PostgreSQL with Django: Complete In-Depth Tutorial

I'll teach you PostgreSQL integration with Django, covering ORM, Migrations, and Docker Integration from beginner to advanced level.

---

1. Django ORM (Object-Relational Mapper)

What is ORM and Why Do We Need It?

Simple Analogy: Think of ORM as a **translator** between two people who speak different languages. You speak Python, and PostgreSQL speaks SQL. The ORM translates your Python code into SQL queries that PostgreSQL understands.

Without ORM (Raw SQL):

```python
import psycopg2
cursor.execute("SELECT * FROM tasks WHERE status = 'completed'")
results = cursor.fetchall()
```

With Django ORM:

```python
tasks = Task.objects.filter(status='completed')
```

Why ORM is Essential:

1. **Security**: Prevents SQL injection attacks
2. **Portability**: Same code works with different databases
3. **Maintainability**: Python code is easier to read than complex SQL
4. **Relationships**: Handles complex table relationships automatically

---

How Django ORM Works Internally

The Three-Layer Architecture

```
Python Code (models.py)
      ↓
Django ORM (QuerySet API)
      ↓
SQL Database (PostgreSQL)
```

1. Model Definition Phase

```python
# models.py
from django.db import models

class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    status = models.CharField(max_length=20, default='pending')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
```

What Django Does Behind the Scenes:

- Creates a **metaclass** that analyzes your model
- Maps each field to a PostgreSQL column type
- Generates SQL CREATE TABLE statements
- Sets up primary keys, indexes, and constraints

2. QuerySet Lazy Evaluation

Key Concept: Django QuerySets are **lazy** - they don't hit the database until you actually need the data.

```python
# This doesn't execute any SQL yet!
pending_tasks = Task.objects.filter(status='pending')

# SQL is executed only when you iterate or evaluate
for task in pending_tasks:  # NOW SQL executes
    print(task.title)
```

Django ORM Core Operations

Basic CRUD Operations

1. CREATE Operations

```python
# Method 1: Create and save
task = Task(title="Learn Django", description="Study ORM concepts")
task.save()

# Method 2: Create in one step
task = Task.objects.create(
    title="Learn PostgreSQL",
    description="Understand database concepts"
)

# Method 3: Bulk create (more efficient for multiple records)
tasks = [
    Task(title="Task 1", description="Description 1"),
    Task(title="Task 2", description="Description 2"),
    Task(title="Task 3", description="Description 3"),
]
Task.objects.bulk_create(tasks)
```

Generated SQL:

```sql
INSERT INTO myapp_task (title, description, status, created_at, updated_at)
VALUES ('Learn Django', 'Study ORM concepts', 'pending', NOW(), NOW());
```

2. READ Operations

```python
# Get all records
all_tasks = Task.objects.all()

# Filter records
pending_tasks = Task.objects.filter(status='pending')
completed_tasks = Task.objects.filter(status='completed')

# Get single record
task = Task.objects.get(id=1)  # Raises exception if not found
task = Task.objects.filter(id=1).first()  # Returns None if not found

# Complex filtering
recent_tasks = Task.objects.filter(
    created_at__gte=timezone.now() - timedelta(days=7),
    status__in=['pending', 'in_progress']
)
```

Generated SQL Examples:

```sql
-- Task.objects.all()
SELECT * FROM myapp_task;

-- Task.objects.filter(status='pending')
SELECT * FROM myapp_task WHERE status = 'pending';

-- Complex filter
SELECT * FROM myapp_task
WHERE created_at >= '2024-08-13'
AND status IN ('pending', 'in_progress');
```

3. UPDATE Operations

```python
# Update single record
task = Task.objects.get(id=1)
task.status = 'completed'
task.save()

# Bulk update (more efficient)
Task.objects.filter(status='pending').update(status='in_progress')

# Update with calculations
from django.db.models import F
Task.objects.filter(priority__lt=5).update(priority=F('priority') + 1)
```

4. DELETE Operations

```python
# Delete single record
task = Task.objects.get(id=1)
task.delete()

# Bulk delete
Task.objects.filter(status='completed').delete()

# Delete all (be careful!)
Task.objects.all().delete()
```

Advanced Querying and Relationships

Model Relationships

python

```python
# models.py
class User(models.Model):
    username = models.CharField(max_length=150)
    email = models.EmailField()

class Category(models.Model):
    name = models.CharField(max_length=100)

class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    user = models.ForeignKey(User, on_delete=models.CASCADE)  # Many-to-One
    category = models.ForeignKey(Category, on_delete=models.SET_NULL, null=True)
    tags = models.ManyToManyField('Tag')  # Many-to-Many

class Tag(models.Model):
    name = models.CharField(max_length=50)
```

Relationship Queries

Forward Relationships

python

```python
# Get user of a task
task = Task.objects.get(id=1)
user = task.user  # This triggers a database query

# More efficient - use select_related
task = Task.objects.select_related('user').get(id=1)
user = task.user  # No additional query needed
```

Reverse Relationships

```python
# Get all tasks for a user
user = User.objects.get(id=1)
user_tasks = user.task_set.all()  # Default reverse relation

# Or with related_name
class Task(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='tasks')

# Now you can use:
user_tasks = user.tasks.all()
```

Complex Queries with Joins

```python
# Get all tasks with their users and categories
tasks = Task.objects.select_related('user', 'category').all()

# Get all users with their tasks
users = User.objects.prefetch_related('tasks').all()

# Complex filtering across relationships
tasks = Task.objects.filter(
    user__username='john_doe',
    category__name='Work',
    tags__name='urgent'
).distinct()
```

Generated SQL:

```sql
-- select_related creates JOINs
SELECT task.*, user.*, category.*
FROM myapp_task task
JOIN myapp_user user ON task.user_id = user.id
JOIN myapp_category category ON task.category_id = category.id;

-- prefetch_related uses separate queries
SELECT * FROM myapp_user;
SELECT * FROM myapp_task WHERE user_id IN (1, 2, 3, ...);
```

Query Optimization Best Practices

1. N+1 Query Problem

Bad Example (Creates N+1 queries):

```python
tasks = Task.objects.all()
for task in tasks:
    print(task.user.username)  # Each iteration hits database!
```

Good Example (Single query with JOIN):

```python
tasks = Task.objects.select_related('user').all()
for task in tasks:
    print(task.user.username)  # No additional queries
```

2. Use only() and defer() for Large Objects

```python
# Only fetch specific fields
tasks = Task.objects.only('title', 'status').all()

# Defer heavy fields
tasks = Task.objects.defer('description').all()
```

3. Aggregate and Annotate

```python
from django.db.models import Count, Avg

# Count tasks per user
users = User.objects.annotate(task_count=Count('tasks'))

# Average task completion time
avg_completion = Task.objects.aggregate(Avg('completion_time'))
```

---

2. Django Migrations

What Are Migrations?

Simple Analogy: Migrations are like **version control for your database schema**. Just like Git tracks changes to your code, migrations track changes to your database structure.

Why Migrations Are Essential:

1. **Team Collaboration**: Everyone gets the same database structure
2. **Deployment Safety**: Apply changes consistently across environments
3. **Rollback Capability**: Undo problematic changes
4. **Change Tracking**: See what changed and when

How Django Migration System Works

The Migration Lifecycle

```
Model Changes → makemigrations → Migration Files → migrate → Database Schema
```

1. Migration Files Structure

```python
# migrations/0001_initial.py
from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True

    dependencies = []

    operations = [
        migrations.CreateModel(
            name='Task',
            fields=[
                ('id', models.AutoField(primary_key=True)),
                ('title', models.CharField(max_length=200)),
                ('description', models.TextField()),
                ('status', models.CharField(default='pending', max_length=20)),
                ('created_at', models.DateTimeField(auto_now_add=True)),
            ],
        ),
    ]
```

2. Django Migration Commands Deep Dive

`python manage.py makemigrations`

What it does:

1. Compares current models.py with last migration
2. Detects changes (new models, fields, deletions)
3. Generates Python migration file
4. Assigns sequential number

Example Workflow:

```bash
# Add new field to model
class Task(models.Model):
    title = models.CharField(max_length=200)
    priority = models.IntegerField(default=1)  # NEW FIELD

# Generate migration
python manage.py makemigrations
```

Generated Migration:

```python
# migrations/0002_task_priority.py
operations = [
    migrations.AddField(
        model_name='task',
        name='priority',
        field=models.IntegerField(default=1),
    ),
]
```

`python manage.py migrate`

What it does:

1. Checks django_migrations table for applied migrations
2. Applies pending migrations in order
3. Updates django_migrations table
4. Executes SQL commands

Example SQL Generated:

```sql
-- For AddField operation
ALTER TABLE myapp_task ADD COLUMN priority INTEGER DEFAULT 1 NOT NULL;

-- Django tracks this in:
INSERT INTO django_migrations (app, name, applied)
VALUES ('myapp', '0002_task_priority', NOW());
```

Migration Types and Operations

1. Model Operations

```python
# Create Model
migrations.CreateModel(
    name='Category',
    fields=[
        ('id', models.AutoField(primary_key=True)),
        ('name', models.CharField(max_length=100)),
    ],
)

# Delete Model
migrations.DeleteModel(name='OldModel'),

# Rename Model
migrations.RenameModel(old_name='Task', new_name='TodoItem'),
```

2. Field Operations

```python
# Add Field
migrations.AddField(
    model_name='task',
    name='due_date',
    field=models.DateField(null=True),
)

# Remove Field
migrations.RemoveField(model_name='task', name='old_field'),

# Alter Field
migrations.AlterField(
    model_name='task',
    name='title',
    field=models.CharField(max_length=300),  # Changed from 200
)

# Rename Field
migrations.RenameField(
    model_name='task',
    old_name='desc',
    new_name='description',
)
```

3. Data Migrations

Sometimes you need to migrate data, not just schema:

```python
# migrations/0003_populate_categories.py
from django.db import migrations

def populate_categories(apps, schema_editor):
    Category = apps.get_model('myapp', 'Category')
    Category.objects.create(name='Work')
    Category.objects.create(name='Personal')
    Category.objects.create(name='Shopping')

def reverse_populate_categories(apps, schema_editor):
    Category = apps.get_model('myapp', 'Category')
    Category.objects.filter(name__in=['Work', 'Personal', 'Shopping']).delete()

class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0002_category'),
    ]

    operations = [
        migrations.RunPython(populate_categories, reverse_populate_categories),
    ]
```

Advanced Migration Techniques

1. Migration Dependencies

```python
class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0001_initial'),
        ('otherapp', '0003_add_user_profile'),  # Cross-app dependency
    ]
```

2. Custom Migration Operations

python

```python
# Custom SQL
migrations.RunSQL(
    "CREATE INDEX idx_task_status_created ON myapp_task(status, created_at);",
    reverse_sql="DROP INDEX idx_task_status_created;"
)

# Custom Python code
migrations.RunPython(
    code=forward_func,
    reverse_code=reverse_func
)
```

3. Squashing Migrations

When you have many small migrations, combine them:

bash

```bash
python manage.py squashmigrations myapp 0001 0005
```

Migration Best Practices and Common Issues

1. Safe Migration Practices

python

```python
# ❌ Dangerous - removes data
migrations.RemoveField(model_name='task', name='important_data')

# ✅ Safe - add new field first, migrate data, then remove old
# Step 1: Add new field
migrations.AddField(
    model_name='task',
    name='new_field',
    field=models.CharField(max_length=100, null=True)
)

# Step 2: Data migration (separate migration)
# Step 3: Remove old field (another migration)
```

2. Handling Migration Conflicts

```bash
# When migrations conflict
python manage.py makemigrations --merge
```

3. Rolling Back Migrations

```bash
# Rollback to specific migration
python manage.py migrate myapp 0001

# Rollback all migrations for an app
python manage.py migrate myapp zero
```

4. Production Migration Strategy

```python
# settings.py - Different migration settings for production
if ENVIRONMENT == 'production':
    # Require explicit migration approval
    MIGRATION_ATOMIC = True
    MIGRATION_TIMEOUT = 300
```

---

3. Docker Integration with PostgreSQL

What is Docker and Why Use It?

Simple Analogy: Docker is like a **shipping container** for your application. Just like shipping containers standardize how goods are transported, Docker containers standardize how applications run across different environments.

Benefits for Django + PostgreSQL:

1. **Consistency**: Same environment everywhere (dev, test, prod)
2. **Isolation**: No conflicts with system PostgreSQL
3. **Easy Setup**: New developers get running quickly
4. **Version Control**: Lock specific PostgreSQL versions

Setting Up PostgreSQL with Docker Compose

1. Basic docker-compose.yml Structure

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  # PostgreSQL Database Service
  db:
    image: postgres:15
    container_name: django_postgres
    restart: always
    environment:
      POSTGRES_DB: myproject_db
      POSTGRES_USER: myproject_user
      POSTGRES_PASSWORD: myproject_password
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - django_network

  # Django Web Application Service
  web:
    build: .
    container_name: django_web
    restart: always
    command: python manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    environment:
      - DEBUG=1
      - DATABASE_URL=postgresql://myproject_user:myproject_password@db:5432/myproject_db
    depends_on:
      - db
    networks:
      - django_network

# Named volumes for data persistence
volumes:
  postgres_data:

# Custom network for service communication
networks:
```

```
networks:
  django_network:
    driver: bridge
```

2. Understanding Each Component

PostgreSQL Service Configuration

```yaml
db:
  image: postgres:15  # Official PostgreSQL image, version 15
  container_name: django_postgres  # Custom container name
  restart: always  # Restart container if it crashes
```

Environment Variables:

- `POSTGRES_DB`: Database name to create
- `POSTGRES_USER`: Database user to create
- `POSTGRES_PASSWORD`: Password for the user

Volume Mounting Explained

```yaml
volumes:
  - postgres_data:/var/lib/postgresql/data  # Named volume for persistence
  - ./init.sql:/docker-entrypoint-initdb.d/init.sql  # Initialization script
```

Volume Types:

1. **Named Volume** (`postgres_data`): Managed by Docker, persists data
2. **Bind Mount** (`./init.sql`): Links host file to container

Network Configuration

```yaml
networks:
  django_network:
    driver: bridge
```

This creates an isolated network where services can communicate using service names as hostnames.

Django Configuration for Docker PostgreSQL

1. Environment-Based Configuration

python

```python
# settings.py
import os
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# Database Configuration
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('DB_NAME', 'myproject_db'),
        'USER': os.environ.get('DB_USER', 'myproject_user'),
        'PASSWORD': os.environ.get('DB_PASSWORD', 'myproject_password'),
        'HOST': os.environ.get('DB_HOST', 'db'),  # 'db' is the service name
        'PORT': os.environ.get('DB_PORT', '5432'),
    }
}

# Alternative: Using DATABASE_URL
import dj_database_url
DATABASES = {
    'default': dj_database_url.parse(
        os.environ.get('DATABASE_URL', 'postgresql://myproject_user:myproject_password@db:5432/myproject_db')
    )
}
```

2. Environment Variable Management

.env File (for development)

bash

```bash
# .env
DEBUG=1
SECRET_KEY=your-secret-key-here
DB_NAME=myproject_db
DB_USER=myproject_user
DB_PASSWORD=myproject_password
DB_HOST=db
DB_PORT=5432
DATABASE_URL=postgresql://myproject_user:myproject_password@db:5432/myproject_db
```

Loading Environment Variables

```python
# settings.py
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Now use os.environ.get() as shown above
```

3. Requirements and Dockerfile

requirements.txt

```txt
Django>=4.2.0
psycopg2-binary>=2.9.0
python-dotenv>=1.0.0
dj-database-url>=2.0.0
```

Dockerfile

dockerfile

```
# Dockerfile
FROM python:3.11-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /app

# Install system dependencies
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        postgresql-client \
        build-essential \
        libpq-dev \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt

# Copy project
COPY . /app/

# Expose port
EXPOSE 8000

# Run migrations and start server
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Advanced Docker Configuration

1. Multi-Environment Setup

Development Environment

```yaml
# docker-compose.dev.yml
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: myproject_dev
      POSTGRES_USER: dev_user
      POSTGRES_PASSWORD: dev_password
    volumes:
      - postgres_dev_data:/var/lib/postgresql/data
    ports:
      - "5433:5432"  # Different port for dev

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    environment:
      - DEBUG=1
      - DATABASE_URL=postgresql://dev_user:dev_password@db:5432/myproject_dev
    ports:
      - "8000:8000"
    depends_on:
      - db

volumes:
  postgres_dev_data:
```

Production Environment

yaml

```yaml
# docker-compose.prod.yml
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: myproject_prod
      POSTGRES_USER: prod_user
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
      - postgres_prod_data:/var/lib/postgresql/data
    networks:
      - internal

  web:
    build:
      context: .
      dockerfile: Dockerfile.prod
    command: gunicorn myproject.wsgi:application --bind 0.0.0.0:8000
    environment:
      - DEBUG=0
      - DATABASE_URL_FILE=/run/secrets/database_url
    secrets:
      - database_url
    depends_on:
      - db
    networks:
      - internal
      - web

secrets:
  db_password:
    file: ./secrets/db_password.txt
  database_url:
    file: ./secrets/database_url.txt

volumes:
  postgres_prod_data:

networks:
  internal:
  web:
    external: true
```

2. Health Checks and Waiting

```yaml
# docker-compose.yml with health checks
services:
  db:
    image: postgres:15
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U myproject_user -d myproject_db"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 60s
    # ... other configurations

  web:
    # ... other configurations
    depends_on:
      db:
        condition: service_healthy
```

3. Initialization Scripts

```sql
-- init.sql
-- This runs when the container starts for the first time

-- Create additional databases
CREATE DATABASE myproject_test;

-- Create extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pg_trgm";

-- Create custom indexes or functions
-- ...
```

Docker Workflow and Best Practices

1. Development Workflow

```bash
# Start services
docker-compose up -d

# Run migrations
docker-compose exec web python manage.py migrate

# Create superuser
docker-compose exec web python manage.py createsuperuser

# View logs
docker-compose logs -f web
docker-compose logs -f db

# Stop services
docker-compose down

# Stop and remove volumes (careful - deletes data!)
docker-compose down -v
```

2. Data Management

Backup Database

```bash
# Create backup
docker-compose exec db pg_dump -U myproject_user myproject_db > backup.sql

# Restore backup
docker-compose exec -T db psql -U myproject_user myproject_db < backup.sql
```

Access Database Directly

```bash
# Connect to PostgreSQL container
docker-compose exec db psql -U myproject_user -d myproject_db

# Or from outside container
docker-compose exec db psql -U myproject_user -d myproject_db -c "SELECT * FROM myapp_task;"
```

3. Environment Separation Best Practices

```bash
# Use different compose files for different environments
docker-compose -f docker-compose.dev.yml up  # Development
docker-compose -f docker-compose.prod.yml up  # Production

# Override specific services
docker-compose -f docker-compose.yml -f docker-compose.override.yml up
```

4. Security Best Practices

1. **Use secrets for passwords**:

```yaml
secrets:
  postgres_password:
    file: ./secrets/postgres_password.txt
```

2. **Limit network exposure**:

```yaml
services:
  db:
    # Don't expose port in production
    # ports: - "5432:5432"  # Remove this line
```

3. **Use non-root user**:

```dockerfile
RUN adduser --disabled-password --gecos '' appuser
USER appuser
```

4. **Set resource limits**:

```yaml
services:
  db:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'
```

Troubleshooting Common Issues

1. Connection Issues

Problem: `django.db.utils.OperationalError: could not connect to server`

Solutions:

```bash
# Check if containers are running
docker-compose ps

# Check container logs
docker-compose logs db
docker-compose logs web

# Verify network connectivity
docker-compose exec web ping db

# Check PostgreSQL is accepting connections
docker-compose exec db pg_isready -U myproject_user
```

2. Permission Issues

Problem: `permission denied for database`

Solution:

```sql
-- Connect as superuser and grant permissions
GRANT ALL PRIVILEGES ON DATABASE myproject_db TO myproject_user;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO myproject_user;
```

3. Migration Issues

Problem: `relation "django_migrations" does not exist`

Solution:

```bash
# Ensure database is created and accessible
docker-compose exec web python manage.py migrate --run-syncdb
```

4. Data Persistence Issues

Problem: Data disappears when containers restart

Solution:

```yaml
# Ensure named volumes are used
volumes:
  - postgres_data:/var/lib/postgresql/data  # Not ./data
```

Summary and Next Steps

You've now learned:

1. **Django ORM**: How to interact with PostgreSQL through Python objects, understanding query generation, relationships, and optimization
2. **Migrations**: How Django tracks and applies database schema changes safely across environments
3. **Docker Integration**: How to containerize your Django + PostgreSQL stack for consistent development and deployment

Key Takeaways:

- Always use `select_related()` and `prefetch_related()` for relationship queries
- Test migrations thoroughly before applying to production
- Use named volumes for data persistence in Docker
- Separate configuration for different environments
- Always backup your data before major migrations

Practice Project: Create a task management system with users, categories, and tags using all these concepts together!