By: Justin Ellingwood

How To Install and Use PostgreSQL on Ubuntu 16.04

Posted May 4, 2016 247.8k POSTGRESQL UBUNTU UBUNTU 16.04

Introduction

Relational database management systems are a key component of many web sites and applications. They provide a structured way to store, organize, and access information.

PostgreSQL, or Postgres, is a relational database management system that provides an implementation of the SQL querying language. It is a popular choice for many small and large projects and has the advantage of being standards-compliant and having many advanced features like reliable transactions and concurrency without read locks.

In this guide, we will demonstrate how to install Postgres on an Ubuntu 16.04 VPS instance and go over some basic ways to use it.

Installation

Ubuntu's default repositories contain Postgres packages, so we can install these easily using the `apt` packaging system.

Since this is our first time using `apt` in this session, we need to refresh our local package index. We can then install the Postgres package and a `-contrib` package that adds some additional utilities and functionality:

```
$ sudo apt-get update
$ sudo apt-get install postgresql postgresql-contrib
```

Now that our software is installed, we can go over how it works and how it may be different from similar database management systems you may have used.

Using PostgreSQL Roles and Databases

By default, Postgres uses a concept called "roles" to handle in authentication and authorization. These are, in some ways, similar to regular Unix-style accounts, but Postgres does not distinguish between users and groups and instead prefers the more flexible term "role".

Upon installation Postgres is set up to use **ident** authentication, which means that it associates Postgres roles with a matching Unix/Linux system account. If a role exists within Postgres, a Unix/Linux username with the same name will be able to sign in as that role.

There are a few ways to utilize this account to access Postgres.

Switching Over to the postgres Account

The installation procedure created a user account called `postgres` that is associated with the default Postgres role. In order to use Postgres, we can log into that account.

Switch over to the `postgres` account on your server by typing:

```
$ sudo -i -u postgres
```

You can now access a Postgres prompt immediately by typing:

```
$ psql
```

You will be logged in and able to interact with the database management system right away.

Exit out of the PostgreSQL prompt by typing:

```
postgres=# \q
```


You can also run the command you'd like with the `postgres` account directly with `sudo`.

For instance, in the last example, we just wanted to get to a Postgres prompt. We could do this in one step by running the single command `psql` as the `postgres` user with `sudo` like this:

```
$ sudo -u postgres psql
```

This will log you directly into Postgres without the intermediary `bash` shell in between.

Again, you can exit the interactive Postgres session by typing:

```
postgres=# \q
```

Create a New Role

Currently, we just have the `postgres` role configured within the database. We can create new roles from the command line with the `createrole` command. The `--interactive` flag will prompt you for the necessary values.

If you are logged in as the `postgres` account, you can create a new user by typing:

```
postgres@server:~$ createuser --interactive
```

If, instead, you prefer to use `sudo` for each command without switching from your normal account, you can type:

```
$ sudo -u postgres createuser --interactive
```

The script will prompt you with some choices and, based on your responses, execute the correct Postgres commands to create a user to your specifications.

```
Output
Enter name of role to add: sammy
Shall the new role be a superuser? (y/n) y
```

You can get more control by passing some additional flags. Check out the options by looking at the `man` page:

```
$ man createuser
```

Create a New Database

By default, another assumption that the Postgres authentication system makes is that there will be an database with the same name as the role being used to login, which the role has access to.

So if in the last section, we created a user called `sammy`, that role will attempt to connect to a database which is also called `sammy` by default. You can create the appropriate database with the `createdb` command.

If you are logged in as the `postgres` account, you would type something like:

```
postgres@server:~$ createdb sammy
```

If, instead, you prefer to use `sudo` for each command without switching from your normal account, you would type:

```
$ sudo -u postgres createdb sammy
```

Open a Postgres Prompt with the New Role

me as your Postgres role and database.

If you don't have a matching Linux user available, you can create one with the `adduser` command. You will have to do this from an account with `sudo` privileges (not logged in as the `postgres` user):

```
$ sudo adduser sammy
```

Once you have the appropriate account available, you can either switch over and connect to the database by typing:

```
$ sudo -i -u sammy
$ psql
```

Or, you can do this inline:

```
$ sudo -u sammy psql
```

You will be logged in automatically assuming that all of the components have been properly configured.

If you want your user to connect to a different database, you can do so by specifying the database like this:

```
$ psql -d postgres
```

Once logged in, you can get check your current connection information by typing:

```
sammy=# \conninfo
```

```
Output
You are connected to database "sammy" as user "sammy" via socket in "/var/run/postgresql" at port "5432".
```

This can be useful if you are connecting to non-default databases or with non-default users.

Create and Delete Tables

Now that you know how to connect to the PostgreSQL database system, we can to go over how to complete some basic tasks.

First, we can create a table to store some data. Let's create a table that describes playground equipment.

The basic syntax for this command is something like this:

```
CREATE TABLE table_name (
    column_name1 col_type (field_length) column_constraints,
    column_name2 col_type (field_length),
    column_name3 col_type (field_length)
);
```

As you can see, we give the table a name, and then define the columns that we want, as well as the column type and the max length of the field data. We can also optionally add table constraints for each column.

You can learn more about how to create and manage tables in Postgres here.

For our purposes, we're going to create a simple table like this:

```
CREATE TABLE playground (
    equip_id serial PRIMARY KEY,
```

```
    type varchar (50) NOT NULL,
    color varchar (25) NOT NULL,
    location varchar(25) check (location in ('north', 'south', 'west', 'east', 'northeast', 'southeast', 'southwest', 'northw
    install_date date
```


We have made a playground table that inventories the equipment that we have. This starts with an equipment ID, which is of the `serial` type. This data type is an auto-incrementing integer. We have given this column the constraint of `primary key` which means that the values must be unique and not null.

For two of our columns (`equip_id` and `install_date`), we have not given a field length. This is because some column types don't require a set length because the length is implied by the type.

We then give columns for the equipment `type` and `color`, each of which cannot be empty. We create a `location` column and create a constraint that requires the value to be one of eight possible values. The last column is a date column that records the date that we installed the equipment.

We can see our new table by typing:

```
sammy=# \d
```

Output

```
                List of relations
 Schema |          Name           |   Type   | Owner
--------+-------------------------+----------+-------
 public | playground              | table    | sammy
 public | playground_equip_id_seq | sequence | sammy
(2 rows)
```

Our playground table is here, but we also have something called `playground_equip_id_seq` that is of the type `sequence`. This is a representation of the `serial` type we gave our `equip_id` column. This keeps track of the next number in the sequence and is created automatically for columns of this type.

If you want to see just the table without the sequence, you can type:

```
sammy=# \dt
```

Output

```
         List of relations
 Schema |    Name    | Type  | Owner
--------+------------+-------+-------
 public | playground | table | sammy
(1 row)
```

Add, Query, and Delete Data in a Table

Now that we have a table, we can insert some data into it.

Let's add a slide and a swing. We do this by calling the table we're wanting to add to, naming the columns and then providing data for each column. Our slide and swing could be added like this:

```
sammy=# INSERT INTO playground (type, color, location, install_date) VALUES ('slide', 'blue', 'south', '2014-04-28');
sammy=# INSERT INTO playground (type, color, location, install_date) VALUES ('swing', 'yellow', 'northwest', '2010-08-16');
```

You should take care when entering the data to avoid a few common hangups. First, keep in mind that the column names should not be quoted, but the column *values* that you're entering do need quotes.

Another thing to keep in mind is that we do not enter a value for the `equip_id` column. This is because this is auto-generated whenever a new row in the table is created.


```
sammy=# SELECT * FROM playground;
```

```
Output
 equip_id | type  | color  | location  | install_date
----------+-------+--------+-----------+--------------
        1 | slide | blue   | south     | 2014-04-28
        2 | swing | yellow | northwest | 2010-08-16
(2 rows)
```

Here, you can see that our `equip_id` has been filled in successfully and that all of our other data has been organized correctly.

If the slide on the playground breaks and we have to remove it, we can also remove the row from our table by typing:

```
sammy=# DELETE FROM playground WHERE type = 'slide';
```

If we query our table again, we will see our slide is no longer a part of the table:

```
sammy=# SELECT * FROM playground;
```

```
Output
 equip_id | type  | color  | location  | install_date
----------+-------+--------+-----------+--------------
        2 | swing | yellow | northwest | 2010-08-16
(1 row)
```

How To Add and Delete Columns from a Table

If we want to modify a table after it has been created to add an additional column, we can do that easily.

We can add a column to show the last maintenance visit for each piece of equipment by typing:

```
sammy=# ALTER TABLE playground ADD last_maint date;
```

If you view your table information again, you will see the new column has been added (but no data has been entered):

```
sammy=# SELECT * FROM playground;
```

```
Output
 equip_id | type  | color  | location  | install_date | last_maint
----------+-------+--------+-----------+--------------+------------
        2 | swing | yellow | northwest | 2010-08-16   |
(1 row)
```

We can delete a column just as easily. If we find that our work crew uses a separate tool to keep track of maintenance history, we can get rid of the column here by typing:

```
sammy=# ALTER TABLE playground DROP last_maint;
```

How To Update Data in a Table

We know how to add records to a table and how to delete them, but we haven't covered how to modify existing entries yet.

You can update the values of an existing entry by querying for the record you want and setting the column to the value you wish to use. We can query for the "swing" record (this will match *every* swing in our table) and change its color to "red". This could be useful if we gave the swing set a paint job:

We can verify that the operation was successful by querying our data again:

```
sammy=# SELECT * FROM playground;
```

```
Output
 equip_id | type  | color | location  | install_date
----------+-------+-------+-----------+--------------
        2 | swing | red   | northwest | 2010-08-16
(1 row)
```

As you can see, our slide is now registered as being red.

Conclusion

You are now set up with PostgreSQL on your Ubuntu 16.04 server. However, there is still *much* more to learn with Postgres. Here are some more guides that cover how to use Postgres:

- A comparison of relational database management systems
- Learn how to create and manage tables with Postgres
- Get better at managing roles and permissions
- Craft queries with Postgres with Select
- Learn how to secure PostgreSQL
- Learn how to backup a Postgres database

By: Justin Ellingwood

Related Tutorials

How To Customize the PostgreSQL Prompt with psqlrc on Ubuntu 14.04

How To Install Concourse CI on Ubuntu 16.04

How To Secure PostgreSQL Against Automated Attacks

How To Set Up Django with Postgres, Nginx, and Gunicorn on Debian 8

How To Use Postgresql with your Django Application on Debian 8

6 Comments

Leave a comment...

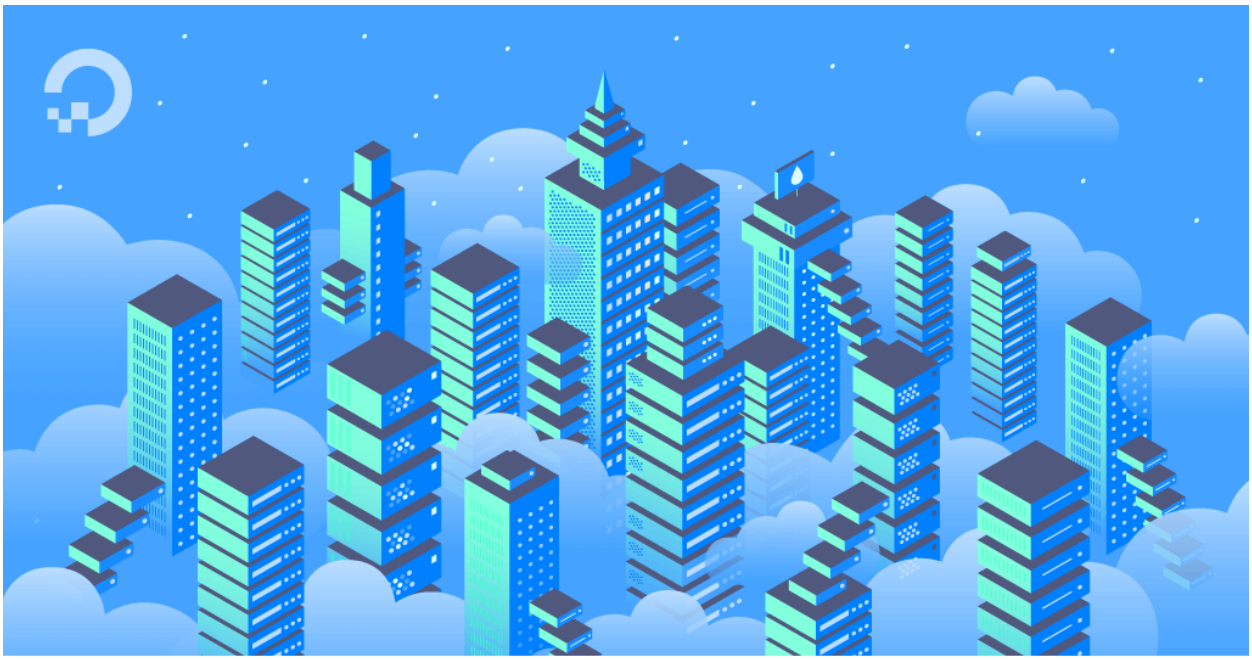
Log In to Comment

jonl316 *June 4, 2016*
0
Wow, thank you so much. This is exactly what I was looking for. Great information to help get started.

sonamhava *August 21, 2016*
0
I want to use the Volume for storing my database storage. Is that possible and some instructions possible?

thanks

jellingwood MOD *August 22, 2016*
0
@sonamhava: Yep! You can find an article on how to do just that here. Hope that helps!



How To Move a PostgreSQL Data Directory to a New Location on Ubuntu 16.04

Whether you're adding more space, evaluating ways to optimize performance, or looking to take advantage of other storage features, this tutorial will guide you through relocating PostgreSQL's data directory.

sonamhava *August 30, 2016*
0
I want to connect from app on droplet A with postgresql on droplet B. I added the private ip entry in postgresql.conf file and restarted pgsql. I am not able to connect from droplet A. I am using the following command: psql -h <IPADDRESS> -p <PORT> -U <USER>.

technotone *October 28, 2016*

0 Firstly, thank you for this post. It's exactly grateful. However, I try to create new role with createrole command but there is no such a command.

-u sammy psql) under the new role, it prompts me for a postgres password which we never created. However, if I simply type 'psql', it logs me in successfully under the new role.

Speaking of passwords, should we create a password for the new database? I'm using an Ubuntu server which has to be ssh'd into. If a mischievous individual managed to get through my ssh, I have much bigger problems to worry about.