Django ORM Transactions

What is a Transaction?
In database terms, a transaction is a unit of work that ensures **ACID** properties (Atomicity, Consistency, Isolation, Durability).

- **The Golden Rule:** Either *all* database operations within the transaction succeed, or *none* of them do. If an error occurs, the database rolls back to its state before the transaction began.

---

1. Global Configuration
The simplest way to handle transactions is to tie them to the HTTP request.
In `settings.py`:

Python
```
DATABASES = {
    'default': {
        # ...
        'ATOMIC_REQUESTS': True,
    }
}
```

- **How it works:** Django starts a transaction when a request comes in. If the view produces a response without errors, it **commits**. If an exception is raised, it **rolls back**.
- **Pros:** Easy safety net.
- **Cons:** Can be inefficient if views verify data but don't modify the DB; triggers transaction overhead on every request.

---

2. Explicit Control: `transaction.atomic`
For granular control (and better performance), use `django.db.transaction`. This is the industry standard approach.

A. As a Context Manager (Preferred)
Use this within a block of code to ensure a specific set of operations are atomic.

Python

```
from django.db import transaction
from .models import Order, LineItem

def create_order(user, items_data):
    try:
        # Start the transaction
        with transaction.atomic():
            # Operation 1: Create the parent object
            order = Order.objects.create(user=user)

            # Operation 2: Create child objects
            for item in items_data:
                LineItem.objects.create(order=order, **item)

            # If code reaches here, it Auto-Commits

    except Exception as e:
        # If an error occurred inside the block, the DB is
        # already rolled back to the state before 'order' was
created.
        print(f"Order failed: {e}")
```

B. As a Decorator
You can wrap an entire function or view.

Python

```
from django.db import transaction

@transaction.atomic
def process_payment(payment_id):
    payment = Payment.objects.get(pk=payment_id)
    payment.status = 'COMPLETED'
    payment.save()

    # If this fails, the payment status change is reverted
    update_ledger(payment)
```

---

3. Nested Transactions (Savepoints)

Django handles nested `atomic()` blocks automatically using **savepoints**.

Python

```
with transaction.atomic():  # Outer transaction
    User.objects.create(username='outer')

    try:
        with transaction.atomic():  # Inner transaction (Savep
oint)
            User.objects.create(username='inner')
            raise RuntimeError("Oops!") # Rolls back ONLY 'inn
er'
    except RuntimeError:
        pass

    # 'outer' is still saved because the exception was caught
    # and the outer block finished successfully.
```

---

4. Handling Side Effects: `on_commit`

Crucial: Never send emails or charge credit cards *inside* a transaction block if you can avoid it. If the transaction rolls back but the email was already sent, you have a phantom action.
Use `transaction.on_commit` to run code **only** after the transaction successfully commits.

Python
```
def register_user(user_data):
    with transaction.atomic():
        user = User.objects.create(**user_data)

        # Only send the email if the user is actually persiste
d to DB
        transaction.on_commit(lambda: send_welcome_email(user.
email))
```

Summary of Best Practices

1. **Keep it small:** Only wrap DB operations in `atomic()`. Avoid wrapping external API calls (which are slow) inside a transaction, as this holds database locks longer than necessary.
2. **Exception Handling:** To trigger a rollback, an exception **must** be raised. Do not catch exceptions *inside*the `atomic` block unless you intend to suppress the rollback or use savepoints. Catch them *outside* the block.
3. **Use `select_for_update`:** If you are reading data, modifying it, and saving it inside a transaction, use `.select_for_update()` on your queryset to lock the rows and prevent race conditions.