TRANSACTION

Django mastery in Nepali

Database transactions

Django gives you a few ways to control how database transactions are managed.

Django's default transaction behavior

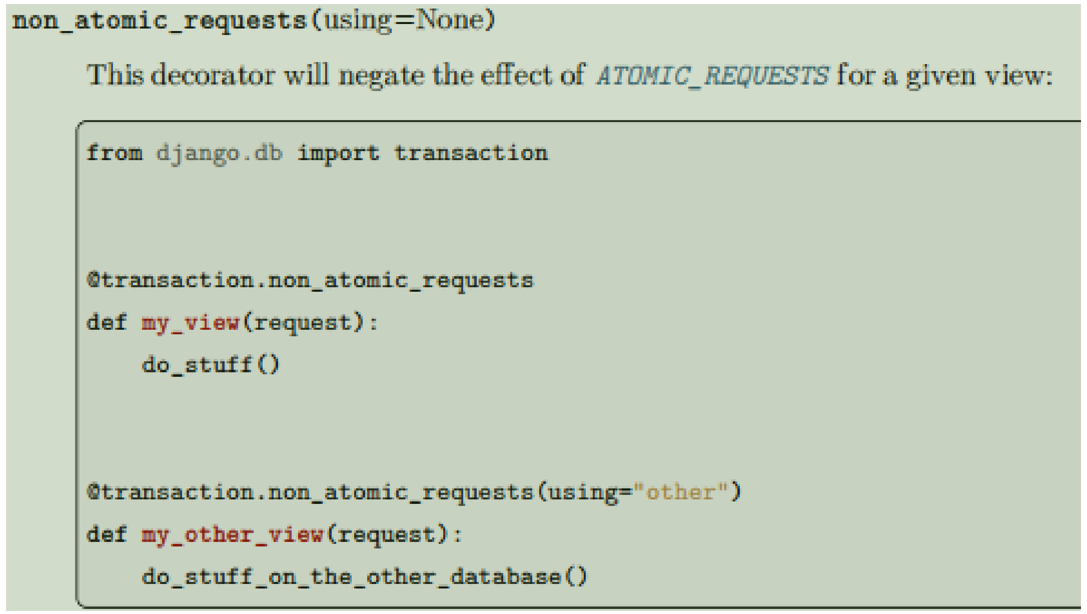
Django's default behavior is to run in autocommit mode. Each query is immediately committed to the database, unless a transaction is active. Django uses transactions or savepoints automatically to guarantee the integrity of ORM operations that require multiple queries, especially delete() and update() queries. Django's TestCase class also wraps each test in a transaction for performance reasons.

Tying transactions to HTTP requests

A common way to handle transactions on the web is to wrap each request in a transaction. Set ATOMIC_REQUESTS to True in the configuration of each database for which you want to enable this behavior. It works like this. Before calling a view function, Django starts a transaction. If the response is produced without problems, Django commits the transaction. If the view produces an exception, Django rolls back the transaction. You may perform subtransactions using savepoints in your view code, typically with the atomic() context manager. However, at the end of the view, either all or none of the changes will be committed

Warning: While the simplicity of this transaction model is appealing, it also makes it inefficient when traffic increases. Opening a transaction for every view has some overhead. The impact on performance depends on the query patterns of your application and on how well your database handles locking.

In practice, this feature wraps every view function in the atomic() decorator described below. Note that only the execution of your view is enclosed in the transactions. Middleware runs outside of the transaction, and so does the rendering of template responses. When ATOMIC_REQUESTS is enabled, it's still possible to prevent views from running in a transaction.



non_atomic_requests(using=None)

This decorator will negate the effect of *ATOMIC_REQUESTS* for a given view:

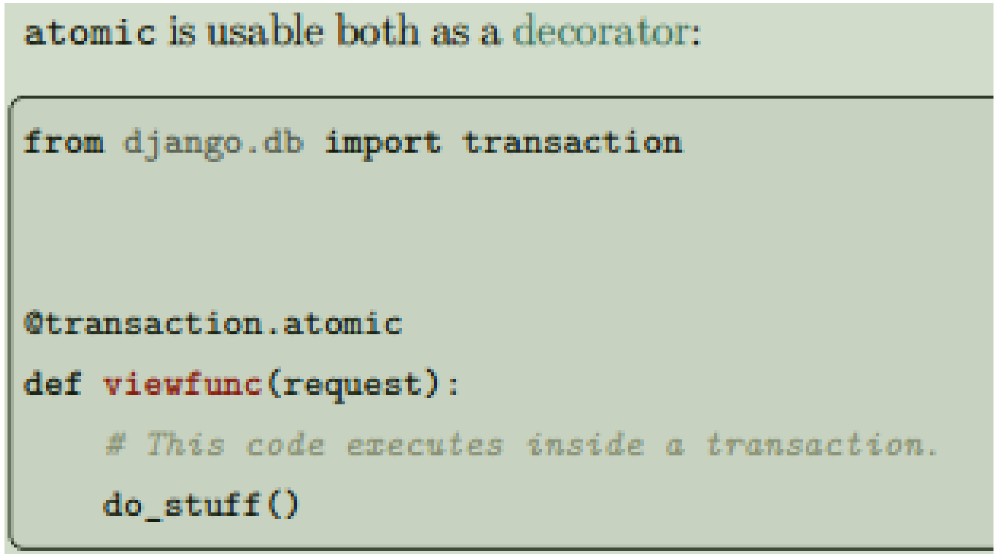
```
from django.db import transaction


@transaction.non_atomic_requests
def my_view(request):
    do_stuff()


@transaction.non_atomic_requests(using="other")
def my_other_view(request):
    do_stuff_on_the_other_database()
```

Controlling transactions explicitly

Django provides a single API to control database transactions.

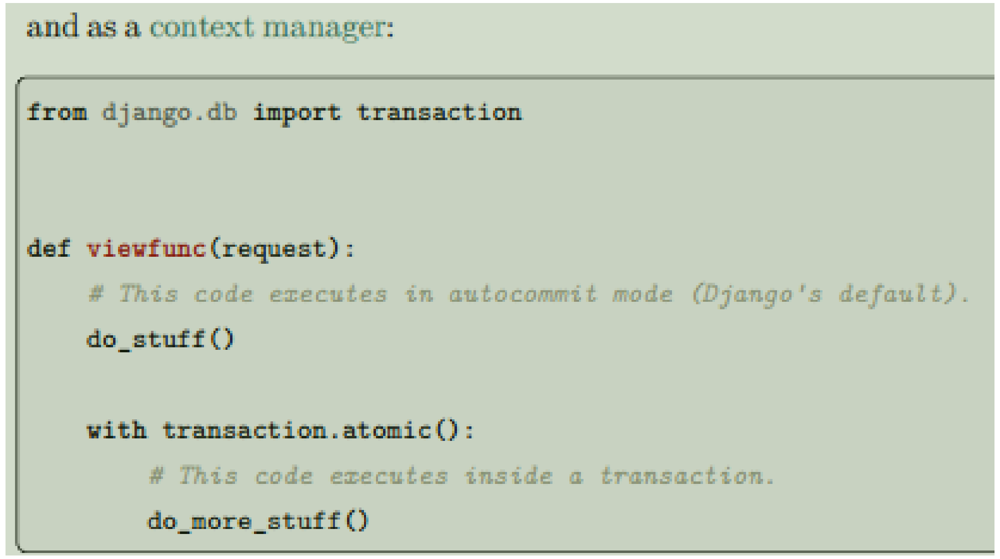
atomic(using=None, savepoint=True, durable=False)

Atomicity is the defining property of database transactions. atomic allows us to create a block of code
within which the atomicity on the database is guaranteed. If the block of code is successfully completed, the changes are committed to the database. If there is an exception, the changes are rolled back.
atomic blocks can be nested. In this case, when an inner block completes successfully, its effects can
still be rolled back if an exception is raised in the outer block at a later point. It is sometimes useful to ensure an atomic block is always the outermost atomic block, ensuring that any database changes are committed when the block is exited without errors. This is known as durability and can be achieved by setting durable=True. If the atomic block is nested within another it raises a RuntimeError.



atomic is usable both as a decorator:

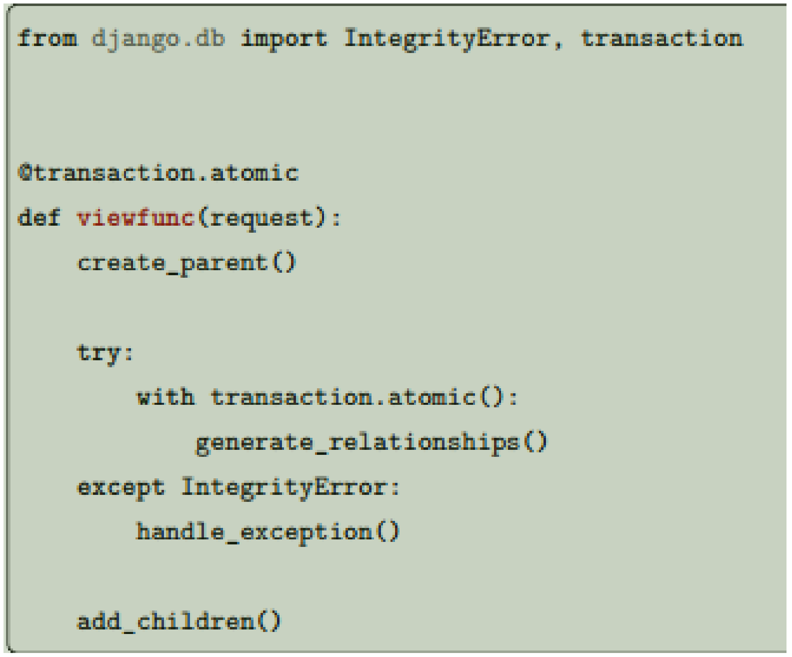
```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```



and as a context manager:

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Wrapping atomic in a try/except block allows for natural handling of integrity errors:



```
from django.db import IntegrityError, transaction

@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

In this example, even if `generate_relationships()` causes a database error by breaking an integrity constraint, you can execute queries in `add_children()`, and the changes from `create_parent()` are still there and bound to the same transaction. Note that any operations attempted in `generate_relationships()` will already have been rolled back safely when `handle_exception()` is called, so the exception handler can also operate on the database if necessary.

Avoid catching exceptions inside atomic!

When exiting an atomic block, Django looks at whether it’s exited normally or with an exception to determine whether to commit or roll back. If you catch and handle exceptions inside an atomic block, you may hide from Django the fact that a problem has happened. This can result in unexpected behavior.

atomic takes a using argument which should be the name of a database. If this argument isn’t provided, Django uses the "default" database. Under the hood, Django’s transaction management code:

- opens a transaction when entering the outermost atomic block;
- creates a savepoint when entering an inner atomic block;
- releases or rolls back to the savepoint when exiting an inner block;
- commits or rolls back the transaction when exiting the outermost block

You may use atomic when autocommit is turned off. It will only use savepoints, even for the outermost block.

Deactivating transaction management

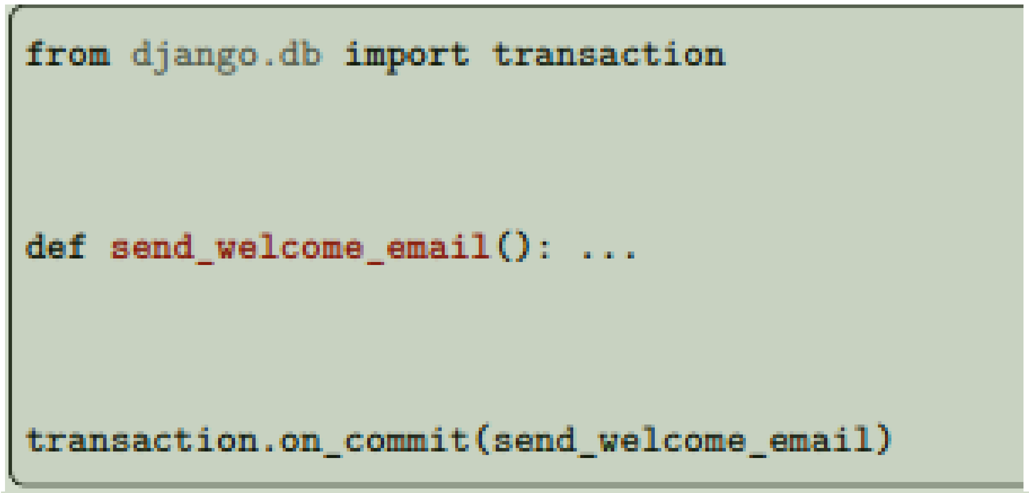
You can totally disable Django’s transaction management for a given database by setting AUTOCOMMIT to False in its configuration. If you do this, Django won’t enable autocommit and won’t perform any commits. You'll get the regular behavior of the underlying database library. This requires you to commit explicitly every transaction, even those started by Django or by third-party libraries.

Performing actions after commit

Sometimes you need to perform an action related to the current database transaction, but only if the transaction successfully commits. Examples might include a background task, an email notification, or a cache invalidation.
on_commit() allows you to register callbacks that will be executed after the open transaction is successfully committed:

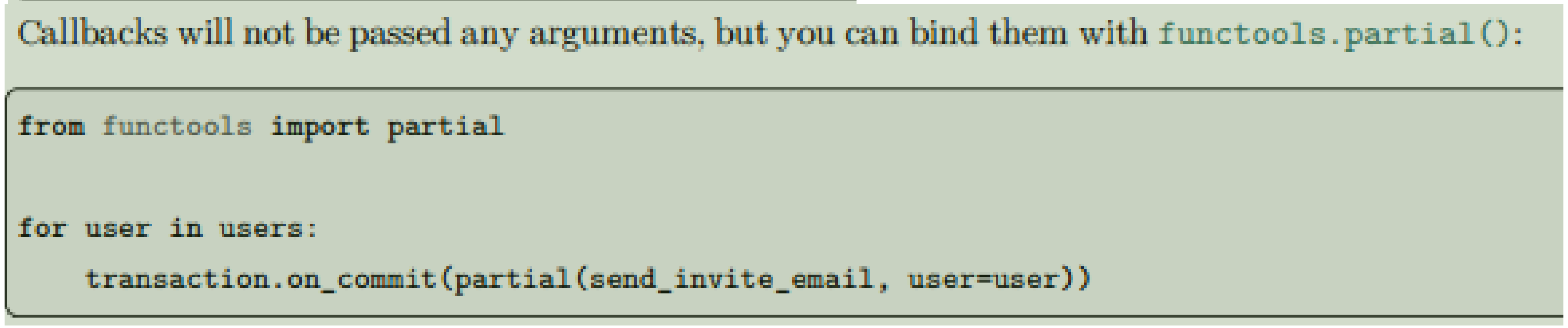
on_commit(func, using=None, robust=False)

Pass a function, or any callable, to on_commit():



```
from django.db import transaction


def send_welcome_email(): ...


transaction.on_commit(send_welcome_email)
```

If you call **on_commit()** while there isn't an open transaction, the callback will be executed immediately. It's sometimes useful to register callbacks that can fail. Passing robust=True allows the next callbacks to be executed even if the current one throws an exception. All errors derived from Python's Exception class are caught and logged to the **django.db.backends.base** logger. You can use **TestCase.captureOnCommitCallbacks()** to test callbacks registered with **on_commit().**

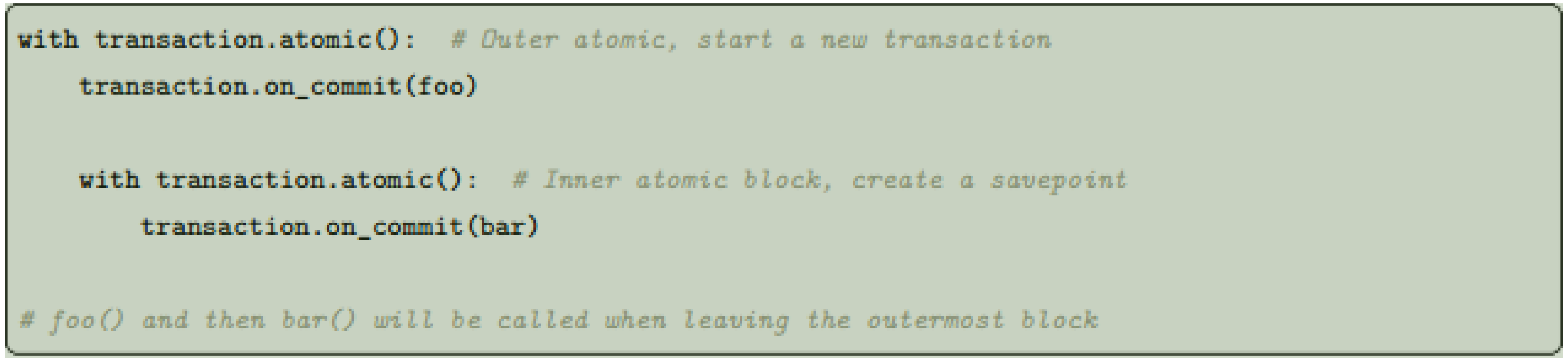


Callbacks will not be passed any arguments, but you can bind them with `functools.partial()`:

```
from functools import partial

for user in users:
    transaction.on_commit(partial(send_invite_email, user=user))
```

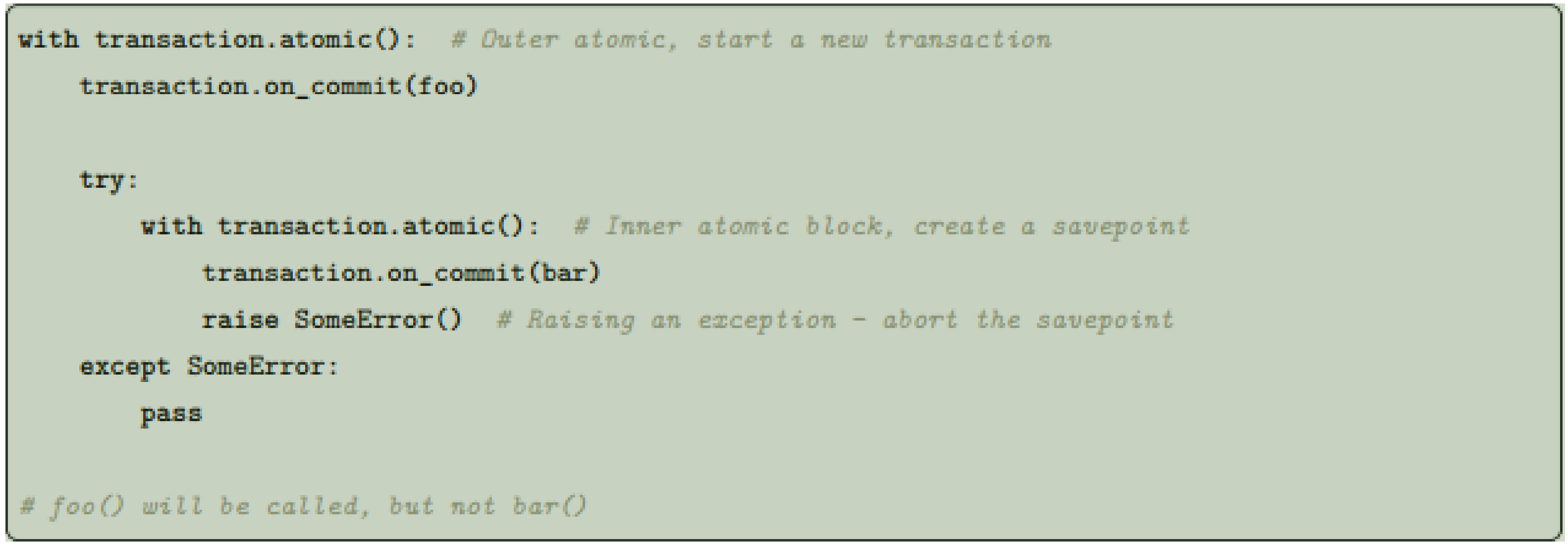
Savepoints

Savepoints (i.e. nested atomic() blocks) are handled correctly. That is, an on_commit() callable 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:



```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    with transaction.atomic():  # Inner atomic block, create a savepoint
        transaction.on_commit(bar)

# foo() and then bar() will be called when leaving the outermost block
```

On the other hand, when a savepoint is rolled back (due to an exception being raised), the inner callable will not be called:



```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    try:
        with transaction.atomic():  # Inner atomic block, create a savepoint
            transaction.on_commit(bar)
            raise SomeError()  # Raising an exception - abort the savepoint
    except SomeError:
        pass

# foo() will be called, but not bar()
```

Timing of execution

Your callbacks are executed after a successful commit, so a failure in a callback 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, background tasks, etc.), this should be 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 to use the on_commit() hook. Instead, you may want two-phase commit such as the psycopg Two-Phase Commit protocol support and the optional Two-Phase Commit Extensions in the Python DB-API specification. Callbacks are not run until autocommit is restored on the connection following the commit (because otherwise any queries done in a callback would open an implicit transaction, preventing the connection from going back into autocommit mode). When in autocommit mode and outside of an atomic() block, the function will run immediately, not on commit. On-commit functions only work with autocommit mode and the atomic() (or ATOMIC_REQUESTS) transaction API. Calling on_commit() when autocommit is disabled and you are not within an atomic block will result in an error.

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()** callbacks will never be run. You can overcome this limitation by using **TestCase.captureOnCommitCallbacks().** This captures your **on_commit()** callbacks in a list, allowing you to make assertions on them, or emulate the transaction committing by calling them. Another way to overcome the limitation is to use **TransactionTestCase** instead of TestCase. This will mean your transactions are committed, and the callbacks will run. However **TransactionTestCase** flushes the database between tests, which is significantly slower than **TestCase's** isolation.

Autocommit

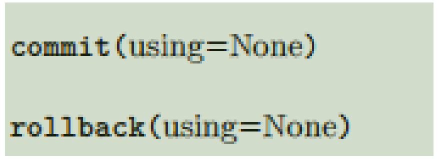
Django provides an API in the django.db.transaction module to manage the autocommit state of each database connection.

```
get_autocommit(using=None)
set_autocommit(autocommit, using=None)
```

These functions take a using argument which should be the name of a database. If it isn't provided, Django uses the "default" database. Autocommit is initially turned on. If you turn it off, it's your responsibility to restore it. Once you turn autocommit off, you get the default behavior of your database adapter, and Django won't help you. Although that behavior is specified in PEP 249, implementations of adapters aren't always consistent with one another. Review the documentation of the adapter you're using carefully. You must ensure that no transaction is active, usually by issuing a commit( ) or a rollback( ), before turning autocommit back on. Django will refuse to turn autocommit off when an atomic( ) block is active, because that would break atomicity

Transactions

A transaction is an atomic set of database queries. Even if your program crashes, the database guarantees that either all the changes will be applied, or none of them. Django doesn't provide an API to start a transaction. The expected way to start a transaction is to disable autocommit with set_autocommit(). Once you're in a transaction, you can choose either to apply the changes you've performed until this point with commit( ), or to cancel them with rollback( ). These functions are defined in django.db.transaction



```
commit(using=None)

rollback(using=None)
```

These functions take a using argument which should be the name of a database. If it isn't provided, Django uses the "default" database. Django will refuse to commit or to rollback when an atomic( ) block is active, because that would break atomicity.

Savepoints

A savepoint is a marker within a transaction that enables you to roll back part of a transaction, rather than the full transaction. Savepoints are available with the SQLite, PostgreSQL, Oracle, and MySQL (when using the InnoDB storage engine) backends. Other backends provide the savepoint functions, but they're empty operations – they don't actually do anything. Savepoints aren't especially useful if you are using autocommit, the default behavior of Django. However, once you open a transaction with atomic(), you build up a series of database operations awaiting a commit or rollback. If you issue a rollback, the entire transaction is rolled back. Savepoints provide the ability to perform a fine-grained rollback, rather than the full rollback that would be performed by transaction. rollback(). When the atomic() decorator is nested, it creates a savepoint to allow partial commit or rollback. You're strongly encouraged to use atomic() rather than the functions described below, but they're still part of the public API, and there's no plan to deprecate them. Each of these functions takes a using argument which should be the name of a database for which the behavior applies. If no using argument is provided then the "default" database is used. Savepoints are controlled by three functions in

django.db.transaction:

savepoint(using=None)

Creates a new savepoint. This marks a point in the transaction that is known to be in a "good" state. Returns the savepoint ID (sid).

savepoint_commit(sid, using=None)

Releases savepoint sid. The changes performed since the savepoint was created become part of the transaction.

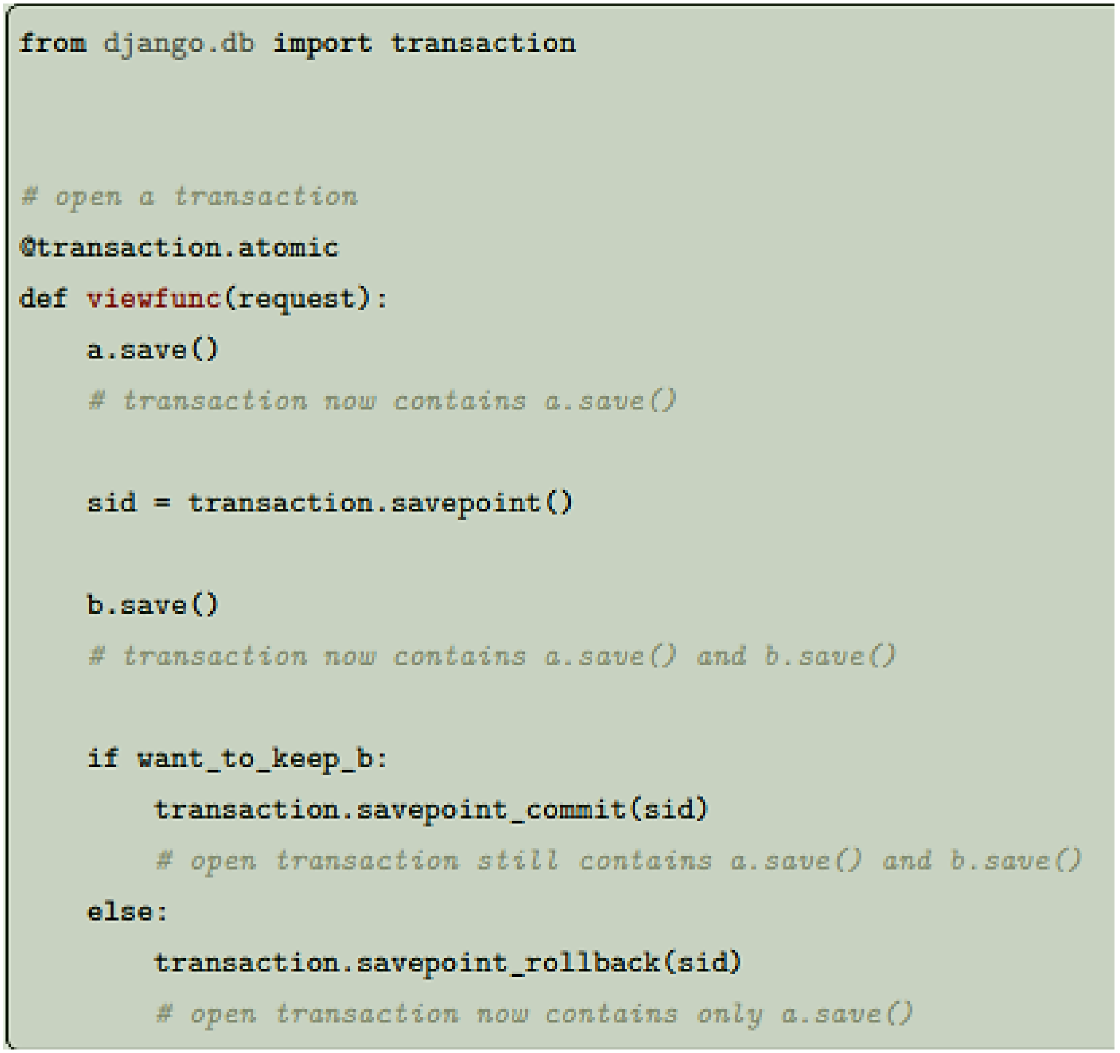
savepoint_rollback(sid, using=None)

Rolls back the transaction to savepoint sid

In addition, there's a utility function:

clean_savepoints(using=None)

Resets the counter used to generate unique savepoint IDs.



```
from django.db import transaction


# open a transaction
@transaction.atomic
def viewfunc(request):
    a.save()
    # transaction now contains a.save()

    sid = transaction.savepoint()

    b.save()
    # transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()
```

Savepoints may be used to recover from a database error by performing a partial rollback. If you're doing this inside an atomic() block, the entire block will still be rolled back, because it doesn't know you've handled the situation at a lower level! To prevent this, you can control the rollback behavior with the following functions.

`get_rollback(using=None)`

`set_rollback(rollback, using=None)`

Setting the rollback flag to True forces a rollback when exiting the innermost atomic block. This may be useful to trigger a rollback without raising an exception.

Setting it to False prevents such a rollback. Before doing that, make sure you've rolled back the transaction to a known-good savepoint within the current atomic block! Otherwise, you're breaking atomicity and data corruption may occur.

Thank you for watching 💖