Então eu resolvi escrever esta série de 3 posts sobre PostgreSQL. Onde os outros 2 são:

PostgreSql e Python3 - parte 2

PostgreSql e Django - parte 3

Sumário:

Ah, talvez isso aqui também seja útil postgresql cheat sheet.

Instalação

Eu apanhei bastante para instalar até descobrir que meu SO estava zuado... então se você tiver problemas na instalação não me pergunte porque eu não saberei responder. Eu recorri à comunidade... e Google sempre! Dai eu

formatei minha máquina instalei um Linux do zero e deu certo... (não sei instalar no Windows)...

Bom, vamos lá. No seu **terminal** digite esta sequência:

$ dpkg -l | grep -i postgres

Este comando é só pra ver se não tem alguma coisa já instalado.

Agora instale...

$ sudo apt-get install -y python3-dev python3-setuptools postgresql-9.3 postgresql-contrib-9.3 pgadmin3 libpq-dev build-essential binutils g++

pgadmin3 é a versão com interface visual... no youtube tem vários tutoriais legais. É bem fácil de mexer.

binutils g++ talvez não seja necessário, mas no meu caso precisou.

No final, se você conseguir ver a versão do programa é porque deu tudo certo.

$ psql -V
psql (PostgreSQL) 9.3.5

Criando um banco de dados

Usando o **terminal** sempre!

$ sudo su - postgres

Seu prompt vai ficar assim:

postgres@usuario:~$

Criando o banco

$ createdb mydb

Se quiser deletar o banco

$ dropdb mydb

Criando um usuário

$ createuser -P myuser

Acessando o banco

$ psql mydb

O comando a seguir define direito de acesso ao novo usuário.

```
$ GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
```

Para sair do programa **psql**

```
\q
```

Para sair do *root* pressione ctrl+d.

Usando o banco de dados

Antes vamos criar 2 arquivos porque nós iremos usá-los mais na frente.

person.csv

```
$ cat > person.csv << EOF
name,age,city_id
Abel,12,1
Jose,54,2
Thiago,15,3
Veronica,28,1
EOF
```

basics.sql

```
$ cat > basics.sql << EOF
CREATE TABLE cities (id SERIAL PRIMARY KEY, city VARCHAR(50), uf
VARCHAR(2));
INSERT INTO cities (city, uf) VALUES ('São Paulo', 'SP');
SELECT * FROM cities;
DROP TABLE cities;
EOF
```

Agora, vamos abrir o banco de dados *mydb*.

```
$ psql mydb
psql (9.3.5)
Type "help" for help.

mydb=>
```

Para rodar os comandos que estão no arquivo *basics.sql* digite

```
mydb=> \i basics.sql
```

Resultado:

```
CREATE TABLE
INSERT 0 1
```

```
 id |   city    | uf
----+-----------+----
  1 | São Paulo | SP
(1 row)
```

DROP TABLE

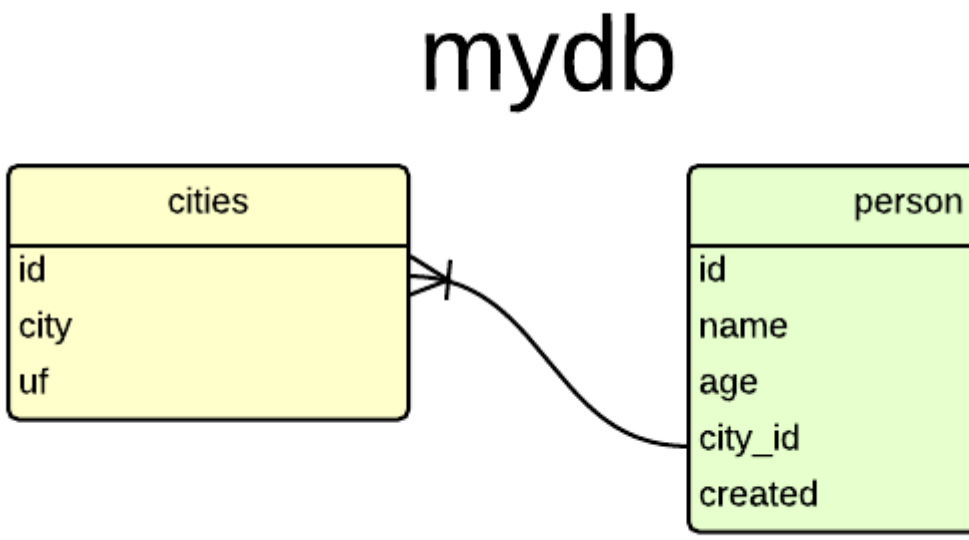
Como você deve ter percebido, criamos uma tabela *cities*, inserimos um registro, lemos o registro e excluimos a tabela.

Criando as tabelas

Daqui pra frente vou omitir o prompt, assumindo que seja este:

mydb=>

Considere as tabelas a seguir:



Então vamos criar as tabelas...

Ah, talvez isso aqui também seja útil postgresql cheat sheet.

```
CREATE TABLE cities (id SERIAL PRIMARY KEY, city VARCHAR(50), uf
VARCHAR(2));
CREATE TABLE person (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    city_id INT REFERENCES cities(id),
    created TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW()
);
```

Alguns comandos:

SERIAL é o conhecido *auto incremento* numérico.

TIMESTAMP WITH TIME ZONE data e hora com *time zone*.

DEFAULT NOW() insere a data e hora atual automaticamente.

Mais tipos de campos em Chapter 8. Data Types.

Para ver as tabelas

\dt

Resultado:

```
        List of relations
 Schema |  Name  | Type  | Owner
--------+--------+-------+--------
 public | cities | table | myuser
 public | person | table | myuser
(2 rows)
```

Para ver o esquema de cada tabela

\d cities

Resultado:

```
                    Table "public.cities"
 Column |         Type          |                     Modifiers
--------+-----------------------+---------------------------------------------------
 id     | integer               | not null default nextval('cities_id_seq'::regclass)
 city   | character varying(50) |
 uf     | character varying(2)  |
Indexes:
    "cities_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "person" CONSTRAINT "person_city_id_fkey" FOREIGN KEY
(city_id) REFERENCES cities(id)
```

Para deletar as tabelas

```
DROP TABLE cities
DROP TABLE person
```

Para definir o *timezone*

```
SET timezone = 'America/Sao_Paulo';
```

Caso dê erro ao inserir a data tente

SET timezone = 'UTC';

Dica: stackoverflow

Inserindo dados

Pra quem já manja de SQL...

INSERT INTO cities (city, uf) VALUES ('São Paulo', 'SP'),('Salvador', 'BA'),('Curitiba', 'PR');
INSERT INTO person (name, age, city_id) VALUES ('Regis', 35, 1);

Se lembra do arquivo *person.csv* que criamos lá em cima?

Troque *user* pelo nome do seu usuário!

COPY person (name,age,city_id) FROM '/home/user/person.csv' DELIMITER ',' CSV HEADER;

Erro: Comigo deu o seguinte erro:

ERROR: must be superuser to COPY to or from a file

Ou seja, você deve entrar como *root*. Saia do programa e entre novamente.

$ sudo su - postgres
$ psql mydb
mydb=# COPY person (name,age,city_id) FROM '/home/user/person.csv' DELIMITER ',' CSV HEADER;

Repare que o prompt ficou com #, ou seja, você entrou como *root*.

Lendo os dados

Pra quem não sabe usar JOIN...

SELECT * FROM person ORDER BY name;
SELECT * FROM person INNER JOIN cities ON (person.city_id = cities.id) ORDER BY name;

Resultado:

```
 id |   name   | age | city_id |           created            | id |   city    | uf
----+----------+-----+---------+------------------------------+----+-----------+----
  2 | Abel     |  12 |       1 | 2015-02-04 03:49:01.597185-02 |  1 | São Paulo | SP
  3 | Jose     |  54 |       2 | 2015-02-04 03:49:01.597185-02 |  2 | Salvador  | BA
  1 | Regis    |  35 |       1 | 2015-02-04 03:47:10.63258-02  |  1 | São Paulo | SP
  4 | Thiago   |  15 |       3 | 2015-02-04 03:49:01.597185-02 |  3 | Curitiba  | PR
```

```
  5 | Veronica |  28 |       1 | 2015-02-04 03:49:01.597185-02 |  1 | São Paulo |
SP
(5 rows)
```

Exemplo de count e inner join

Um exemplo interessante, e talvez útil, é saber quantas pessoas moram em cada cidade.

```
SELECT cities.city, COUNT(person.city_id) AS persons
FROM cities INNER JOIN person ON cities.id = person.city_id
GROUP BY cities.city;
```

Mais em 2.7. Aggregate Functions.

```
   city    | persons
-----------+---------
 São Paulo |       3
 Curitiba  |       1
 Salvador  |       1
(3 rows)
```

E apenas para não esquecer, o operador para *diferente* é

```
SELECT * FROM person WHERE city_id <> 1;
```

Atualizando

```
UPDATE person SET name = 'Jose da Silva', age = age - 2 WHERE name =
'Jose';
```

antes: SELECT * FROM person WHERE name Like 'Jose';

```
 id | name | age | city_id |            created
----+------+-----+---------+-------------------------------
  3 | Jose |  54 |       2 | 2015-02-04 03:49:01.597185-02
```

depois: SELECT * FROM person WHERE id=3;

```
 id |     name      | age | city_id |            created
----+---------------+-----+---------+-------------------------------
  3 | Jose da Silva |  52 |       2 | 2015-02-04 03:49:01.597185-02
```

Note que age = age - 2 fez com que a idade diminuisse de 54 para 52. Ou seja, dá pra fazer operações algébricas com UPDATE.

Deletando

```
DELETE FROM person WHERE age < 18;
```

Fazendo SELECT * FROM person; repare que foram excluidos *Abel* e *Thiago*.

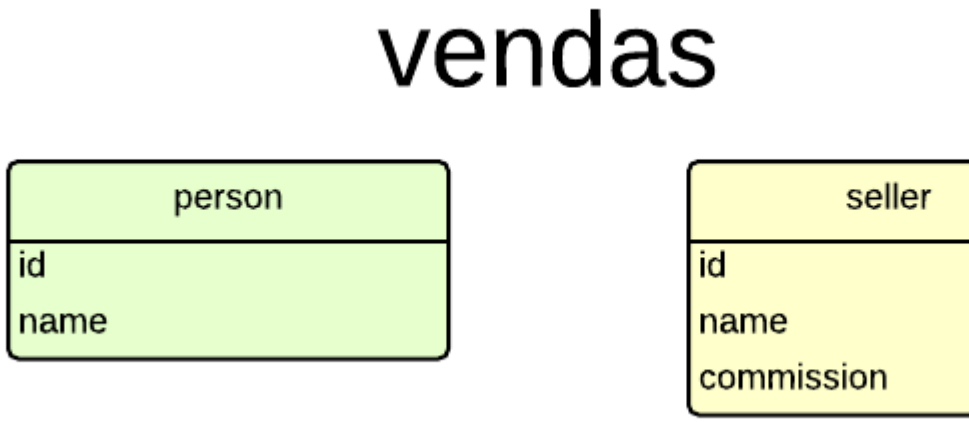
```
 id |     name      | age | city_id |            created
----+---------------+-----+---------+-------------------------------
  1 | Regis         |  35 |       1 | 2015-02-04 03:47:10.63258-02
  5 | Veronica      |  28 |       1 | 2015-02-04 03:49:01.597185-02
  3 | Jose da Silva |  52 |       2 | 2015-02-04 03:49:01.597185-02
```

Mais informações em Chapter 2. The SQL Language.

Herança

Considere o banco de dados chamado *vendas*.

Suponha que você tenha duas tabelas: *person* (pessoa) e *seller* (vendedor).



Então façamos:

```
$ sudo su - postgres
$ createdb vendas
$ psql vendas
CREATE TABLE person (
    id SERIAL PRIMARY KEY,
    name TEXT
);
CREATE TABLE seller (
    id SERIAL PRIMARY KEY,
    name TEXT,
    commission DECIMAL(6,2)
);
INSERT INTO person (name) VALUES ('Paulo');
INSERT INTO seller (name,commission) VALUES ('Roberto',149.99);
```

Dai criamos uma VIEW:

```
CREATE VIEW peoples AS
    SELECT name FROM person
        UNION
    SELECT name FROM seller;

SELECT * FROM peoples;
```

Que retorna:

```
  name
---------
 Paulo
 Roberto
(2 rows)
```

Lembre-se que 'Paulo' pertence a *person* e 'Roberto' pertence a *seller*.

Mas esta não é a melhor solução. Usando a herança façamos da seguinte forma:

```
DROP VIEW peoples;
DROP TABLE person, seller;

CREATE TABLE person (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50)
);
CREATE TABLE seller (
    commission DECIMAL(6,2)
) INHERITS (person);
```

Fazendo

```
\d person
                            Table "public.person"
 Column |         Type          |                     Modifiers
--------+-----------------------+---------------------------------------------------
 id     | integer               | not null default nextval('person_id_seq'::regclass)
 name   | character varying(50) |
Indexes:
    "person_pkey" PRIMARY KEY, btree (id)
Number of child tables: 1 (Use \d+ to list them.)
```

E

```
\d seller
                          Table "public.seller"
   Column   |         Type          |                     Modifiers
```

```
------------+----------------------+-----------------------------------------------------
 id         | integer              | not null default nextval('person_id_seq'::regclass)
 name       | character varying(50) |
 commission | numeric(6,2)         |
Inherits: person
```

A diferença é que com menos código criamos as duas tabelas e não precisamos criar VIEW. Mas a tabela *seller* depende da tabela *person*.

Portanto não conseguimos deletar a tabela *person* sozinha, precisaríamos deletar as duas tabelas de uma vez.

Vantagem:

- a associação é do tipo *one-to-one*
- o esquema é extensível
- evita duplicação de tabelas com campos semelhantes
- a relação de dependência é do tipo pai e filho
- podemos consultar o modelo pai e o modelo filho

Desvantagem:

- adiciona sobrecarga substancial, uma vez que cada consulta em uma tabela filho requer um *join* com todas as tabelas pai.

Vamos inserir alguns dados.

```
INSERT INTO person (name) VALUES ('Paulo'),('Fernando');
INSERT INTO seller (name,commission) VALUES
   ('Roberto',149.99),
   ('Rubens',85.01);
```

Fazendo

```
SELECT name FROM person;
```

```
      name
----------
 Paulo
 Fernando
 Roberto
 Rubens
(4 rows)
```

Obtemos todos os nomes porque na verdade um *seller* também é um *person*.

Agora vejamos somente os registros de *person*.

```
SELECT name FROM ONLY person;
```

```
   name
----------
 Paulo
 Fernando
(2 rows)
```

E somente os registros de *seller*.

```
SELECT name FROM seller;

  name
---------
 Roberto
 Rubens
(2 rows)
```

Mais informações em 3.6. Inheritance.

Modificando tabelas

Vejamos agora como inserir um novo campo numa tabela existente e como alterar as propriedades de um outro campo.

Para inserir um novo campo façamos

```
ALTER TABLE person ADD COLUMN email VARCHAR(30);
```

Para alterar as propriedades de um campo existente façamos

```
ALTER TABLE person ALTER COLUMN name TYPE VARCHAR(80);
```

Antes era name VARCHAR(50), agora é name VARCHAR(80).

Também podemos inserir um campo com um valor padrão já definido.

```
ALTER TABLE seller ADD COLUMN active BOOLEAN DEFAULT TRUE;
\d seller
                      Table "public.seller"
   Column   |         Type          |                    Modifiers
------------+-----------------------+-------------------------------------------------
 id         | integer               | not null default nextval('person_id_seq'::regclass)
 name       | character varying(80) |
 commission | numeric(6,2)          |
 active     | boolean               | default true
Inherits: person
```

Façamos SELECT novamente.

```
SELECT * FROM seller;
 id |  name   | commission | active
```

```
----+---------+------------+--------
  3 | Roberto |     149.99 | t
  4 | Rubens  |      85.01 | t
(2 rows)
```

Vamos definir um email para cada pessoa. O comando lower torna tudo **minúsculo** e || **concatena** textos.

```
UPDATE person SET email = lower(name) || '@example.com';
SELECT * FROM person;

 id |   name   |        email
----+----------+---------------------
  1 | Paulo    | paulo@example.com
  2 | Fernando | fernando@example.com
  3 | Roberto  | roberto@example.com
  4 | Rubens   | rubens@example.com
(4 rows)
```

Leia 9.4. String Functions and Operators e ALTER TABLE.

Backup

```
pg_dump mydb > bkp.dump
# ou
pg_dump -f bkp.dump mydb
```

Excluindo o banco

```
dropdb mydb
```

Criando novamente e **recuperando os dados**

```
createdb mydb; psql mydb < bkp.dump
```

Leia 24.1. SQL Dump.

Parte 2

Além da instalação mostrada no primeiro post precisaremos de

```
$ sudo apt-get install python-psycopg2 # para python2
# ou
$ sudo apt-get install python3-psycopg2 # para python3
```

Começando...

```
$ sudo su - postgres
```

Veja o prompt:

postgres@myuser:~$

Criando o banco

$ createdb mydb

Se existir o banco faça

$ dropdb mydb

e crie novamente. Para sair digite

$ exit

Abra o python3.

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Importe o psycopg2

```
>>> import psycopg2
```

Conectando a um banco de dados existente

```
>>> conn = psycopg2.connect("dbname=mydb user=myuser")
```

Abrindo um cursor para manipular o banco

```
>>> cur = conn.cursor()
```

Criando uma nova tabela

```
>>> cur.execute("CREATE TABLE person (id serial PRIMARY KEY, name text,
age integer);")
```

Inserindo dados.O Psycopg faz a conversão correta. Não mais injeção SQL.

```
>>> cur.execute("INSERT INTO person (name, age) VALUES (%s,
%s)",("O'Relly", 60))
>>> cur.execute("INSERT INTO person (name, age) VALUES (%s,
%s)",('Regis', 35))
```

Grava as alterações no banco

```
>>> conn.commit()

# Select

>>> cur.execute("SELECT * FROM person;")
>>> cur.fetchall()
```

Fecha a comunicação com o banco

```
>>> cur.close()
>>> conn.close()
>>> exit()
```

Precisamos criar o banco manualmente

Começando...

```
$ sudo su - postgres
```

Veja o prompt:

```
postgres@myuser:~$
```

Criando o banco

```
$ createdb mydb
```

Se existir o banco faça

```
$ dropdb mydb
```

e crie novamente. Para sair digite

```
$ exit
```

Django

Vamos criar um virtualenv e instalar o psycopg2, além do django.

```
virtualenv -p /usr/bin/python3 teste
cd teste
source bin/activate
pip install psycopg2 django
pip freeze
pip freeze > requirements.txt
```

Dica: Para diminuir o caminho do prompt digite

```
$ PS1="(`basename \"$VIRTUAL_ENV\"`):/\W$ "
```

Dica:

```
vim ~/.bashrc +
alias manage='python $VIRTUAL_ENV/manage.py'
```

Com isto nós podemos usar apenas manage ao invés de python manage.py.

Criando o projeto

```
django-admin.py startproject myproject .
cd myproject
python ../manage.py startapp core
```

Edite o settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydb',
        # 'NAME': os.path.join(BASE_DIR, 'mydb'),
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '', # 8000 is default
    }
}
```

Rode a aplicação

```
python manage.py migrate
python manage.py runserver
```

http://127.0.0.1:8000/ ou http://localhost:8000/

Edite o models.py

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Person(models.Model):
    name = models.CharField(_('Nome'), max_length=50)
    email = models.EmailField(_('e-mail'), max_length=30, unique=True)
    age = models.IntegerField(_('Idade'))
    active = models.BooleanField(_('Ativo'), default=True)
    created_at = models.DateTimeField(
        _('Criado em'), auto_now_add=True, auto_now=False)
```

```
    class Meta:
        ordering = ['name']
        verbose_name = "pessoa"
        verbose_name_plural = "pessoas"

    def __str__(self):
        return self.name
```

Leia mais em

Tutorial Django 1.7

Como criar um site com formulário e lista em 30 minutos?

Edite o settings.py novamente

Em *INSTALLED_APPS* insira a app *core*.

```
INSTALLED_APPS = (
    ...
    'myproject.core',
)
```

Faça um migrate

```
python manage.py makemigrations core
python manage.py migrate
python manage.py createsuperuser
```

Um pouco de shell

```
$ python manage.py shell
Python 3.4.0 (default, Apr 11 2014, 13:05:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Serve para manipular a app pelo **terminal**.

```
>>> from myproject.core.models import Person
>>> p =
Person.objects.create(name='Regis',email='regis@example.com',age=35)
>>> p.id
>>> p.name
>>> p.email
>>> p.age
>>> p.active
>>> p.created_at
```

```
>>> p =
Person.objects.create(name='Xavier',email='xavier@example.com',age=66,acti
ve=False)
>>> persons = Person.objects.all().values()
>>> for person in persons: print(person)
>>> exit()
```