Tutorial 3 – PostgreSQL

The purpose of this tutorial is to provide an overview of using postgreSQL a relational SQL database hosted via Heroku.

1. Deploy new_app1 as a heroku project

On the command line, “cd” into your Java web application that you've created for your group project in tutorial #2. This should be called “new_app1”.

<< note this tutorial can be done individually from sample_maven_web_app from tutorial #1 >>

Since this is a new source project, go ahead and deploy/build this project to heroku:

```
heroku login
heroku create
git push heroku master
```

2. Add the heroku postgresql plugin to your heroku project

Now add the postgresl add on to your heroku project:

```
heroku addons:create heroku-postgresql:hobby-dev
```

Check the DATABASE_URL and HEROKU_POSTGRESQL_COLOR_URL settings

```
heroku config
```

Check out the pginfo

```
heroku pg:info
```

This shows information about the database we've created on heroku.
The free databases are limited to:
A maximum of 20 user connections
A maximum of 10,000 rows of data

Check out the automatically created database credentials:

```
heroku pg:credentials
```

Your credentials should appear as below:

```
$ heroku pg:credentials
Connection info string:
   "dbname=d19m0j1erhvr7v host=ec2-54-235-204-221.compute-1.amazonaws.com
port=5432 user=wxojhmodfpbmsv
password=80cfef5defecd78ff44e5e2bab48a26b06f930d1f57e097a6be957be98358c53
sslmode=require"
Connection URL:

postgres://wxojhmodfpbmsv:80cfef5defecd78ff44e5e2bab48a26b06f930d1f57e097a6be9
57be98358c53@ec2-54-235-204-221.compute-1.amazonaws.com:5432/d19m0j1erhvr7v
```

Note the value of “dbname”.
Either write this down, or save it in your clipboard, or in a text file for easy retrieval.

3. Next, let's explore the pgsql command line

PostgreSQL one of the leading open source databases has a full feature command-line interface which enables interaction with a database. The advantage with using the command line is that it is possible to fully script everything you would ever need to do with the database by writing “.sql” files.

First let's try out using the psql command line to explore our new heroku hosted postgreSQL database.

Heroku is once again great. Heroku let's you access psql without even installing it locally! Normally if you were to connect to a postgresql database that is not hosted on your computer, you would have to install the command-line client.

To invoke psql via heroku:

```
heroku pg:psql
```

This should connect you to your postgresql relational database hosted by heroku in the cloud!

```
--> Connecting to postgresql-globular-28873
psql (9.1.24, server 9.6.1)
WARNING: psql version 9.1, server version 9.6.
        Some psql features might not work.
SSL connection (cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256)
Type "help" for help.

obscure-ravine-82904::DATABASE=>
```

Now let's list the databases on this server. Type “back-slash” followed by lower-case L.

```
\l
```

Oh my. That's a lot of databases!
Which one is ours?

In order to provide postgresql-as-a-service in the cloud, they've backed a large number of free databases onto a single database server.

Now let's try to query this information from the system table:

```
select * from pg_database;
```

Here's all that data again!
How many rows are there? _____

Let's refine the query to only see our database:

```
select * from pg_database where datname='{put the name of your database here}';
```

Now you should only see one row, for our database.
This provides metadata that describes attributes about our specific database.

4. Creating tables and schemas

Now, let's look at what tables exist in our database.

```
\dt
```

There aren't any.

Let's create one:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT  code_title PRIMARY KEY(code,title)
);
```

Now let's check out the list of tables again:

```
\dt
```

We can check the definition of tables as follows:

```
\d+ public.films
```

That looks familiar.
Wait, what is “public”?
Public is your default schema name.
What is a schema?
Any guesses? _____

Schemas are like collections of tables and objects in a database.
So this means a single database can have multiple schemas, and schemas can have multiple tables.

It is important to note that in postgresql tables belong to schemas
And schemas belong to databases.

The * means many.

One cool feature with a schema is that you can copy/backup/move around sets of tables by working with schemas. You can import an entire schema from one database to another. This was you can move database between databases in partitions.

Let's create a new schema.

```
create schema test;
```

Now, let's look at your schemas:

```
\dn
```

It's “\dn” because sequences, a counter variable type, took precedence over schemas.

Now, postgresql hides certain the system schema names. We can display them with a *.

```
\dn *
```

Now, how do we create a table in a sequence?
There's two ways.
Let's create two tables, one with each method:

Method #1 – set the search_path:

We could switch our command line environment to operate in the sequence:

```
set search_path to test;
```

and then create a table:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40)
);
```

Method #2 – fully qualify the schema/table name within the database:

Let's start by setting the search path to public:

```
set search_path to public;
```

```
CREATE TABLE test.reviews (
    rid     integer,
    name    varchar(40),
    review  text
);
```

Now, let's check out the list of tables in public:

```
\dt
```

Wait, where is the reviews table?
We created it with the search_path set to schema?

You can check the tables in any schema by qualifying your search.
Try this out:

```
\dt test.*
```

```
\dt public.*
```

Next, let's try deleting the tables in test:

```
drop table test.distributors;
```

```
\dn test.*
```

Now this is work, let's just delete the schema test instead of deleting the reviews table:

```
drop schema test;
```

Uh oh, it doesn't work. SQL likes to be VERBOSE!

```
drop table test.reviews;
drop schema test;
```

Now list our schemas:

```
\dn *
```

5. <<C>> RUD

Next, let's add some data to our films table:

Be aware postgresql doesn't like double quotes:

```
insert into films values ('ABC','A Good Movie',3,'01/01/2005','drama','2:20');
```

Now let's inspect our data:

```
select * from films;
```

Let's add a few more rows

```
insert into films values ('ABD','A Long Movie',3,'07/01/2015','action','4:20');
insert into films values ('ABE','A Happy Movie',3,'03/21/2012','comedy','1:50');
insert into films values ('ABF','A Sad Movie',3,'08/14/2011','drama','2:30');
insert into films values ('ABG','I Need Better Titles',3,'09/01/2016','comedy','1:32');
insert into films values ('ABH','A New Movie',3,'01/01/2017','drama','2:00');
```

6. C<<R>>UD

Now let's try to filter data:

```
select * from films where kind='drama';
```

How about a long movie?

```
select * from films where len>'2:00';
```

A newer movie?

```
select * from films where date_prod>'01/01/2012';
```

We can do a similarity match with “%”...

```
select * from films where title like '%Movie%';
```

A compound search:

```
select * from films where title like '%Movie%' and kind='comedy';
```

Let's try a function:

```
select length(title) from films;
```

And now let's alias the column name to be “title_length”:

```
select length(title) title_length from films;
```

To count the number of rows in a table:

```
select count(*) from films;
```

And a sort:

```
select * from films where title like '%Movie%' order by len;
```

Chop a string:

```
select substring(title from 1 for length(title)-5) from films where title like '%Movie%';
```

Let's query a query:

```
select * from (select * from films where title like '%Movie%') a where a.len < '2:20';
```

7. CR<<U>>D

Now let's try to modify some data:

Add the word Really to this movie:

```
update films
set title='A Really Happy Movie'
where title like '%Movie%' and kind='comedy';
```

Modify a bunch of titles at once:

```
update films
set title=substring(title from 1 for length(title)-5) || 'Incredible Movie'
where title like '%Movie%';
```

Check out the data:

select * from films;

Now try a transaction with rollback:

```
begin;
```

```
update films set title=substring(title from 1 for length(title)-5) || 'Inedible Movie' where title like '%Movie%';
```

```
select * from films;
```

```
rollback;
```

```
select * from films;
```

8. CRU<<D>>

Now using some filters from before, let's delete some rows:

```
begin;
```

```
delete from films where title like '%Movie%';
```

```
select * from films;
```

Oops, let's rollback:

```
rollback;
```

```
select * from films;
```

Now this time, let's delete for real:

```
delete from films;
```

Now using a text editor, let's create an SQL script file to load in all of the rows

Create a file called “films.sql” in the same directory as your new_app1 project with the following lines:

```
insert into films values ('ABC','A Good Movie',3,'01/01/2005','drama','2:20');
insert into films values ('ABD','A Long Movie',3,'07/01/2015','action','4:20');
insert into films values ('ABE','A Happy Movie',3,'03/21/2012','comedy','1:50');
insert into films values ('ABF','A Sad Movie',3,'08/14/2011','drama','2:30');
insert into films values ('ABG','I Need Better Titles',3,'09/01/2016','comedy','1:32');
insert into films values ('ABH','A New Movie',3,'01/01/2017','drama','2:00');
```

Now from within psql, check that films is empty:

```
select * from films;
```

Now load in the script file:

```
\i films.sql
```

Now check out the table again:

```
select * from films;
```

Further documentation on postgresql can be found here:
https://www.postgresql.org/docs/9.6/static/index.html

Further documentation on the heroku pg command line can be found here:
https://devcenter.heroku.com/articles/heroku-postgresql#using-the-cl

9. PGADMIN: a graphical IDE for working with PostgreSQL

Now the command-line is great, but sometimes you just would like a GUI to explore the database more graphically.

Let's install PostgreSQL's free pgAdmin database IDE.

For Windows/MAC, visit the following URL and follow the instructions:
https://www.pgadmin.org/

For Ubuntu, install as follows:

```
sudo apt-get update
```

```
sudo apt-get install pgadmin3
```

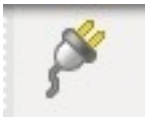
Once installed, let's revisit what our DB connection string was again:

```
heroku pg:credentials
```

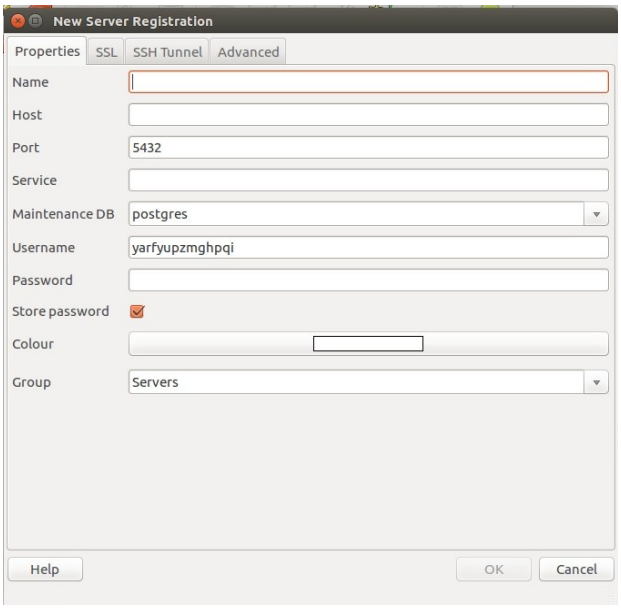
Note the values for dbname, host, port, user, password, and sslmode.
You may want to email these to yourself so you can pull the values into the ubuntu VM, unless you've installed the VBOX Guest Additions so copy and paste works between your Host OS and the Virtual Box VM.

Now from within PGADMIN let's create a connection to this Heroku DB (which we now see is actually hosted in the Amazon Cloud!)

Click on the plug icon:

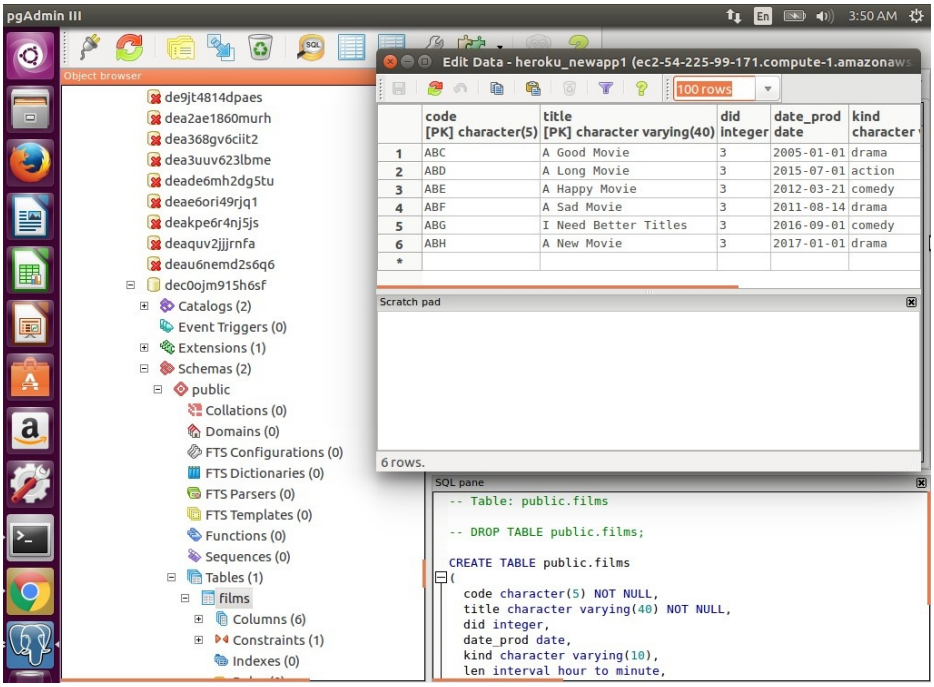


Now fill in the fields of the dialog from what you've noted before:



You may receive a warning when connecting. This is because Heroku is using PostgreSQL 9.6, and pgAdmin3 only guarantees supports up through 9.5... There is a new version of pgAdmin4, which you can install on your own. PgAdmin4 is deployed as a server-based application so users can interface remotely to the GUI, where pgadmin3 is a desktop application.

Now once you connect, scroll through the really long list of database names. There should be one that does not have a red “X”. Double click, and explore, this is your database. You'll see that you can do everything we did before, but now within the GUI.



10. Receiving Credit for this Tutorial

To complete this tutorial and receive a grade, a group assignment has been set up in Canvas. Submit to canvas, your connection information. Provide complete values for dbname, host, port, user, password, and sslmode.