Topic 08 — PostgreSQL Setup and Security

This topic demonstrates how to install PostgreSQL on a local server, configure security, and adapt the pets application from Topic 04 to use PostgreSQL instead of SQLite. PostgreSQL is a production-grade relational database commonly used in enterprise environments, and understanding its security model is essential for professional development.

Why PostgreSQL?

PostgreSQL is an open-source, enterprise-class relational database system with:

- **ACID compliance**: Guarantees data integrity
- **Advanced features**: JSON support, full-text search, GIS capabilities
- **Robust security**: Role-based access control, SSL/TLS, row-level security
- **Scalability**: Handles large datasets and concurrent users
- **Wide adoption**: Used by Apple, Instagram, Spotify, Reddit, and many others

SQLite vs PostgreSQL:

Feature	SQLite	PostgreSQL
Architecture	File-based, embedded	Client-server
Concurrency	Limited (single writer)	Excellent (MVCC)
Security	File permissions only	Role-based access control
Users	Single user	Multiple users/roles
Network	Local only	Network accessible
Use Case	Development, embedded	Production, multi-user

Part 1: Installing PostgreSQL

macOS Installation

Option 1: Homebrew (Recommended)

```
# Install PostgreSQL
brew install postgresql@15

# Start PostgreSQL service
brew services start postgresql@15

# Verify installation
psql --version
```

Option 2: Postgres.app Download from postgresapp.com - GUI application with simple setup.

Linux (Ubuntu/Debian) Installation

```
# Update package list
sudo apt update

# Install PostgreSQL
sudo apt install postgresql postgresql-contrib

# Start PostgreSQL service
sudo systemctl start postgresql
sudo systemctl enable postgresql  # Start on boot

# Verify installation
psql --version
```

Verify PostgreSQL is Running

```
# Check service status
# macOS (Homebrew):
brew services list | grep postgresql

# Linux:
sudo systemctl status postgresql

# Check if listening on port 5432
lsof -i :5432
# or
netstat -an | grep 5432
```

Part 2: PostgreSQL Security Setup

Understanding PostgreSQL Authentication

PostgreSQL uses `pg_hba.conf` (Host-Based Authentication) to control who can connect and how they authenticate.

Common authentication methods:

- **trust**: No password required (INSECURE - dev only!)
- **password**: Plain text password (avoid in production)
- **md5**: MD5-hashed password (deprecated, use scram-sha-256)
- **scram-sha-256**: Modern encrypted password (recommended)
- **peer**: Use OS user authentication (Unix sockets only)
- **ident**: Use OS user authentication (TCP connections)

Initial Setup as Postgres Superuser

PostgreSQL creates a default `postgres` superuser during installation.

macOS (Homebrew):

```
# Connect as your current user (has superuser privileges)
psql postgres
```

Linux:

```
# Switch to postgres user
sudo -i -u postgres

# Connect to PostgreSQL
psql
```

Step 1: Change Postgres User Password

```
-- Set a password for the postgres superuser
ALTER USER postgres WITH PASSWORD 'securePostgresPass123!';

-- Exit
\q
```

Step 2: Create Application Database and User

Best practice: Don't use the `postgres` superuser for applications. Create dedicated users with minimal privileges.

```
# Connect as postgres user
psql -U postgres
```

```
-- Create a database for the pets application
CREATE DATABASE pets_db;

-- Create a user for the application
CREATE USER pets_app WITH PASSWORD 'petsAppPassword456!';

-- Grant connection privilege
GRANT CONNECT ON DATABASE pets_db TO pets_app;

-- Connect to the pets_db database
\c pets_db
```

```
-- Grant schema usage
GRANT USAGE ON SCHEMA public TO pets_app;

-- Grant table privileges (all tables in public schema)
-- These must be set after tables are created, or use ALTER DEFAULT
PRIVILEGES
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO
pets_app;

-- Grant sequence privileges (for auto-increment primary keys)
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO pets_app;

-- Set default privileges for future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO pets_app;

ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT USAGE, SELECT ON SEQUENCES TO pets_app;

-- Exit
\q
```

Step 3: Configure Authentication (pg_hba.conf)

Location of pg_hba.conf:

- macOS (Homebrew): `/opt/homebrew/var/postgresql@15/pg_hba.conf`
- Linux: `/etc/postgresql/15/main/pg_hba.conf`

Find location:

```
# Connect as postgres and check
psql -U postgres -c "SHOW hba_file;"
```

Edit pg_hba.conf:

```
# macOS
nano /opt/homebrew/var/postgresql@15/pg_hba.conf

# Linux
sudo nano /etc/postgresql/15/main/pg_hba.conf
```

Recommended configuration for development:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# Local connections (Unix socket)
```

```
local   all             postgres                                peer
local   all             all                                     scram-sha-
256

# IPv4 local connections
host    all             all             127.0.0.1/32            scram-sha-
256

# IPv6 local connections
host    all             all             ::1/128                 scram-sha-
256
```

For production (more restrictive):

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# Superuser only via Unix socket
local   all             postgres                                peer

# Application user to pets_db only
local   pets_db         pets_app                                scram-sha-
256
host    pets_db         pets_app        127.0.0.1/32            scram-sha-
256

# Deny all other connections
local   all             all                                     reject
host    all             all             0.0.0.0/0               reject
```

Reload configuration:

```
# macOS
brew services restart postgresql@15

# Linux
sudo systemctl reload postgresql
```

Step 4: Test Authentication

```
# Test connection as pets_app user
psql -U pets_app -d pets_db -h localhost

# You'll be prompted for password: petsAppPassword456!

# Once connected, verify:
SELECT current_database(), current_user;
```

```
# Should show: pets_db | pets_app

# Exit
\q
```

Part 3: Creating the Pets Database Schema

SQL Schema for PostgreSQL

Create `setup_database.sql`:

```
-- Drop tables if they exist (for clean setup)
DROP TABLE IF EXISTS pet CASCADE;
DROP TABLE IF EXISTS kind CASCADE;

-- Create kind table (pet types)
CREATE TABLE kind (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    food VARCHAR(100),
    sound VARCHAR(50)
);

-- Create pet table with foreign key to kind
CREATE TABLE pet (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    kind_id INTEGER NOT NULL,
    age INTEGER CHECK (age >= 0),
    owner VARCHAR(100),
    CONSTRAINT fk_kind
        FOREIGN KEY (kind_id)
        REFERENCES kind(id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
);

-- Create indexes for better query performance
CREATE INDEX idx_pet_kind_id ON pet(kind_id);
CREATE INDEX idx_pet_owner ON pet(owner);

-- Insert sample pet kinds
INSERT INTO kind (name, food, sound) VALUES
    ('dog', 'dogfood', 'bark'),
    ('cat', 'catfood', 'meow'),
    ('bird', 'seeds', 'chirp'),
    ('fish', 'flakes', 'bubble');

-- Insert sample pets
INSERT INTO pet (name, kind_id, age, owner) VALUES
```

```
    ('Dorothy', 1, 9, 'Greg'),
    ('Suzy', 1, 9, 'Greg'),
    ('Casey', 2, 9, 'Greg'),
    ('Heidi', 2, 15, 'David'),
    ('Tweety', 3, 2, 'Alice'),
    ('Nemo', 4, 1, 'Bob');
```

Key Differences from SQLite

1. **SERIAL vs AUTOINCREMENT**: PostgreSQL uses `SERIAL` for auto-incrementing primary keys
2. **VARCHAR instead of TEXT**: More explicit size constraints
3. **CHECK constraints**: Age validation at database level
4. **UNIQUE constraint**: Prevent duplicate kind names
5. **Indexes**: Explicit index creation for performance
6. **CASCADE options**: More control over foreign key behavior

Load the Schema

```
# As pets_app user
psql -U pets_app -d pets_db -h localhost -f setup_database.sql

# Or as postgres superuser initially
psql -U postgres -d pets_db -f setup_database.sql

# Then grant permissions to pets_app
psql -U postgres -d pets_db -c "GRANT SELECT, INSERT, UPDATE, DELETE ON
ALL TABLES IN SCHEMA public TO pets_app;"
psql -U postgres -d pets_db -c "GRANT USAGE, SELECT ON ALL SEQUENCES IN
SCHEMA public TO pets_app;"
```

Verify the Setup

```
psql -U pets_app -d pets_db -h localhost
```

```
-- List tables
\dt

-- Describe tables
\d kind
\d pet

-- Query data
SELECT * FROM kind;
SELECT * FROM pet;

-- Test join (like Topic 4)
```

```
SELECT
    pet.id,
    pet.name,
    pet.age,
    pet.owner,
    kind.name AS kind_name,
    kind.food,
    kind.sound
FROM pet
JOIN kind ON pet.kind_id = kind.id;
```

Part 4: Python Application with PostgreSQL

Install Required Package

```
pip install psycopg2-binary
# or for production:
pip install psycopg2
```

Database Connection Module (database.py)

```
import psycopg2
from psycopg2.extras import RealDictCursor

connection = None

def initialize(host='localhost', database='pets_db', user='pets_app',
password='petsAppPassword456!'):
    """Initialize PostgreSQL connection"""
    global connection
    connection = psycopg2.connect(
        host=host,
        database=database,
        user=user,
        password=password,
        cursor_factory=RealDictCursor  # Return rows as dictionaries
    )
    # PostgreSQL uses autocommit=False by default (transactions)
    connection.autocommit = False

def get_pets():
    """Retrieve all pets with kind information"""
    cursor = connection.cursor()
    cursor.execute("""
        SELECT
            pet.id,
            pet.name,
            pet.age,
```

```
            pet.owner,
            kind.name as kind_name,
            kind.food,
            kind.sound
        FROM pet
        JOIN kind ON pet.kind_id = kind.id
        ORDER BY pet.name
    """)
    pets = cursor.fetchall()
    cursor.close()
    return pets

def get_kinds():
    """Retrieve all pet kinds"""
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM kind ORDER BY name")
    kinds = cursor.fetchall()
    cursor.close()
    return kinds

def get_pet(id):
    """Retrieve a single pet by ID"""
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM pet WHERE id = %s", (id,))
    pet = cursor.fetchone()
    cursor.close()
    return pet if pet else {}

def get_kind(id):
    """Retrieve a single kind by ID"""
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM kind WHERE id = %s", (id,))
    kind = cursor.fetchone()
    cursor.close()
    return kind if kind else {}

def create_pet(data):
    """Create a new pet"""
    try:
        data["age"] = int(data.get("age", 0))
    except (ValueError, TypeError):
        data["age"] = 0

    cursor = connection.cursor()
    cursor.execute(
        """INSERT INTO pet (name, age, kind_id, owner)
           VALUES (%s, %s, %s, %s)""",
        (data["name"], data["age"], data["kind_id"], data["owner"])
    )
    connection.commit()
    cursor.close()

def create_kind(data):
    """Create a new kind"""
```

```
    cursor = connection.cursor()
    cursor.execute(
        """INSERT INTO kind (name, food, sound)
           VALUES (%s, %s, %s)""",
        (data["name"], data["food"], data["sound"])
    )
    connection.commit()
    cursor.close()

def update_pet(id, data):
    """Update an existing pet"""
    try:
        data["age"] = int(data.get("age", 0))
    except (ValueError, TypeError):
        data["age"] = 0

    cursor = connection.cursor()
    cursor.execute(
        """UPDATE pet
           SET name = %s, age = %s, kind_id = %s, owner = %s
           WHERE id = %s""",
        (data["name"], data["age"], data["kind_id"], data["owner"], id)
    )
    connection.commit()
    cursor.close()

def update_kind(id, data):
    """Update an existing kind"""
    cursor = connection.cursor()
    cursor.execute(
        """UPDATE kind
           SET name = %s, food = %s, sound = %s
           WHERE id = %s""",
        (data["name"], data["food"], data["sound"], id)
    )
    connection.commit()
    cursor.close()

def delete_pet(id):
    """Delete a pet"""
    cursor = connection.cursor()
    cursor.execute("DELETE FROM pet WHERE id = %s", (id,))
    connection.commit()
    cursor.close()

def delete_kind(id):
    """Delete a kind (will fail if pets reference it due to foreign
key)"""
    cursor = connection.cursor()
    try:
        cursor.execute("DELETE FROM kind WHERE id = %s", (id,))
        connection.commit()
    except psycopg2.IntegrityError as e:
        connection.rollback()
```

```
            raise Exception(f"Cannot delete kind: pets still reference it")
    from e
        finally:
            cursor.close()

    # Testing functions
    def setup_test_database():
        """Set up test database with sample data"""
        initialize(database='test_pets_db')

        cursor = connection.cursor()

        # Drop and recreate tables
        cursor.execute("DROP TABLE IF EXISTS pet CASCADE")
        cursor.execute("DROP TABLE IF EXISTS kind CASCADE")

        cursor.execute("""
            CREATE TABLE kind (
                id SERIAL PRIMARY KEY,
                name VARCHAR(50) NOT NULL,
                food VARCHAR(100),
                sound VARCHAR(50)
            )
        """)

        cursor.execute("""
            CREATE TABLE pet (
                id SERIAL PRIMARY KEY,
                name VARCHAR(100) NOT NULL,
                kind_id INTEGER NOT NULL,
                age INTEGER,
                owner VARCHAR(100),
                FOREIGN KEY (kind_id) REFERENCES kind(id) ON DELETE RESTRICT
            )
        """)

        connection.commit()

        # Insert test data
        cursor.execute(
            "INSERT INTO kind (name, food, sound) VALUES (%s, %s, %s)",
            ("dog", "dogfood", "bark")
        )
        cursor.execute(
            "INSERT INTO kind (name, food, sound) VALUES (%s, %s, %s)",
            ("cat", "catfood", "meow")
        )
        connection.commit()

        pets = [
            {"name": "dorothy", "kind_id": 1, "age": 9, "owner": "greg"},
            {"name": "suzy", "kind_id": 1, "age": 9, "owner": "greg"},
            {"name": "casey", "kind_id": 2, "age": 9, "owner": "greg"},
            {"name": "heidi", "kind_id": 2, "age": 15, "owner": "david"},
```

```
    ]

    for pet in pets:
        create_pet(pet)

    cursor.close()

def test_get_pets():
    print("Testing get_pets...")
    pets = get_pets()
    assert type(pets) is list
    assert len(pets) > 0
    assert type(pets[0]) is dict
    pet = pets[0]
    print(f"Sample pet: {pet}")

    required_fields = ["id", "name", "age", "owner", "kind_name"]
    for field in required_fields:
        assert field in pet, f"Field {field} missing from {pet}"

    print("✓ get_pets test passed")

def test_get_kinds():
    print("Testing get_kinds...")
    kinds = get_kinds()
    assert type(kinds) is list
    assert len(kinds) > 0
    assert type(kinds[0]) is dict
    kind = kinds[0]

    required_fields = ["id", "name", "food", "sound"]
    for field in required_fields:
        assert field in kind, f"Field {field} missing from {kind}"

    print("✓ get_kinds test passed")

if __name__ == "__main__":
    # Note: You need to create test_pets_db first:
    # psql -U postgres -c "CREATE DATABASE test_pets_db;"
    # psql -U postgres -c "GRANT ALL PRIVILEGES ON DATABASE test_pets_db
TO pets_app;"

    setup_test_database()
    test_get_pets()
    test_get_kinds()
    print("All tests passed!")
```

Key Differences from SQLite Version

1. **Parameter placeholders**: PostgreSQL uses `%s` instead of `?`
2. **RealDictCursor**: Returns rows as dictionaries (like SQLite's row_factory)
3. **Explicit commits**: PostgreSQL uses transactions by default

4. **Rollback on error**: Better error handling with try/except/finally
5. **Connection parameters**: Host, user, password instead of file path
6. **Foreign key errors**: More informative IntegrityError exceptions

---

Part 5: Flask Application (app.py)

The Flask application remains mostly unchanged - just update the initialization:

```
from flask import Flask, render_template, request, redirect, url_for
import os
import database

app = Flask(__name__)

# Initialize with environment variables (secure!)
database.initialize(
    host=os.environ.get('POSTGRES_HOST', 'localhost'),
    database=os.environ.get('POSTGRES_DB', 'pets_db'),
    user=os.environ.get('POSTGRES_USER', 'pets_app'),
    password=os.environ.get('POSTGRES_PASSWORD', 'petsAppPassword456!')
)

# Routes remain the same as Topic 4...
@app.route("/", methods=["GET"])
@app.route("/list", methods=["GET"])
def get_list():
    pets = database.get_pets()
    return render_template("list.html", pets=pets)

# ... (all other routes from Topic 4)
```

Running the Application

```
# Set environment variables (more secure than hardcoding)
export POSTGRES_HOST=localhost
export POSTGRES_DB=pets_db
export POSTGRES_USER=pets_app
export POSTGRES_PASSWORD=petsAppPassword456!

# Run Flask app
python app.py
```

Or create a `.env` file (add to `.gitignore`!):

```
POSTGRES_HOST=localhost
POSTGRES_DB=pets_db
```

```
POSTGRES_USER=pets_app
POSTGRES_PASSWORD=petsAppPassword456!
```

Use with `python-dotenv`:

```
from dotenv import load_dotenv
load_dotenv()
```

Part 6: PostgreSQL Security Best Practices

1. User Management and Least Privilege

Create role hierarchy:

```
-- Read-only role for reporting
CREATE ROLE pets_readonly;
GRANT CONNECT ON DATABASE pets_db TO pets_readonly;
GRANT USAGE ON SCHEMA public TO pets_readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO pets_readonly;

-- Create read-only user
CREATE USER reporting_user WITH PASSWORD 'reportPass789!';
GRANT pets_readonly TO reporting_user;

-- Application role with full CRUD
CREATE ROLE pets_crud;
GRANT CONNECT ON DATABASE pets_db TO pets_crud;
GRANT USAGE ON SCHEMA public TO pets_crud;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO
pets_crud;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO pets_crud;

-- Create application user
CREATE USER pets_app WITH PASSWORD 'petsAppPassword456!';
GRANT pets_crud TO pets_app;
```

2. SSL/TLS Encryption

Enable SSL in postgresql.conf:

```
ssl = on
ssl_cert_file = '/path/to/server.crt'
ssl_key_file = '/path/to/server.key'
ssl_ca_file = '/path/to/root.crt'
```

Require SSL in pg_hba.conf:

```
hostssl    pets_db    pets_app    0.0.0.0/0    scram-sha-256
```

Connect with SSL in Python:

```
connection = psycopg2.connect(
    host='localhost',
    database='pets_db',
    user='pets_app',
    password='petsAppPassword456!',
    sslmode='require'  # or 'verify-full' for certificate verification
)
```

3. Connection Pooling

For production, use connection pooling to manage database connections efficiently:

```
from psycopg2 import pool

connection_pool = None

def initialize_pool(minconn=1, maxconn=10):
    global connection_pool
    connection_pool = psycopg2.pool.SimpleConnectionPool(
        minconn,
        maxconn,
        host='localhost',
        database='pets_db',
        user='pets_app',
        password='petsAppPassword456!'
    )

def get_connection():
    return connection_pool.getconn()

def release_connection(conn):
    connection_pool.putconn(conn)
```

4. Audit Logging

Enable logging in `postgresql.conf`:

```
logging_collector = on
log_directory = 'pg_log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

```
log_connections = on
log_disconnections = on
log_duration = on
log_statement = 'all'  # or 'ddl', 'mod' for less verbosity
```

5. Password Security

Store passwords securely:

```
import os
from psycopg2 import connect

def get_connection():
    return connect(
        host=os.environ['POSTGRES_HOST'],
        database=os.environ['POSTGRES_DB'],
        user=os.environ['POSTGRES_USER'],
        password=os.environ['POSTGRES_PASSWORD']
    )
```

Never:

- Hardcode passwords in source code
- Commit passwords to version control
- Log passwords
- Display passwords in error messages

6. SQL Injection Prevention

PostgreSQL's parameter binding prevents SQL injection:

```
# ✓ SAFE: Parameterized query
cursor.execute("SELECT * FROM pet WHERE owner = %s", (owner,))

# ✗ DANGEROUS: String concatenation
cursor.execute(f"SELECT * FROM pet WHERE owner = '{owner}'")  # DON'T DO
THIS!
```

7. Regular Updates

```
# macOS
brew upgrade postgresql@15

# Linux
sudo apt update
sudo apt upgrade postgresql
```

Part 7: PostgreSQL vs SQLite Comparison

Migration Checklist

When converting from SQLite to PostgreSQL:

- [ ] Change `AUTOINCREMENT` to `SERIAL`
- [ ] Change `INTEGER PRIMARY KEY` to `SERIAL PRIMARY KEY`
- [ ] Change `TEXT` to `VARCHAR(n)` where appropriate
- [ ] Update parameter placeholders: `?` → `%s`
- [ ] Add explicit `connection.commit()` calls
- [ ] Handle transactions and rollbacks
- [ ] Update connection initialization (file → host/user/password)
- [ ] Add environment variables for credentials
- [ ] Test foreign key constraints (behavior may differ)
- [ ] Update boolean handling (SQLite: 0/1, PostgreSQL: true/false)
- [ ] Review date/time handling (PostgreSQL has native types)

When to Use Each

Use SQLite when:

- Prototyping or learning
- Embedded applications (mobile, desktop)
- Single-user applications
- Simple data storage needs
- No network access required
- Small datasets (< 1GB)

Use PostgreSQL when:

- Multi-user applications
- Production web applications
- Need concurrent writes
- Complex queries and joins
- Need advanced features (JSON, full-text search, GIS)
- Large datasets (> 1GB)
- Security requirements (user authentication, encryption)
- Scalability and performance critical

Part 8: Common PostgreSQL Commands

Database Management

```
-- List all databases
\l
```

```
-- Connect to database
\c pets_db

-- List all tables
\dt

-- Describe table structure
\d pet
\d+ pet  -- More details

-- List all users/roles
\du

-- Show current user
SELECT current_user;

-- Show current database
SELECT current_database();
```

User and Permission Management

```
-- Create user
CREATE USER newuser WITH PASSWORD 'password';

-- Grant privileges
GRANT ALL PRIVILEGES ON DATABASE mydb TO newuser;

-- Revoke privileges
REVOKE ALL PRIVILEGES ON DATABASE mydb FROM newuser;

-- Change user password
ALTER USER pets_app WITH PASSWORD 'newPassword789!';

-- Drop user
DROP USER username;

-- View table permissions
\dp tablename
```

Backup and Restore

```
# Backup database
pg_dump -U postgres pets_db > pets_db_backup.sql

# Backup with custom format (compressed)
pg_dump -U postgres -Fc pets_db > pets_db_backup.dump

# Restore from SQL file
```

```
psql -U postgres pets_db < pets_db_backup.sql

# Restore from custom format
pg_restore -U postgres -d pets_db pets_db_backup.dump

# Backup single table
pg_dump -U postgres -t pet pets_db > pet_table_backup.sql
```

Part 9: Troubleshooting

Connection Issues

Error: "FATAL: role "username" does not exist"

```
-- Create the missing user
CREATE USER username WITH PASSWORD 'password';
```

Error: "FATAL: database "dbname" does not exist"

```
createdb -U postgres dbname
```

Error: "FATAL: Peer authentication failed"

- Edit `pg_hba.conf` and change `peer` to `scram-sha-256`
- Reload PostgreSQL

Error: "connection refused"

```
# Check if PostgreSQL is running
# macOS:
brew services list | grep postgresql

# Linux:
sudo systemctl status postgresql

# Start if not running:
brew services start postgresql@15  # macOS
sudo systemctl start postgresql    # Linux
```

Permission Issues

Error: "permission denied for table"

```

```

```
-- Grant permissions as postgres superuser
GRANT SELECT, INSERT, UPDATE, DELETE ON tablename TO username;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO username;
```

Error: "must be owner of table"

```
-- Transfer table ownership
ALTER TABLE tablename OWNER TO username;
```

Performance Issues

```
-- View slow queries
SELECT * FROM pg_stat_activity WHERE state = 'active';

-- View table sizes
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS
size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;

-- Analyze query performance
EXPLAIN ANALYZE SELECT * FROM pet JOIN kind ON pet.kind_id = kind.id;
```

Key Takeaways

1. **PostgreSQL is production-grade** - Use it for real applications, not just learning
2. **Security from the start** - Always use authentication, even in development
3. **Least privilege principle** - Create specific users for specific purposes
4. **Environment variables** - Never hardcode credentials
5. **Parameter binding** - Always use `%s` placeholders to prevent SQL injection
6. **Transactions matter** - Explicit commit/rollback for data integrity
7. **SSL/TLS in production** - Encrypt database connections
8. **Regular backups** - Use `pg_dump` and test restoration
9. **Monitor and audit** - Enable logging and review regularly
10. **Keep PostgreSQL updated** - Apply security patches promptly

Understanding PostgreSQL security and proper configuration prepares you for real-world application development and demonstrates professional database management skills to employers.

Additional Resources

- **Official Documentation**: postgresql.org/docs
- **Security Guide**: postgresql.org/docs/current/security.html
- **psycopg2 Documentation**: psycopg.org
- **PgAdmin** (GUI tool): pgadmin.org