter(phone__startswith='555').all()

# Queries can of course traverse relationships, too:
clubs_with_g_members = Club.objects.current.filter(members__name__startswith='G').
↪all()
```

1.5.4 Versioning objects being part of a versioned M2M relationship

Versioning an object in a ManyToMany relationship requires 3 steps to be done, including the initial setup:

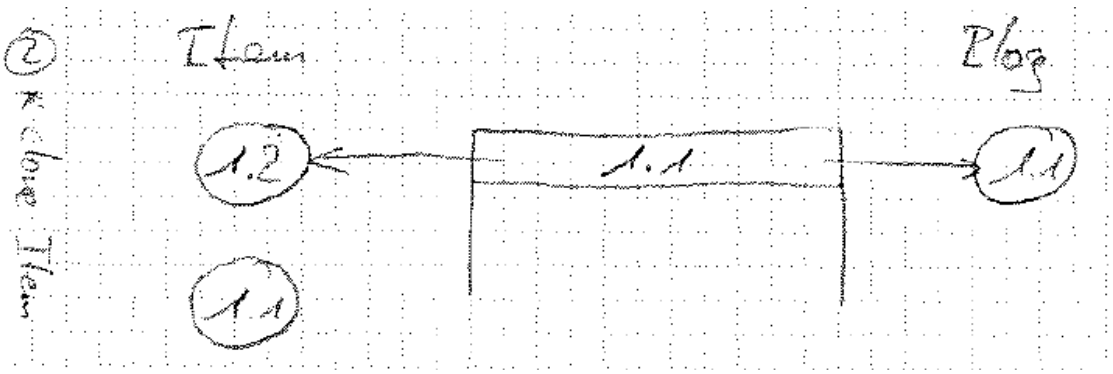
1. Setting up the situation requires to add at least two objects to a M2M relationship:

```
blog1.items.add(item1)
```



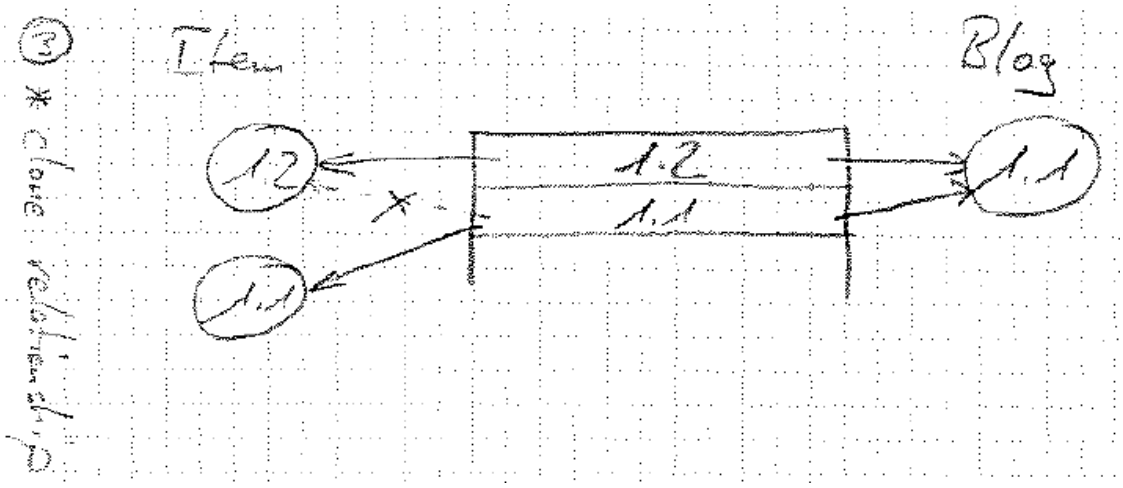
2. Further on, let's clone the Item-instance:

```
new_item1 = item1.clone()
```



3. CleanerVersion takes care of cloning and re-linking also the relationships:

```
# done automatically by cleanerversion when item1.clone() was called
```



The records in ManyToMany intermediary tables are versioned: they have `version_birth_date`, `version_start_date` and `version_end_date` columns. The ForeignKey columns in ManyToMany Intermediary tables store the `id` of the referenced records. Note that this is different than the VersionedForeignKeys in Versionable models, which store the `identity` of the referenced objects. This is transparent in normal usage, but can be important to keep in mind when you need to write a query that directly references the ForeignKey columns.

1.5.5 Removing objects from a versioned M2M relationship

Changing many-to-many relationships is only allowed when using the current version of the object.

Deleting an object from a many-to-many relationship results in the record in the relationship table being soft-deleted. In other words, a `version_end_date` is set on the relationship record.

The syntax for soft-deleting is the same as the standard Django Model deletion syntax:

```
# Various ways to remove one or more associations:
club.members.remove(person1)
club.members.remove(person2, person3)
club.members.remove(person4.id)
club.members = []
```

1.5.6 Notes about using prefetch_related

prefetch_related accepts simple sting lookups or Prefetch objects.

When using `prefetch_related` with CleanerVersion, the generated query that fetches the related objects will be time-restricted based on the base queryset. If you provide a `Prefetch` object that specifies a queryset, the queryset must either not be time-limited (using `.as_of()` or `.current`), or be time-limited with the same `.as_of` or `.current` as the base queryset. If the `Prefetch` queryset is not time-limited, but the base queryset is, the `Prefetch` queryset will adopt the same time limitation as the base queryset.

For example, assuming you want everything at the time `end_of_last_month`, you can do this:

```
# Prefetch queryset is not explicitly time-restricted, and will adopt the base␣
↪queryset's time-restriction.
disciplines_prefetch = Prefetch(
    'sportsclubs__discipline_set',
    queryset=Discipline.objects.filter('name__startswith'='B'))
people_last_month = Person.objects.as_of(end_of_last_month).prefetch_
↪related(disciplines_prefetch)
```

or this:

```
# Prefetch queryset is explicitly time-restricted with the same time restriction as␣
↪the base queryset.
disciplines_prefetch = Prefetch(
    'sportsclubs__discipline_set',
    queryset=Discipline.objects.as_of(end_of_last_month).filter('name__startswith'='B
↪'))
people_last_month = Person.objects.as_of(end_of_last_month).prefetch_
↪related(disciplines_prefetch)
```

However, the following `Prefetch`, without a time restriction that differs from the base queryset, will raise a `ValueError` when evaluated:

```
# Don't do this, the Prefetch queryset's time restriction doesn't match it's parent's:
disciplines_prefetch = Prefetch(
    'sportsclubs__discipline_set',
    queryset=Discipline.objects.current.filter('name__startswith'='B'))
people_last_month = Person.objects.as_of(end_of_last_month).prefetch_
↪related(disciplines_prefetch)
```

If a `Prefetch` without an explicit queryset is used, or a simple string lookup, the generated queryset will be appropriately time-restricted. The following statements will propagate the base query's `as_of` value to the generated related-objects queryset:

```
people1 = Person.objects.as_of(end_of_last_month).prefetch_related(Prefetch(
↪'sportsclubs__discipline_set'))
people2 = Person.objects.as_of(end_of_last_month).prefetch_related('sportsclubs__
↪discipline_set')
```

1.6 Navigating between different versions of an object

1.6.1 Accessing the version at a given point in time

If you have an object item1, and know that it existed at some other time t1, you can get the other version like this:

```
# Will throw exception if no object exists:
version = Item.objects.as_of(t1).get(identity=item1.identity)

# Or like this, which will return None if no object exists:
version = Item.objects.as_of(t1).filter(identity=item1.identity).first()
```

1.6.2 Accessing the current version of an object

`current_version(obj)` will return the latest version of the obj, or `None` if no version is currently active.

Note that if the current object thinks that it is the current object (e.g. `version_end_date` is `None`), this does not check the database. This means that if you fetched a copy of obj, and some other code has created a new version of obj before you call `current_version()`, you will get your existing obj returned, not the newest version from the database.

```
current_version = Items.objects.current_version(item1)
```

1.6.3 Accessing the previous and next versions of an object

You can navigate between the versions of an object that you have.

`previous_version(obj)` will provide the previous version of obj. If there is no previous version, the returned object will be the same object.

```
previous = Items.objects.previous_version(item1)
```

`next_version(obj)` will provide the next version of obj. If there is no next version, the returned object will be the same object.

Note that if the current object's `version_end_date` is `None`, this does not check the database. This means that if you fetched a copy of obj, and some other code has created a new version of obj before you call `next_version()`, you will get your existing obj returned, not the newest version from the database.

```
next = Items.objects.next_version(item1)
```

`current_version`, `previous_version` and `next_vesion` accept an optional parameter `relations_as_of`. This allows you to control the point in time which is used for accessing related objects (e.g. related by foreign key, reverse foreign key, one-to-one or many-to-many fields). Valid values for `relations_as_of` are:

- `'end'`: use version_end_date minus one microsecond. If the version is current, current related objects are returned when accessing relation fields. This is the default.
- `'start'`: use version_start_date
- `datetime object`: use this datetime. If the supplied datetime lies outside of the validity range of the version, a `ValueError` will be raised.
- `None`: no restriction is done. All objects ever associated with this object will be returned when accessing relation fields.

1.7 Deleting objects

You can expect `delete()` to behave like you are accustomed to in Django, with these differences:

1.7.1 Not actually deleted from the database

When you call `delete()` on a versioned object, it is not actually removed from the database. Instead, it's `version_end_date` is changed from None to a timestamp.

The same is true for the VersionedManyToManyField entries associated with the object you call `delete()` on: they are terminated by setting a `version_end_date`.

1.7.2 on_delete handlers

on_delete handlers behave like this:

The deletion is cascaded. In the CleanerVersion context, this means that the cascaded-to versions are terminated.

The cascaded-to objects are cloned before SET, SET_NULL, or SET_DEFAULT are applied.

Does nothing, just like in standard Django. This has the effect of leaving a current object with a reference to a deleted object. However, if you ask the current object for it's relations, it will not return the deleted object, because the deleted object does not match the current object's query time restriction (e.g. only current objects).

Behaves just like in standard Django.

1.8 Restoring previous versions

Previous versions can be restored like this:

```
restored_version = old_version.restore()
```

`restored_version` will now be the current version. This creates a new version, the old version is left untouched. If any current version existed when this code ran, it was terminated before the restored version was created.

Be aware that relations (VersionedForeignKey, ManyToManyField, reverse foreign keys, etc.) are not restored. You will need to restore relations yourself if necessary.

If the object being restored has a non-nullable VersionedForeignKey, you will need to supply a value (object instance or pk) for this field. If you do not supply a value, a `versions.ForeignKeyRequiresValueError` will be raised.

Values can also be provided for other, non-ForeignKey fields at restore time.

Example:

Models:

```
class Team(Versionable):
    name = models.CharField(max_length=50)

class Mascot(Versionable):
    name = models.CharField(max_length=50)
    age = models.IntegerField()
    team = VersionedForeignKey(Team, null=False)
```

Code:

```
beaver = beaver_v1.restore(team=mascot_v1.team)

# You can also use an id instead of an object when providing ForeignKeys, just be
# sure to use the field.attname (usually: field name + '_id') as the parameter name:
new_team_pk = Team.objects.current.get(name='Black Stripes').pk
tiger = tiger_v4.restore(team_id=new_team_pk, age=33)
```

1.9 Deferred fields

It is not possible to clone or restore a version that has been fetched from the database without all of it's fields, for example using one of these three equivalent statements:

```
club = Club.objects.current.defer(
    'phone', 'identity, 'version_start_date', 'version_end_date', 'version_birth_date'
).first()
club = Club.objects.current.only('name').first()
club = Club.objects.raw("""
    SELECT id, name FROM {} WHERE version_end_date IS NULL
""".format(Club._meta.db_table)[0]
```

Trying to do so will raise a ValueError. Any versioned object that needs to be cloned or restored must be fetched from the database without using `defer()` or `only()` (or `raw()` with only some of the model's fields).

1.10 Unique Indexes

To have unique indexes with versioned models takes a bit of care. The issue here is that multiple versions having the same data can exist; potentially the only difference will be with the `id`, `version_start_date`, and `version_end_date` columns.

For example, what if we want the name and phone_number to be unique together for current versions:

id (pk)	identity	version_birth_date	version_start_date	version_end_date	name	phone_number
123	123	2014-08-14 14:43:00	2014-08-14 15:21:00	None	Petra Mauser	555-1234
124	123	2014-08-14 14:43:00	2014-08-14 14:43:00	2014-08-14 15:09:00	Peter Muster	555-1234

In Postgresql, it's possible to create a partially unique index which enforces that name and phone_number are unique together when the version_end_date is null. Other databases may have a similar capability. A helper method for creating these partially unique indexes is provided for Postgresql, see the *Postgresql specific* section for more detail.

1.11 Specifying the id of an object at creation time

It is possible to specify an id when creating a new object, instead of letting CleanerVersion do this for you. The id must be a unicode string representing a version 4 UUID.

Be careful if you do this!. The possibility of collisions can increase greatly if not all sources that specify a UUID use sufficient entropy. See this for more details.

The database-level unique constraint on the id will prohibit a duplicate uuid from being inserted, but your application will need to be ready to handle that.

1.12 Postgresql specific

Django creates extra indexes for CharFields that are used for like queries (e.g. WHERE foo like 'fish%'). Since Django 1.6 (the version CleanerVersion originally targeted) did not have native database UUID fields, the UUID fields that are used for the id and identity columns of Versionable models have these extra indexes created. In fact, these fields will never be compared using the like operator. Leaving these indexes would create a performance penalty for inserts and updates, especially for larger tables. `versions.util.postgresql` has a function `remove_uuid_id_like_indexes` that can be used to remove these extra indexes.

For the issue of *Unique Indexes*, `versions.util.postgresql` has a function `create_current_version_unique_indexes` that can be used to create unique indexes. For this to work, it's necessary to define a VERSION_UNIQUE attribute when defining the model:

```
class Person(Versionable):
    name = models.CharField(max_length=40)
    phone_number = models.CharField(max_length=20)

    VERSION_UNIQUE = [['name', 'phone_number']]
```

If there are multiple sets of columns that should be unique, use something like this:

```
VERSION_UNIQUE = [['field1', 'field2'], ['field3', 'field4']]
```

As an extra method of protection against bad data appearing, it is good to ensure that only one version of an object is current at the same time. This can be done by adding a partially unique index for the `identity` column. You can use `versions.util.postgresql.create_current_version_unique_identity_indexes()` for this.

For an example of how to transparently create the database indexes for these VERSION_UNIQUE definitions in a Django app, removing the extra like indexes created on the CharField columns, and enforcing that only one version is current at the same time, see:

- https://github.com/swisscom/cleanerversion/blob/master/versions_tests/__init__.py
- https://github.com/swisscom/cleanerversion/blob/master/versions_tests/apps.py

Note that this example is for Django >= 1.7; it makes use of the application registry that was introduced in Django 1.7.

1.13 Integrating CleanerVersion versioned models with non-versioned models

It is possible to combine both, versioned models (as described up to this point) and non-versioned models.

In order to have your relationships work out correctly, make use of `VersionedForeignKey` as described in the following table. For example, one has to read the table as follows: "If a model inheriting directly from Django's `Model` is pointing a model inheriting from `Versionable`, then a `VersionedForeignKey` relation has to be used."

Model def. FK \ Model pointed by FK	models.Model	Versionable
models.Model	ForeignKey()	VersionedForeignKey()
Versionable	ForeignKey()	VersionedForeignKey()

Note that M2M-relationships have not been extended yet to work in a heterogeneous use case as described here.

1.14 VersionedAdmin admin for Django Admin

VersionedAdmin has three boolean fields that allow subclasses to easily control if the shortened identity, the version end date, and the version start date show in the change view. These fields are `list_display_show_identity`, `list_display_show_end_date`, and `list_display_show_start_date` and by default they are set to `True`.

Out of the box, VersionedAdmin allows for filtering the change view by the `as_of` queryset filter, and whether the object is current.

1.15 Upgrade notes

1.15.1 CleanerVersion 2.x / Django 1.9/1.10/1.11

In Django 1.9 major changes to the ORM layer have been introduced, which made existing versions of CleanerVersion for incompatible with Django 1.9 onwards. We decided to release a separate major version to support the Django 1.9 to 1.11.

1.15.2 CleanerVersion 1.6.0 / Django 1.8.3

Starting with CleanerVersion 1.6.0, Django's `UUIDField` will be used for the `id`, `identity`, and `VersionedForeignKey` columns if the Django version is 1.8.3 or greater.

If you are upgrading from lower versions of CleanerVersion or Django, you have two choices:

1. Add a setting to your project so that CleanerVersion will continue to use `CharField` for `Versionable`'s UUID fields. Add this to your project's settings:

```
VERSIONS_USE_UUIDFIELD = False
```

This value defaults to `True` if not explicitly set when using Django >= 1.8.3.

2. Convert all of the relevant database fields to the type and size that Django uses for UUID fields for the database that you are using. This may be possible using Django's migrations, or could be done manually by altering the column type as necessary for your database type for all the `id`, `identity`, and foreign key columns of your `Versionable` models (don't forget the auto-generated many-to-many tables). This is not a trivial undertaking; it will involve for example dropping and recreating constraints. An example of column altering syntax for PostgreSQL:

```
ALTER TABLE blog_author ALTER COLUMN id type uuid USING id:uuid;
ALTER TABLE blog_author ALTER COLUMN identity type uuid USING identity:uuid;
```

You must choose one or the other solution; not doing so will result in your application no longer working.

1.16 Known Issues

- No multi-table inheritance support. Multi-table inheritance currently does not work if the parent model has a Versionable base class. See this issue for more details.
- Creating *Unique Indexes* is a bit tricky for versioned database tables. A solution is provided for Postgresql (see the *Postgresql specific* section). Pull requests are welcome if you solve this problem for another database system.

For a more up-to-date state please check our project page.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search