# **How To Install and Use PostgreSQL on Ubuntu 12.04**

## 

## Prerequisites a regular, non-root user with sudo privileges configured on server. OS - Ubuntu 12.04

## Packages required: postgresql

## postgresql-contrib

## 

## Install Postgres

# apt-get update

# sudo apt-get install postgresql postgresql-contrib

## Create Your PostgreSQL Roles and Databases

# Postgres uses the concept of roles to distinguish the variety of users that can connect to a database.

# The default postgres user is actual named “postgres”.

# The postgres default is to use *ident* authentication

# Configuration file : **/etc/postgresql/9.1/main/pg\_hba.conf**

# To begin creating custom users, first switch into the default user:

# sudo su – postgres

# createuser

# To outfit your user with a password:

# createuser --pwprompt

## Connecting to the Postgres Databases

# You can create the Postgres database as a superuser. In this case, we will use the default super user.

# switch into the postgres user:

# su – postgres

# create database:

# createdb events

# And with that you can finally connect to the postgres shell.

## 

## How to Create and Delete a Postgres Tables

## To login in database

## psql -d <db\_name>

# Login in to database

# To create a new Postgres table:(E.g.)

# CREATE TABLE potluck (name VARCHAR(20), food VARCHAR(30), confirmed CHAR(1), signup\_date DATE);

# This command accomplishes a number of things:

# It has created a table called potluck within the database, newdb.

# We have set up 4 columns in the table—name, food, confirmed, and signup date.

# The “name” column has been limited by the VARCHAR command to be under 20 characters long.

# The “food” column designates the food each person will bring. The VARCHAR limits text to be under 30 characters.

# The “confirmed” column records whether the person has RSVP’d with one letter, Y or N.

# The “date” column will show when they signed up for the event. Postgres requires that dates be written as yyyy-mm-dd

# 

# You can additionally see all of the tables within the database with the following command:

# \dt

## How to Add Information to a Postgres Table

# To insert:

# INSERT INTO potluck (name, food, confirmed, signup\_date) VALUES('John', 'Casserole', 'Y', '2012-04-11');

# Show table:

# SELECT \* FROM potluck; Delete row :

# DELETE FROM potluck WHERE name = 'John' ;

## How to Add and Delete a Column

# ALTER TABLE potluck ADD email VARCHAR(40);

# This command puts the new column called "email" at the end of the table by default, and the VARCHAR command limits it to 40 characters.

# To delete l:

# ALTER TABLE potluck DROP email;

## How to Update Information in the Table

# E.g. UPDATE potluck set confirmed = 'Y' WHERE name = 'Sandy';

# 

# 

# **How To Use Roles and Manage Grant Permissions in PostgreSQL on a VPS**

sudo su - postgres

## PostgreSQL Permission Concepts

PostgreSQL manages permissions through the concept of "roles".

Roles are different from traditional Unix-style permissions in that there is no distinction between users and groups. Roles can be manipulated to resemble both of these conventions, but they are also more flexible.

For instance, roles can be members of other roles, allowing them to take on the permission characteristics of previously defined roles. Roles can also own objects and control access to those object for other roles.

### How to View Roles in PostgreSQL

Login to promt interface:

psql

To get a list of roles, type this:

\du

## How to Create Roles in PostgreSQL

There are a number of different ways to create roles for Postgres. It is possible to create roles from within Postgres, or from the command line.

### How to Create Roles From Within PostgreSQL

The most basic way of creating new roles is from within the Postgres prompt interface.

You can create a new role with the following syntax:

CREATE ROLE new\_role\_name;

Let's create a new role called "demo\_role":

CREATE ROLE demo\_role;

CREATE ROLE

To check:

\du

### How to Create Roles from the Command Line

An alternative method of creating roles is using the "createuser" command.

Get out of the PostgreSQL command prompt for a moment by typing:

\q

Create a role called "test\_user" with the following command:

createuser test\_user

## How to Delete Roles In PostgreSQL

To delete a role:

DROP ROLE role\_name;

## How to Define Privileges Upon Role Creation

Now, we are ready to recreate the "demo\_role" role with altered permissions. We can do this by specifying the permissions we want after the main create clause:

CREATE ROLE role\_name WITH optional\_permissions;

You can see a full list of the options by typing:

\h CREATE ROLE

We want to give this user the ability to log in, so we will type:

CREATE ROLE demo\_role WITH LOGIN;

CREATE ROLE

If we check the attributes again, we can see that the two users now have identical privileges:

\du

List of roles  
 Role name | Attributes | Member of   
-----------+------------------------------------------------+-----------  
 demo\_role | | {}  
 postgres | Superuser, Create role, Create DB, Replication | {}  
 test\_user | | {}

If we want to get to this state without specifying the "login" attribute with every role creation, we can actually use the following command instead of the "CREATE ROLE" command:

CREATE USER role\_name;

The only difference between the two commands is that "CREATE USER" automatically gives the role login privileges.

## How to Change Privileges of Roles in PostgreSQL

To change the attributes of an already created role, we use the "ALTER ROLE" command.

This command allows us to define privilege changes without having to delete and recreate users as we demonstrated earlier.

The basic syntax is:

ALTER ROLE role\_name WITH attribute\_options;

For instance, we can change "demo\_role" back to its previous state by issuing this command:

ALTER ROLE demo\_role WITH NOLOGIN;

ALTER ROLE

We can see the privileges have reverted to their previous state:

\du

List of roles  
 Role name | Attributes | Member of   
-----------+------------------------------------------------+-----------  
 demo\_role | Cannot login | {}  
 postgres | Superuser, Create role, Create DB, Replication | {}  
 test\_user | | {}

We can easily change it back with the following command:

ALTER ROLE demo\_role WITH LOGIN;

## How to Log In as a Different User in PostgreSQL

In **postgresql.conf** file make an entry for

listen\_addresses='\*'

By default, users are only allowed to login locally if the system username matches the PostgreSQL username.

We can get around this by either changing the login type, or by specifying that PostgreSQL should use the loopback network interface, which would change the connection type to remote, even though it is actually a local connection.

We will discuss the second option. First, we need to give the user we'd like to connect as a password so that we can authenticate.

Give the "test\_user" a password with the following command:

\password test\_user

You will be prompted to enter and confirm a password. Now, exit the PostgreSQL interface and exit back to your normal user.

\q  
exit

PostgreSQL assumes that when you log in, you will be using a username that matches your operating system username, and that you will be connecting to a database with the same name as well.

This is not the case with the situation we are demonstrating, so we will need to explicitly specify the options we want to use. Use the following syntax:

psql -U user\_name -d database\_name -h 127.0.0.1 -W

The "user\_name" should be replaced with the username we want to connect with. Similarly, the "database\_name" should be the name of an existing database that you have access to.

The "-h 127.0.0.1" section is the part that specifies that we will be connecting to the local machine, but through a network interface, which allows us to authenticate even though our system username does not match. The "-W" flag tells PostgreSQL that we will be entering a password.

To log in with our "test\_user" we can issue the following command:

psql -U test\_user -d postgres -h 127.0.0.1 -W

Password for user test\_user:

You will need to enter the password you configured. In our example, we use the database "postgres". This is the default database set up during install.

If you attempt to perform some actions in this session, you will see that you don't have the ability to do many things. This is because we did not give "test\_user" permissions to administer many things.

Let's exit and get back into the administrative session:

\q  
sudo su - postgres  
psql

## How to Grant Permissions in PostgreSQL

When a database or table is created, usually only the role that created it (not including roles with superuser status) has permission to modify it. We can alter this behavior by granting permissions to other roles.

We can grant permissions using the "GRANT" command. The general syntax is here:

GRANT permission\_type ON table\_name TO role\_name;

Create a simple table to practice these concepts:

CREATE TABLE demo (  
name varchar(25),  
id serial,  
start\_date date);

NOTICE: CREATE TABLE will create implicit sequence "demo\_id\_seq" for serial column "demo.id"  
CREATE TABLE

We can see the result with:

\d

List of relations  
 Schema | Name | Type | Owner   
--------+-------------+----------+----------  
 public | demo | table | postgres  
 public | demo\_id\_seq | sequence | postgres  
(2 rows)

We can now grant some privileges to the new "demo" table to "demo\_role". Give the user "UPDATE" privileges with the following command:

GRANT UPDATE ON demo TO demo\_role;

We can grant full permissions to a user by substituting the permission type with the word "all":

GRANT ALL ON demo TO test\_user;

If we want to specify permissions for every user on the system, we can use the word "public" instead of a specific user:

GRANT INSERT ON demo TO PUBLIC;

To view the grant table, use the following command:

\z

Access privileges  
 Schema | Name | Type | Access privileges | Column access privileges   
--------+-------------+----------+----------------------------+--------------------------  
 public | demo | table | postgres=arwdDxt/postgres +|   
 | | | demo\_role=w/postgres +|   
 | | | test\_user=arwdDxt/postgres+|   
 | | | =a/postgres |   
 public | demo\_id\_seq | sequence | |   
(2 rows)

This shows all of the grant permissions we have just assigned.

## How to Remove Permissions in PostgreSQL

You can remove permissions using the "REVOKE" command. The revoke command uses almost the same syntax as grant:

REVOKE permission\_type ON table\_name FROM user\_name;

Again, we can use the same shorthand words (all and public) to make the command easier:

REVOKE INSERT ON demo FROM PUBLIC;

## How to Use Group Roles in PostgreSQL

Roles are flexible enough to allow grouping of other roles to allow for widespread permissions control.

For instance, we can create a new role called "temporary\_users" and then add "demo\_role" and "test\_user" to that role:

CREATE ROLE temporary\_users;  
GRANT temporary\_users TO demo\_role;  
GRANT temporary\_users TO test\_user;

Now these two users can have their permissions managed by manipulating the "temporary\_users" group role instead of managing each member individually.

We can see the role membership information by typing:

\du

List of roles  
 Role name | Attributes | Member of   
-----------------+------------------------------------------------+-------------------  
 demo\_role | | {temporary\_users}  
 postgres | Superuser, Create role, Create DB, Replication | {}  
 temporary\_users | Cannot login | {}  
 test\_user | | {temporary\_users}

Any member of a group role can act as the group role they are a member of by using the "set role" command.

Since the "postgres" user we are logged into currently has superuser privileges, we can use "set role" even though we are not a member of that group:

SET ROLE temporary\_users;

Now, any tables that are created are owned by the temporary\_users role:

CREATE TABLE hello (  
name varchar(25),  
id serial,  
start\_date date);

We can check the table ownership by issuing this command:

\d

List of relations  
 Schema | Name | Type | Owner   
--------+--------------+----------+-----------------  
 public | demo | table | postgres  
 public | demo\_id\_seq | sequence | postgres  
 public | hello | table | temporary\_users  
 public | hello\_id\_seq | sequence | temporary\_users  
(4 rows)

As you can see, the new table (and the sequence associated with the serial data type) is owned by the "temporary\_users" role.

We can get back to our original role permissions with the following command:

RESET ROLE;

If we give a user the "inherit" property with the "alter role" command, that user will automatically have all of the privileges of the roles they belong to without using the "set role" command:

ALTER ROLE test\_user INHERIT;

Now test\_user will have every permission of the roles it is a member of.

We can remove a group role (or any role) with the "drop role" command:

DROP ROLE temporary\_users;

ERROR: role "temporary\_users" cannot be dropped because some objects depend on it  
DETAIL: owner of table hello  
owner of sequence hello\_id\_seq

This will give you an error, because we created a table that is owned by "temporary\_users". We can solve this problem by transferring ownership to a different role:

ALTER TABLE hello OWNER TO demo\_role;

If we check, we can see that "temporary\_users" no longer owns any of the tables:

\d

List of relations  
 Schema | Name | Type | Owner   
--------+--------------+----------+-----------  
 public | demo | table | postgres  
 public | demo\_id\_seq | sequence | postgres  
 public | hello | table | demo\_role  
 public | hello\_id\_seq | sequence | demo\_role  
(4 rows)

We can now drop the "temporary\_users" role successfully by issuing the command again:

DROP ROLE temporary\_users;

This will destroy the temporary\_users role. The former members of temporary\_users are not removed.

## Conclusion

You should now have the basic skills necessary to administer your PostgreSQL database permissions. It is important to know how to manage permissions so that your applications can access the databases they need, while not disrupting data used by other applications.

**reset the password for user name "postgres"** :

1. In *pg\_hba.conf*, insert or change the below line.

*from :*

local all postgres

*to*

local all postgres trust sameuser

2. Restart *PostgreSQL* services in order for **Step 1** changes to take effect :

* In [Linux](http://psybermonkey.blogspot.com/search/label/Linux),
* /etc/init.d/postgresql-8.3 restart
* In [FreeBSD](http://psybermonkey.blogspot.com/search/label/FreeBSD),
* /usr/local/etc/rc.d/postgres restart

3. Login to *PostgreSQL* on the local machine with the user name "*postgres*" to change the password :

e.g.

psql -U postgres

4. At the "*postgres=#*" prompt, change the user name "*postgres*" password :

e.g.

ALTER USER postgres with password 'secure-password';

5. Quit *PostgreSQL* interactive session by executing "*\q*", to exit

6. Alter the configuration (what we did in **Step 1**) to disable password-less login from local machine to *PostgreSQL* by changing the word "*trust*" to "*md5*" in *pg\_hba.conf*.

e.g.

*from:*

local all postgres trust sameuser

*to:*

local all postgres md5 sameuser

7. Restart *PostgreSQL* to make **Step 6** changes take effect by repeating **Step 2**.

8. Re-login to *PostgreSQL* using the new password by :

psql -U postgres

**NOTES :**

\*\*Grant permission to create db to user.

=> ALTER USER myusername CREATEDB

\*\*Config file : /etc/postgresql/9.3/main/pg\_hba.conf

TYPE DATABASE USER ADDRESS METHOD