Acceso a Bases de datos (Práctico)

tiiMiit by

Tema 1. Práctica de PostgreSQL

Instalación PostgreSQL

La instalación de PostgreSQL quedará sujeta al tipo y versión de sistema operativo que se disponga. Se aconseja buscar un tutorial específico para cada caso y consultar todo lo que sea necesario..

En el siguiente documento se tomará como referencia la instalación en MacOS.

Para instalar PostgreSQL ejecutaremos el siguiente comando

brew install postgresql

Luego de la instalación comenzaremos a ejecutar como servicio del SO a PostgreSQL con el siguiente comando

brew services start postgresql

Veremos algo similar a lo siguiente

```
==> Tapping homebrew/services
Cloning into '/opt/homebrew/Library/Taps/homebrew/homebrew-services'...
remote: Enumerating objects: 2139, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 2139 (delta 11), reused 24 (delta 8), pack-reused 2106
Receiving objects: 100% (2139/2139), 598.29 KiB | 966.00 KiB/s, done.
Resolving deltas: 100% (952/952), done.
Tapped 1 command (45 files, 754.3KB).
Warning: Use postgresql@14 instead of deprecated postgresql
==> Successfully started `postgresql@14` (label: homebrew.mxcl.postgresql@14)
```

Para detener el servicio en caso de ser necesario utilizaremos el siguiente comando:

brew services stop postgresql

Y para reiniciarlo el siguiente:

brew services restart postgresql

Primeros comandos en PostgreSQL

Una vez instalado PostgreSQL e inicializado el servicio procederemos a conectarnos a tu interfaz por línea de comando del siguiente modo

psql postgres

Una vez hecho ese veremos lo siguiente en pantalla

```
psql (14.5 (Homebrew))
Type "help" for help.

postgres=#
```

Luego vamos a tipear el comando ****pg-start**** para luego pedir información del usuario que está actualmente conectado a la interfaz con el comando ***\conninfo***

```
postgres=# *pg-start*
postgres-# \conninfo
You are connected to database "postgres" as user "santiago" via socket in "/tmp" at port "5432".
```

Adicionalmente con el comando \du podremos ver toda la información asociada a cada uno de los usuarios

```
postgres-# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 santiago  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Creación de usuario y configuración de password en PostgreSQL

A continuación veremos la forma de crear un usuario dentro de PostgreSQL. Para ello utilizaremos el siguiente comando

CREATE USER user_name;

Por ejemplo, vamos a crear el usuario llamado developer

```
postgres=# CREATE USER developer;
CREATE ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 developer |                                                            | {}
 santiago  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Como se ve en la imagen, al ejecutar el comando \du vemos que aún no tiene ningún rol asignado al usuario developer.

Le asignaremos un permiso de ejemplo para luego continuar con la creación de la base de datos con la cuál trabajaremos en la práctica.

```
postgres=# ALTER ROLE developer CREATEDB;
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 developer | Create DB                                                  | {}
 santiago  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Para más detalle sobre la creación de usuarios y configuración de roles se puede visitar la documentación oficial de postgresql:

https://www.postgresql.org/docs/8.1/sql-createrole.html
https://www.postgresql.org/docs/8.1/sql-createrole.html

Creación de base de datos y tablas en PostgreSQL

Para la creación de una base de datos dentro de PostgreSQL utilizaremos el siguiente comando

CREATE DATABASE database_name;

Como ejemplo, crearemos una base de datos llamada test

```
postgres=# CREATE DATABASE test;
CREATE DATABASE
```

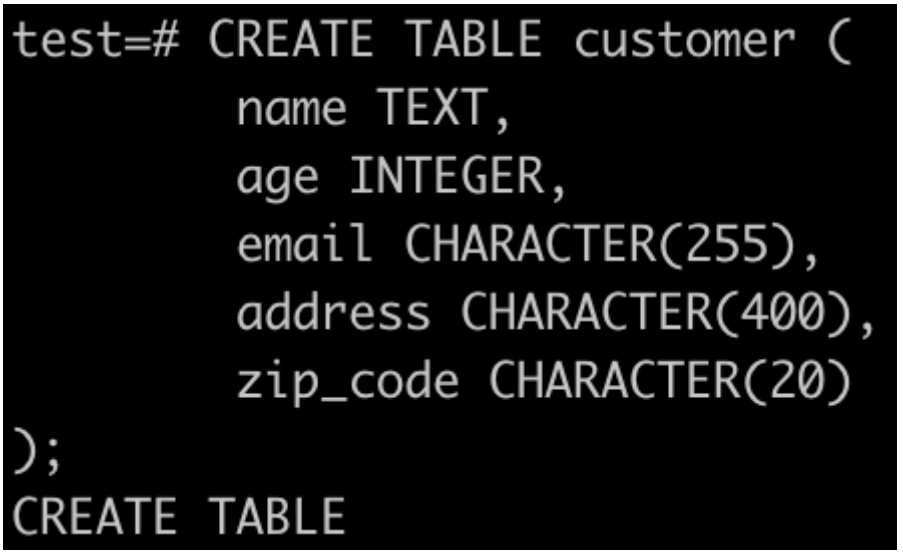
Una vez creada una base de datos podremos crear tablas dentro de la misma con el comando CREATE TABLE pero previo a eso debemos establecer nuestra conexión hacia dicha base de datos con el comando **\c *db_name***, en nuestro caso será \c test

```
postgres=# \c test
You are now connected to database "test" as user "santiago"
test=#
```

Ahora si a continuación crearemos una tabla de ejemplo llamada customer.

Comando para la creación de tabla

```
CREATE TABLE customer (
    name TEXT,
    age INTEGER,
    email CHARACTER(255),
    address CHARACTER(400),
    zip_code CHARACTER(20)
);
```



Con los comandos **\d** , **\d *table_name*** podremos acceder al esquema de la base de datos y al detalle de cada tabla.

```
test=# \d customer
                Table "public.customer"
  Column  |      Type      | Collation | Nullable | Default
----------+----------------+-----------+----------+---------
 name     | text           |           |          |
 age      | integer        |           |          |
 email    | character(255) |           |          |
 address  | character(400) |           |          |
 zip_code | character(20)  |           |          |
```

```
test=# \d
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | customer | table | santiago
(1 row)
```

Prestar atención al nombre del esquema que es public y que dentro está la tabla customer.

Por último insertamos datos en la tabla customer que hemos creado por lo que vamos a ejecutar el siguiente comandos que figuran a continuación

Comando para la inserción de datos en la tabla:

INSERT INTO customer(name,age,email,address,zip_code)
VALUES
('Paul',23,'paul@gmail.com','address from paul','2321LL'),
('Felipe',32,'felipegarcia@gmail.com','address from felipe','3413MS'),
('Teddy',90,'teddy@gmail.com','address from teddy','3423PO'),
('Mark',17,'mark@gmail.com','address from mark','9423MA'),
('David',35,'david@gmail.com','address from david','2341DA'),
('Allen',56,'allen@gmail.com','address from allen','3423PO'),
('James',56,'james@gmail.com','address from james','3423PO');

Finalmente crearemos un usuario llamado ***user_test*** con una password llamada ***test_password*** que le daremos acceso a la base de datos que ya creamos previamente llamada ***test***

Para ello utilizaremos los siguientes comandos

create user user_test with encrypted password 'test_password'; **grant all privileges on database test to user_test;**

```
test=# create user test_develop with encrypted password 'test_password';
CREATE ROLE
test=# grant all privileges on database test to test_develop;
GRANT
```

En la imagen se muestra la ejecución de ambos comandos para tener de referencia.

Con todo lo visto hasta aquí tenemos lo suficiente para realizar la práctica de SQLAlchemy para conectarnos desde Python con esta base de datos que está en PostgreSQL.

Para más información sobre comandos de PostgreSQL https://medium.com/thedevproject/most-important-sql-postgresql-commands-that-you-need-to-know-65f4197233a1

Tema 2. SQLAlchemy conexión a PostgreSQL

Instalación SQLAlchemy en Python

A continuación se listan las dependencias necesarias para instalar SQLAlchemy en python

pip install sqlalchemy
pip install psycopg2

En Mac es probable que no funcione con psycopg2, en dicho caso se puede probar con

pip install psycopg2-binary

Creación de conector a PostgreSQL con SQLAlchemy

En primer lugar, se va a crear un archivo llamado base de ***database.py***, y dentro de ese archivo se va a importar sqlalchemy y crear una clase llamada **Database**.

Los códigos que se muestran a continuación deben probarse utilizando un intérprete de python, o bien, desde una jupyter notebook de forma local.

El código de referencia puede ser el siguiente:

```
import sqlalchemy as db

class Database():
    # replace the user, password, hostname and database according to your configuration
    engine = db.create_engine('postgresql://user:password@hostname/database_name')
    def __init__(self):
        self.connection = self.engine.connect()
        print("DB Instance created")
```

Debe reemplazarse user, password, hostname y database_name por los valores que correspondan.

Lo primero que queremos hacer es conectarnos a nuestra base de datos y para hacer eso debemos tenemos una primera forma que es utilizando ***create_engine()*** y ***connect()*** como se muestra en el ejemplo. Una segunda forma es crear una Session, que por detrás también creará una conexión como en el caso anterior.

Una vez establecida la conexión ya estamos en condiciones de hacer nuestra primer query a la base de datos.

Vamos a crear un método que reciba un argumento que será parte de la query y realice una request a la base de datos con la información que buscamos

Veamos el ejemplo a continuación:

```
def fetchByQyery(self, query):
    fetchQuery = self.connection.execute(f"SELECT * FROM {query}")

    for data in fetchQuery.fetchall():
        print(data)
```

Se puede ver que se ejecuta una query genérica sobre la tabla que le pasamos como parámetro. Por el momento estamos utilizando un SELECT * FROM que devuelve todas las filas de la tabla para conocer el funcionamiento. Se aconseja fuertemente probar la ejecución de los códigos que se van presentando.

Una aclaración respecto de la conexión, es que podríamos, por ejemplo, cerrar nosotros mismos, solo necesitamos hacer esto self.connection.close().

La otra forma de conectarnos con PostgreSQL es crear nuestra conexión usando una sesión, y lo más probable es que cuando se tengan muchos lugares que necesiten conectarse a la base de datos, debería usar esta. Para más detalles puede revisarse la documentación de SQLAlchemy

Conexión:https://docs.sqlalchemy.org/en/13/core/connections.html
Sesión:https://docs.sqlalchemy.org/en/13/orm/session_basics.html

En la siguiente sección profundizaremos en los distintos tipos de consultas que podemos hacer a través de SQLAlchemy sin necesidad de formularlo como una query tradicional de SQL.

Tema 3. SQLAlchemy Consultas a PostgreSQL

Consultas utilizando SQLAlchemy

Ahora bien, visto un ejemplo de base podremos parametrizar nuestras querys ajustadas a demanda de lo que necesitemos utilizando objetos para insertar en una tabla y posteriormente usaremos los conceptos de ORM que hasta ahora no estamos usando.

Vamos a definir una clase customer que respetará la misma estructura de campos que la tabla customer de la base de datos

```
class Customer():
    def __init__(self,name, age, email, address, zip_code):
        self.name = name
        self.age = age
        self.email = email
        self.address = address
        self.zip_code = zip_code
```

Ahora crearemos un método dentro de la clase Database llamado saveData para insertar datos en la tabla utilizando la clase que definimos previamente.

```
def saveData(self, customer):
    self.connection.execute(f"""INSERT INTO customer(name, age, email, address, zip_code)
                                VALUES( '{customer.name}',
                                        '{customer.age}',
                                        '{customer.email}',
                                        '{customer.address}',
                                        '{customer.zip_code}')""")
```

Una vez creado instanciamos un objeto de la clase customer y utilizando el método saveData pasando como parámetro el objeto customer haremos un insert en la tabla customer de la base de datos. Por último para verificar volveremos a invocar al método fetchByQuery y veremos que estará el nuevo dato agregado.

```
customer = Customer("Leonel", 28, "leo@gmail.com", "Av. Rivadavia 2500", "1455")
```

```
db.saveData(customer)
```

```
db.fetchByQyery('public.customer')
```

Ahora bien, visto como podemos insertar en una tabla, pasaremos a ver de qué forma podemos consultar, y para ello nos apalancamos en ORM.

Para usar ORM necesitaremos importar más módulos que importamos de SQLAlchemy de la siguiente forma

from sqlalchemy import MetaData, Table, Column.

A continuación definiremos el siguiente método llamado fetchUserByName que nos permitirá traernos los nombres de la columna name de la tabla customer.

```
def fetchUserByName(self):
    meta = MetaData()
    customer = Table('customer', meta, Column('name'))
    data = self.connection.execute(customer.select())
    for cust in data:
        print(cust)
```

Este método estará dentro de la clase DataBase definida previamente y requiere de los imports que se mencionaron previamente.

Al ejecutar el método veremos el siguiente resultado

```
db_object.fetchUserByName()
```

```
('Paul',)
('Felipe',)
('Teddy',)
('Mark',)
('David',)
('Allen',)
('James',)
('Leonel',)
```

Ahora debemos hacer ciertas modificaciones sobre nuestra clase customer para que se ajuste al uso de las bibliotecas ORM

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, String, Integer

Base = declarative_base()

class Customer(Base):
    __tablename__ = 'customer'
    name = Column(String)
    age = Column(Integer)
    email = Column(String)
    address = Column(String)
    zip_code = Column(String)
    id = Column(Integer, primary_key=True)
    def __repr__(self):
        return "<Customer(name='%s', age='%s', email='%s', address='%s', zip code='%s')>"
                % (self.name, self.age, self.email, self.address, self.zip_code)
```

Aquí necesitaremos importar un nuevo módulo llamado Base, que es la clase base que todos los modelos deben heredar para poder mapear este objeto en la base de datos. Para tener este objeto "Base" necesitamos "crearlo" antes de la declaración de esta clase.

Tenemos un nuevo atributo en nuestra clase llamado ***__tablename__*** que le asignamos el nombre 'customer' para que coincida con el nombre de la tabla en la base de datos.

Para cada columna se necesita declarar el tipo, es por eso que tenemos una para cada atributo.

Finalmente tenemos un método ***def __repr__(self)*** que es solo una forma de tener una buena representación de este objeto cuando se imprime

Ahora con esta nueva definición de la clase volvemos a los métodos de la clase DataBase para definir el método fetchAllUsers como se describe a continuación

```
def fetchAllUsers(self):
    # bind an individual Session to the connection
    self.session = Session(bind=self.connection)
    customers = self.session.query(Customer).all()
    for cust in customers:
        print(cust)
```

Este método utilizará la clase definida customer para hacer una query sobre la base de datos mapeando cada atributo de la clase con cada una de las columnas de la tabla.

Aquí tenemos que usar una sesión, que tiene la particularidad de tener muchas sesiones sobre la misma conexión.

De forma análoga se pueden definir distintas funciones como las que se mencionan a continuación

Create Customer

```
def saveData(self, customer):
    session = Session(bind=self.connection)
    session.add(customer)
        session.commit()
```

Update Customer

```
def updateCustomer(self, customerName, address):
        session = Session(bind=self.connection)
        dataToUpdate = {Customer.address: address}
        customerData = session.query(Customer).filter(Customer.name==customerName)
        customerData.update(dataToUpdate)
        session.commit()
```

Delete Customer

```
def deleteCustomer(self, customer):
    session = Session(bind=self.connection)
    customerData = session.query(Customer).filter(Customer.name==customer).first()
    session.delete(customerData)
    session.commit()
```

Cierre

Durante esta práctica abordamos la instalación de PostgreSQL para utilizar SQLAlchemy. Para ello instalamos PostgreSQL de manera local haciendo las configuraciones iniciales de usuario, creando una base de datos y tabla de ejemplo para luego poder conectarse.

Posteriormente a toda la puesta en marcha de PostgreSQL realizamos una práctica de conexión utilizando SQLAlchemy en Python para interactuar con dicha base de datos haciendo diversas consultas y manipulaciones de datos utilizando las herramientas de ORM.


tiimiit
by