Previous Next

Meet PostgreSQL

Estimated Time: 2-3 hours

In this assignment you'll set up a PostgreSQL database to store snippets. Postgres is a high performance database engine, and in recent years has become the go-to database choice for a wide range of applications.

A Postgres database is made up of various different elements. The **database** contains any number of **tables** and the tables contain **rows** of data. Each table has a **schema** which describes what **fields** (i.e. columns) the data should contain.

To give a concrete example, think of a user management system. The database would have a table to hold users. The table would have a schema describing what fields a user should have – for example, a name, an email address and a password. Each user would be represented by a row in the database – for example, Alice would be the first row, and Bob would be the second row.

Setting up Postgres

Setting up the Postgres database on Cloud9 is a little complicated, Follow these instructions carefully, and you'll be good to go. And, of course, if you have any problems, ask your mentor for help.

Cloud9 comes with Postgres installed, so all we need to do is configure it and start the Postgres server. To do so, we need to learn about another command, `sudo`.

`sudo` is an abbreviation for "*s* et *u* ser and *do*". This allows you to run a command as if you were a different user. By default this is as the *superuser*, or system admin. Normally, the `sudo` command is restricted to trusted admin users, but this workspace is all yours - you can do anything with it (including break it, so take special care when running a command using as a superuser).

Step One: Start the Postgres server

First you need to start the Postgres server. Remember this command - any time you get connection errors talking about port 5432, run this again:

```
$ sudo service postgresql start
 * Starting PostgreSQL 9.3 database server
   ...done.
```

In this command, we're using a system administration command *service* to configure a service, which is named `postgresql`. In this case, we give it the `start` command, which starts the Postgres service. In the future, you may want to stop the Postgres service, so you would use the command `stop` in place of `start`. If you want to stop the service and then immediately start it again then the command is `restart`.

Step Two: Create your own user

Postgres, by default, doesn't allow the superuser to run the command-line tool (so the superuser isn't so super after all). Instead, it creates a special user named `postgres`.

It would be fairly tedious if you had to switch to the `postgres` user every time you wanted to work with your database. So the next step is to create a

user which matches your default user.

To do this you can use the `createuser` command. Enter the command, and select the following options:

```
$ sudo sudo -u postgres createuser ubuntu --interactive
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
```

"Wait!" you say, "isn't the `sudo` repeated?" And yes, it is. Here's what is happening. The `sudo` command, by default, runs the command to its right as the *superuser*, or system admin:

```
$ sudo run the command here
```

Adding the `-u` flag to `sudo` says "set user **to the named user** and *do*'. So the following command says become the superuser, then become the postgres user, and finally run the command:

```
$ sudo sudo -u postgres run the command here
```

Here you switch to work as the postgres user (`sudo sudo -u postgres`), and run the `createuser` command. The first argument is the username which you want to use. In this case we use `ubuntu` , the default username on Cloud9. Finally you pass the `--interactive` switch so Postgres asks you for details about which capabilities the user should have.

Each user needs a database associated with it, so create that now:

```
$ createdb ubuntu
```

Try it!

If all has gone well, you can simply type the `psql` command and get directly into the postgres command prompt.

```
$ psql
psql (9.3.7)
Type "help" for help.

ubuntu=#
```

Notice that you don't need to run `sudo sudo -u postgres` any more - your user now has access to the database.

Run the `\password` command to give your user a password, then exit `psql`:

```
ubuntu=> \password
Enter new password: (type in "thinkful" as the password)
Enter it again: (type in "thinkful" again)
ubuntu=> \q
```

The commands starting with a backslash are specific to Postgres. `\password` is used to change the user's password, and `\q` is used to exit the command prompt.

The last stage of setup is to create the new database for the app. To create a database called snippets:

```
$ createdb snippets
```

Creating the table

Now that you have the database set up you'll start putting it to good use. A Postgres database is controlled using a commands written in a language

called SQL. To get started create a file in your project directory called `schema.sql` - we'll build up the table definitions here.

The first table you'll need will contain the actual snippets. Each one will be identified by a case sensitive keyword, which will form a natural key, and will carry a text message. Add the following to your `schema.sql` file:

```
create table snippets (
keyword text primary key,
message text not null default ''
);
```

There's quite a lot to take in here, so let's break it down line-by-line:

```
create table snippets (
```

Here you are issuing a `create table` command. You are telling the database that the table should be called *snippets*.

```
keyword text primary key,
```

This defines the first field in your table. The first part is the name of the field – in this case `keyword`. Next you tell the database what type of data the field should contain – in this case `text`. Postgres has data types which are similar to Python's. The big difference is that we have to declare what type of data a field will hold in advance. `text` holds a varying length sequence of characters, similar to a Python string. To see a description of all of the data types that you can use for a Postgres field take a look at this table.

The final part says that the field should be the primary key for the table. This is known as a **constraint** because it constrains the values which the field can hold. The primary key constraint ensures that any values in the field are not

`NULL`, and are unique (i.e. there are no other rows which have the same primary key). You can always identify an individual row by its primary key.

```
message text not null default ''
```

Here you are creating the final field: a `text` field called `message`. The field has two constraints placed upon its values. The first, `not null`, says that the field cannot hold a `NULL` value. The second, `default ''` says that if no value is provided for the field it should default to an empty string. Note that Postgres differentiates between single and double quotes, so it is best to stick to single quoting strings.

Now that you know what's happening, try running the *schema.sql* script to create the table. In the terminal run:

```
$ psql -d snippets < schema.sql
```

This tells Postgres to run the *schema.sql* file on the `snippets` database.

To check that it worked, open up the interactive Postgres interpreter by saying `psql -d snippets`. Then enter the following:

```
snippets=> \d+ snippets
                              Table "public.snippets"
 Column  | Type |         Modifiers         | Storage  | Stats target
---------+------+---------------------------+----------+--------------
 keyword | text | not null                  | extended |
 message | text | not null default ''::text | extended |
Indexes:
    "snippets_pkey" PRIMARY KEY, btree (keyword)
Has OIDs: no
```

The `\d+` command is used to get information about a table. Here you can see that you successfully create the snippets table with two columns. You can

then enter `\q` to exit the interpreter.

Creating rows

Of course, a table isn't much use on its own, so you'll add some rows to it. To do this, use the `insert into` command. Open up the `psql` interpreter again, and enter the following:

```
insert into snippets values ('insert', 'Add new rows to a table');
```

Here you've told Postgres to insert a new row into the `snippets` table. You've supplied two values, which will be entered in order into the columns. So the string `'insert'` will be entered into the keyword, the first column, and the string `'Add new rows to a table'` will be added to message, the second column.

You can also manually describe which value will be inserted into each column. For example the following SQL statement is equivalent to the one above:

```
insert into snippets (message, keyword) values ('Add new rows to a tab
```

Try it!

- Use the `insert into` command to add two more snippets to your table.
- Try adding a snippet with the same keyword as one of your other snippets. What do you see?

Querying for rows

The database would not be very useful if we could add rows, and not retrieve them. To query the database, use the `select` statement. Try running the following examples:

```
select * from snippets;
select message from snippets;
select keyword, message from snippets where keyword='insert';
```

In the first example we ask Postgres to query for all of the fields (`*`) in the `snippets` table. You should see the insert row plus the extra rows which you added.

The second example is more explicit. Rather than asking for all columns it only asks for the messages column. In general it is better practice to explicitly ask for columns rather than use the `*` wildcard so you are only fetching the information you need.

The final example narrows down the query using a `where` clause. So here you're requesting the specific row where the keyword is `insert` .

Try it!

Write queries to find:

- All of the keywords
- The entire row for one of your own snippets

Updating rows

To change the values held in a row we can use the `update` command. For example:

```
update snippets set message='Insert new rows into a table' where keywo
```

Here you're asking Postgres to update the `snippets` table, setting a new value in the message field. To ensure that you only target one row, you'll use a `where` clause to find the specific row you want to update.

Try it!

- Update the message of another snippet.
- Update the keyword of one of your snippets. Changing the value of the primary key is unusual, but perfectly legal!

Deleting rows

The final part of this whistle-stop tour of SQL is deleting rows using the `delete from` command. For example:

```
delete from snippets where keyword='insert';
```

You tell Postgres to delete from the `snippets` table, selecting a single row using the `where` clause.

Try it!

Create a new snippet, then delete it from your table. Make a query to ensure you deleted the snippet successfully.

Congratulations! You have a fully working table set up to hold the snippets. Next up: Accessing the table from Python!

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous

Next