Database transactions ¶

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

Managing database transactions ¶

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. See below for details.

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.

Per-request transactions and streaming responses

When a view returns a **`StreamingHttpResponse`**, reading the contents of the response will often execute code to generate the content. Since the view has already returned, such code runs outside of the transaction.

Generally speaking, it isn't advisable to write to the database while generating a streaming response, since there's no sensible way to handle errors after starting to send the response.

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`*`)[source]` ¶

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

Getting Help

Language: **en**

Documentation version: **5.0**

```
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()
```

It only works if it’s applied to the view itself.

Controlling transactions explicitly ¶

Django provides a single API to control database transactions.

atomic(*using=None*, *savepoint=True*, *durable=False*)[source] ¶

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.

This is mostly a concern for **`DatabaseError`** and its subclasses such as **`IntegrityError`**. After such an error, the transaction is broken and Django will perform a rollback at the end of the **`atomic`** block. If you attempt to run database queries before the rollback happens, Django will raise a **`TransactionManagementError`**. You may also encounter this behavior when an ORM-related signal handler raises an exception.

The correct way to catch database errors is around an **`atomic`** block as shown above. If necessary, add an extra **`atomic`** block for this purpose. This pattern has another advantage: it delimits explicitly which operations will be rolled back if an exception occurs.

If you catch exceptions raised by raw SQL queries, Django's behavior is unspecified and database-dependent.

You may need to manually revert app state when rolling back a transaction.

The values of a model's fields won't be reverted when a transaction rollback happens. This could lead to an inconsistent model state unless you manually restore the original field values.

For example, given **`MyModel`** with an **`active`** field, this snippet ensures that the **`if obj.active`** check at the end uses the correct value if updating **`active`** to **`True`** fails in the transaction:

```
from django.db import DatabaseError, transaction

obj = MyModel(active=False)
obj.active = True
try:
    with transaction.atomic():
        obj.save()
except DatabaseError:
    obj.active = False

if obj.active:
    ...
```

This also applies to any other mechanism that may hold app state, such as caching or global variables. For example, if the code proactively updates data in the cache after saving an object, it's recommended to use transaction.on_commit() instead, to defer cache alterations until the transaction is actually committed.

In order to guarantee atomicity, **`atomic`** disables some APIs. Attempting to commit, roll back, or change the autocommit state of the database connection within an **`atomic`** block will raise an exception.

`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 can disable the creation of savepoints for inner blocks by setting the **`savepoint`** argument to **`False`**. If an exception occurs, Django will perform the rollback when exiting the first parent block with a savepoint if there is one, and the outermost block otherwise. Atomicity is still guaranteed by the outer transaction. This option should only be used if the overhead of savepoints is noticeable. It has the drawback of breaking the error handling described above.

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

Performance considerations

Open transactions have a performance cost for your database server. To minimize this overhead, keep your transactions as short as possible. This is especially important if you're using **`atomic()`** in long-running processes, outside of Django's request / response cycle.

Autocommit ¶

Why Django uses autocommit ¶

In the SQL standards, each SQL query starts a transaction, unless one is already active. Such transactions must then be explicitly committed or rolled back.

This isn't always convenient for application developers. To alleviate this problem, most databases provide an autocommit mode. When autocommit is turned on and no transaction is active, each SQL query gets wrapped in its own transaction. In other words, not only does each such query start a transaction, but the transaction also gets automatically committed or rolled back, depending on whether the query succeeded.

PEP 249, the Python Database API Specification v2.0, requires autocommit to be initially turned off. Django overrides this default and turns autocommit on.

To avoid this, you can deactivate the transaction management, but it isn't recommended.

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. Thus, this is best used in situations where you want to run your own transaction-controlling middleware or do something really strange.

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`*`)[source]` ¶

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

```
from django.db import transaction


def send_welcome_email(): ...


transaction.on_commit(send_welcome_email)
```

Getting Help

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

Language: **en**

Documentation version: **5.0**

```
from functools import partial

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

Callbacks are called after the open transaction is successfully committed. If the transaction is instead rolled back (typically when an unhandled exception is raised in an **`atomic()`** block), the callback will be discarded, and never called.

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()`**.

Changed in Django 4.2:

The **`robust`** argument was added.

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()
```

Order of execution ¶

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

Exception handling ¶

If one on-commit function registered with **`robust=False`** within a given transaction raises an uncaught exception, no later registered functions in that same transaction will run. This is the same behavior as if you'd executed the functions sequentially yourself without **`on_commit()`**.

Changed in Django 4.2:

The **`robust`** argument was added.

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.

Getting Help
Language: **en**
Documentation version: **5.0**

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.

Why no rollback hook? ¶

A rollback hook is harder to implement robustly than a commit hook, since a variety of things can cause an implicit rollback.

For instance, if your database connection is dropped because your process was killed without a chance to shut down gracefully, your rollback hook will never run.

But there is a solution: 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!

Low-level APIs ¶

Warning

Always prefer `atomic()` if possible at all. It accounts for the idiosyncrasies of each database and prevents invalid operations.

The low level APIs are only useful if you're implementing your own transaction management.

Autocommit ¶

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

`get_autocommit(`*`using=None`*`)`[source] ¶

`set_autocommit(`*`autocommit`*`, `*`using=None`*`)`[source] ¶

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`*`)`[source] ¶

`rollback(`*`using=None`*`)`[source] ¶

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*)[source] ¶

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*)[source] ¶

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

`savepoint_rollback`(*sid*, *using=None*)[source] ¶

Rolls back the transaction to savepoint **`sid`**.

These functions do nothing if savepoints aren't supported or if the database is in autocommit mode.

In addition, there's a utility function:

`clean_savepoints`(*using=None*)[source] ¶

Resets the counter used to generate unique savepoint IDs.

The following example demonstrates the use of savepoints:

```
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*)[source] ¶

`set_rollback`(*rollback*, *using=None*)[source] ¶

Getting Help

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.

Language: **en**

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.

Documentation version: **5.0**

Database-specific notes ¶

Savepoints in SQLite ¶

While SQLite supports savepoints, a flaw in the design of the **`sqlite3`** module makes them hardly usable.

When autocommit is enabled, savepoints don't make sense. When it's disabled, **`sqlite3`** commits implicitly before savepoint statements. (In fact, it commits before any statement other than **`SELECT`**, **`INSERT`**, **`UPDATE`**, **`DELETE`** and **`REPLACE`**.) This bug has two consequences:

- The low level APIs for savepoints are only usable inside a transaction i.e. inside an **`atomic()`** block.
- It's impossible to use **`atomic()`** when autocommit is turned off.

Transactions in MySQL ¶

If you're using MySQL, your tables may or may not support transactions; it depends on your MySQL version and the table types you're using. (By "table types," we mean something like "InnoDB" or "MyISAM".) MySQL transaction peculiarities are outside the scope of this article, but the MySQL site has information on MySQL transactions.

If your MySQL setup does *not* support transactions, then Django will always function in autocommit mode: statements will be executed and committed as soon as they're called. If your MySQL setup *does* support transactions, Django will handle transactions as explained in this document.

Handling exceptions within PostgreSQL transactions ¶

Note

This section is relevant only if you're implementing your own transaction management. This problem cannot occur in Django's default mode and **`atomic()`** handles it automatically.

Inside a transaction, when a call to a PostgreSQL cursor raises an exception (typically **`IntegrityError`**), all subsequent SQL in the same transaction will fail with the error "current transaction is aborted, queries ignored until end of transaction block". While the basic use of **`save()`** is unlikely to raise an exception in PostgreSQL, there are more advanced usage patterns which might, such as saving objects with unique fields, saving using the **`force_insert`**/**`force_update`** flag, or invoking custom SQL.

There are several ways to recover from this sort of error.

Transaction rollback ¶

The first option is to roll back the entire transaction. For example:

```
a.save()  # Succeeds, but may be undone by transaction rollback
try:
    b.save()  # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save()  # Succeeds, but a.save() may have been undone
```

Calling **`transaction.rollback()`** rolls back the entire transaction. Any uncommitted database operations will be lost. In this example, the changes made by **`a.save()`** would be lost, even though that operation raised no error itself.

Savepoint rollback ¶

You can use savepoints to control the extent of a rollback. Before performing a database operation that could fail, you can set or update the savepoint; that way, if the operation fails, you can roll back the single offending operation, rather than the entire transaction. For example:

```
a.save()  # Succeeds, and never undone by savepoint rollback
sid = transaction.savepoint()
try:
    b.save()  # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save()  # Succeeds, and a.save() is never undone
```

In this example, **`a.save()`** will not be undone in the case where **`b.save()`** raises an exception.