django-transaction-hooks Documentation

Release 0.2

Carl Meyer

September 27, 2017

A better alternative to the transaction signals Django will never have.

Sometimes you need to fire off an action related to the current database transaction, but only if the transaction successfully commits. Examples: a Celery task, an email notification, or a cache invalidation.

Doing this correctly while accounting for savepoints that might be individually rolled back, closed/dropped connections, and idiosyncrasies of various databases, is non-trivial. Transaction signals just make it easier to do it wrong.

`django-transaction-hooks` does the heavy lifting so you don't have to.

CHAPTER 1

Prerequisites

`django-transaction-hooks` supports Django 1.6.1 and later on Python 2.6, 2.7, 3.2, and 3.3.

SQLite3, PostgreSQL, and MySQL are currently the only databases with built-in support; you can experiment with whether it works for your favorite database backend with just a few lines of code.

CHAPTER 2

Installation

`django-transaction-hooks` is available on PyPI. Install it with:

```
pip install django-transaction-hooks
```

CHAPTER 3

Setup

django-transaction-hooks is implemented via custom database backends. (Why backends?)

For example, to use the PosgreSQL backend, set the ENGINE in your DATABASES setting to transaction_hooks.backends.postgresql_psycopg2 (in place of django.db.backends.postgresql_psycopg2). For example:

```
DATABASES = {
    'default': {
        'ENGINE': 'transaction_hooks.backends.postgresql_psycopg2',
        'NAME': 'foo',
        },
    }
```

MySQL, SQLite, and PostGIS are similarly supported, via transaction_hooks.backends.mysql, transaction_hooks.backends.sqlite3, and transaction_hooks.backends.postgis.

Using the mixin

If you're currently using Django's built-in database backend for SQLite, Postgres, PostGIS, or MySQL, you can skip this section; just use the appropriate backend from transaction_hooks.backends as outlined above.

Not using one of those? No worries - all the magic happens in a mixin, so making it happen with your favorite database backend may not be hard (no guarantees it'll work right, though.)

You'll need to create your own custom backend that inherits both from transaction_hooks.mixin.TransactionHooksDatabaseWrapperMixin and from the database backend you're currently using. To do this, make a Python package (a directory with an __init__.py file in it) somewhere, and then put a base.py module inside that package. Its contents should look something like this:

```
from django.db.backends.postgresql_psycopg2 import base
from transaction_hooks.mixin import TransactionHooksDatabaseWrapperMixin

class DatabaseWrapper(TransactionHooksDatabaseWrapperMixin,
                      base.DatabaseWrapper):
    pass
```

Obviously you'll want to replace django.db.backends.postgresql_psycopg2 with whatever existing backend you are currently using.

Then set your database ENGINE (as above) to the Python dotted path to the package containing that base.py module. For example, if you put the above code in myproject/mybackend/base.py, your ENGINE setting would be myproject.mybackend.

CHAPTER 4

Usage

Pass any function (that takes no arguments) to `connection.on_commit`:

```
from django.db import connection

def do_something():
    # send a mail, fire off a Celery task, what-have-you.

connection.on_commit(do_something)
```

You can also wrap your thing up in a lambda:

```
connection.on_commit(lambda: some_celery_task.delay('arg1'))
```

The function you pass in will be called immediately after a hypothetical database write made at the same point in your code is successfully committed. If that hypothetical database write is instead rolled back, your function will be discarded and never called.

If you register a callback while there is no transaction active, it will be executed immediately.

Notes

Warning: This code is new, not yet battle-tested, and probably has bugs. If you find one, please report it.

Use autocommit and transaction.atomic

`django-transaction-hooks` is only built and tested to work correctly in autocommit mode, which is the default in Django 1.6+, and with the transaction.atomic / ATOMIC_REQUESTS transaction API. If you set autocommit off on your connection and/or use lower-level transaction APIs directly, `django-transaction-hooks` likely won't work as you expect.

For instance, commit hooks are not run until autocommit is restored on the connection following the commit (because otherwise any queries done in a commit hook would open an implicit transaction, preventing the connection from going back into autocommit mode). Also, even though with autocommit off you'd generally be in an implicit transaction outside of any `atomic` block, callback hooks registered outside an `atomic` block will still run immediately, not on commit. And there are probably more gotchas here.

Use autocommit mode and transaction.atomic (or ATOMIC_REQUESTS) and you'll be happier.

Order of execution

On-commit hooks for a given transaction are executed in the order they were registered.

Exception handling

If one on-commit hook within a given transaction raises an uncaught exception, no later-registered hooks in that same transaction will run. (This is, of course, the same behavior as if you'd executed the hooks sequentially yourself without `on_commit()`.)

Timing of execution

Your hook functions are executed *after* a successful commit, so if they fail, it will not cause the transaction to roll back. They are executed conditionally upon the success of the transaction, but they are not *part* of the transaction. For the intended use cases (mail notifications, Celery tasks, etc), this is probably fine. If it's not (if your follow-up action is so critical that its failure should mean the failure of the transaction itself), then you don't want `django-transaction-hooks`. (Instead, you may want two-phase commit.)

Use with South

If you use South, you will probably need to set the SOUTH_DATABASE_ADAPTERS setting when you switch to a custom database backend (e.g. to `{'default': 'south.db.postgresql_psycopg2'}`, if you are using PostgreSQL).

Use in tests

Django's TestCase class wraps each test in a transaction and rolls back that transaction after each test, in order to provide test isolation. This means that no transaction is ever actually committed, thus your `on_commit` hooks will never be run. If you need to test the results of an `on_commit` hook, you may need to use TransactionTestCase instead.

Savepoints

Savepoints (i.e. nested `transaction.atomic` blocks) are handled correctly. That is, an `on_commit` hook registered after a savepoint (in a nested `atomic` block) will be called after the outer transaction is committed, but not if a rollback to that savepoint or any previous savepoint occurred during the transaction.

Why database backends?

Yeah, it's a bit of a pain. But since all transaction state is stored on the database connection object, this is the only way it can be done without monkeypatching. And I hate monkeypatching.

(The worst bit about a custom database backend is that if you need two different ones, they can be hard or impossible to compose together. In this case, the mixin should make that less painful.)

If this turns out to be really popular, it might be possible to get something like it into the Django core backends, which would remove that issue entirely.

Why no rollback hook?

A rollback hook is even harder to implement robustly than a commit hook, since a variety of things can cause an implicit rollback. For instance, your database connection was dropped because your process was killed without a chance to shutdown gracefully: your rollback hook will never run.

The solution is simple: instead of doing something during the atomic block (transaction) and then undoing it if the transaction fails, use `on_commit` to delay doing it in the first place until after the transaction succeeds. It's a lot easier to undo something you never did in the first place!

CHAPTER 5

Contributing

See the contributing docs.