Week 6 - Databases in python(Postgresql)

Content of practical work:

1. Database Schema
2. Connecting to databases
3. Creating tables
4. Inserting Records
5. Retrieving Records
6. Content update
7. Deleting table entries

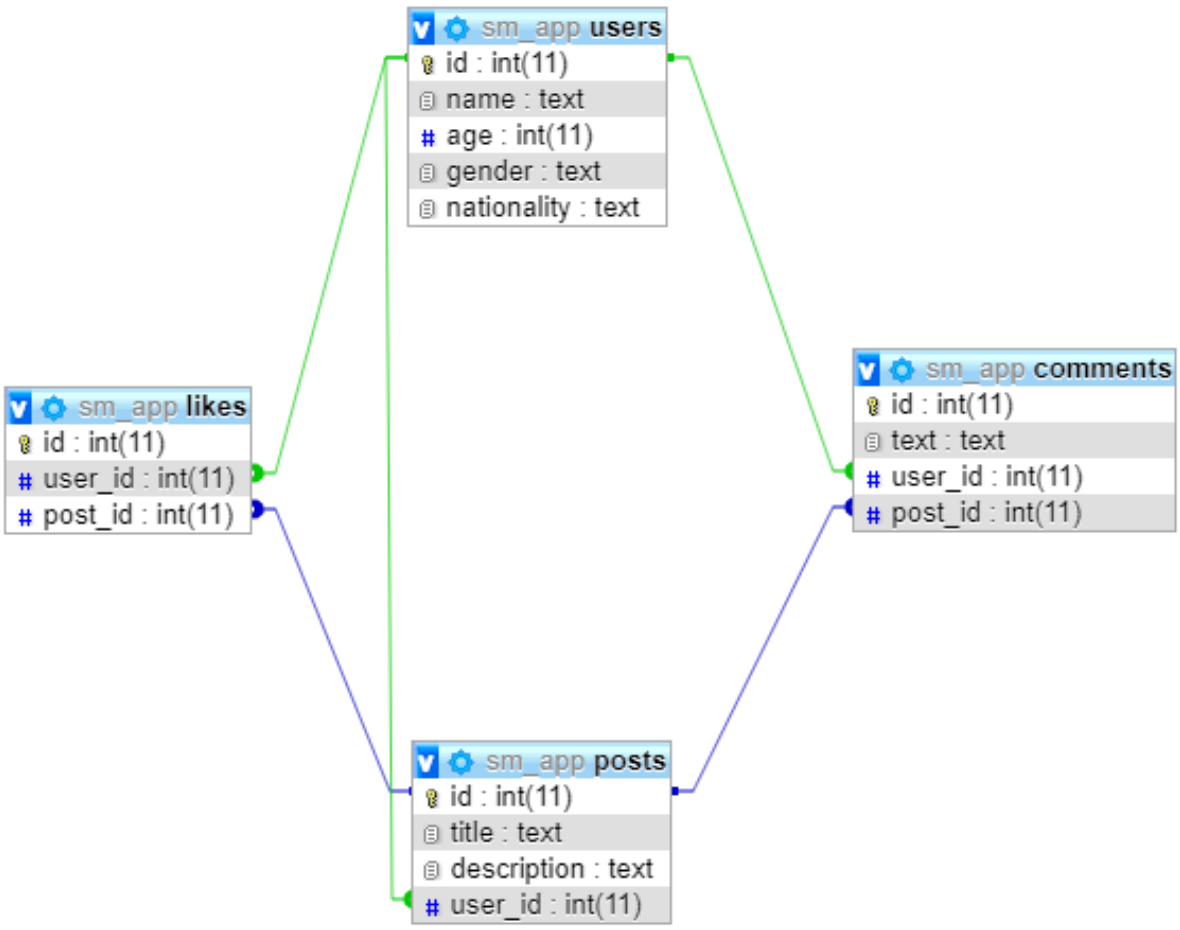
Each section has three subsections: SQLite, MySQL, and PostgreSQL.

1. Database schema for training

In this tutorial, we will develop a very small database of a social media application. The database will consist of four tables:

1. users
2. posts
3. comments
4. likes

The database schema is shown in the figure below.



Users (users) and publications (posts) will be of the type of relationship one-to-many: one reader can like several posts. Similarly, the same user can leave many comments (comments), and one

post can have several comments. Thus both users and posts have the same relationship type with respect to comments. And likes are identical to comments in this regard.

2. Connecting to databases

Before interacting with any database through the SQL library, it must be contacted. In this section, we'll look at how to connect from a Python application to databases.SQLite,MySQLandPostgreSQL. We recommend making your own .py file for each of the three databases.

Note. To complete the sections onMySQLandPostgreSQLyou need to start the corresponding servers yourself. For a quick guide on how to start a MySQL server, check out the MySQL section in the post.Starting a Django Project(English). To learn how to create a database in PostgreSQL, go to Setting Up a Database in the postPreventing SQL injection attacks with Python(English).

PostgreSQL

As with MySQL, there is no module in the Python standard library for PostgreSQL to interact with the database. But for this task there is a solution - the psycopg2 module:

```
pip install psycopg2
```

Let's define the create_connection() function to connect to a PostgreSQL database:

```
frompsycopg2importOperationalError

def create_connection(db_name, db_user, db_password, db_host, db_port):
connection=None
 try:
connection = psycopg2.connect(
database=db_name,
user=db_user,
password=db_password,
host=db_host,
port=db_port,
)
print("Connection to PostgreSQL DB successful")
 exceptOperationalErrorase:
print(f"The error '{e}' occurred")
 returnconnections
```

The connection is made through the psycopg2.connect() interface. Next, we use the function we wrote:

```
connection = create_connection(
 "postgres","postgres","abc123","127.0.0.1","5432"
)
```

Now, inside the default postgres database, we need to create the sm_app database. The corresponding create_database() function is defined below:

```
def create_database(connection, query):
connection.autocommit =True
cursor = connection.cursor()
 try:
cursor.execute(query)
print("Query executed successfully")
 exceptOperationalErrorase:
```

```
print(f"The error '{e}' occurred")

create_database_query ="CREATE DATABASE sm_app"
create_database(connection, create_database_query)
```

By running the above script, we will see the sm_app database on our PostgreSQL server. Let's connect to it:

```
connection = create_connection(
 "sm_app","postgres","abc123","127.0.0.1","5432"
)
```

Here 127.0.0.1 and 5432 are respectively the IP address and port of the server host.

3. Create tables

In the previous section, we saw how to connect to SQLite, MySQL, and PostgreSQL database servers using different Python libraries. We have created the sm_app database on all three DB servers. In this section, we'll look at how to create tables within these three databases.

As discussed earlier, we need to fetch and link four tables:

1. users
2. posts
3. comments
4. likes

PostgreSQL

Using the psycopg2 library in execute_query() also involves working with a cursor:

```
def execute_query(connection, query):
connection.autocommit =True
cursor = connection.cursor()
 try:
cursor.execute(query)
print("Query executed successfully")
 exceptOperationalErrorase:
print(f"The error '{e}' occurred")
```

We can use this feature to organize tables, insert, modify and delete records in your PostgreSQL database.

Let's create a users table inside the sm_app database:

```
create_users_table ="""
CREATE TABLE IF NOT EXISTS users (
id SERIAL PRIMARY KEY,
name TEXT NOT NULL,
age INTEGER,
gender TEXT,
nationality TEXT
)
"""
```

```
execute_query(connection, create_users_table)
```

The query to create a users table in PostgreSQL is slightly different from SQLite and MySQL. Here, the SERIAL keyword is used to specify auto-increment columns. In addition, the way to specify references to foreign keys is different:

```
create_posts_table ="""
CREATE TABLE IF NOT EXISTS posts (
id SERIAL PRIMARY KEY,
title TEXT NOT NULL,
description TEXT NOT NULL,
user_id INTEGER REFERENCES users(id)
)
"""

execute_query(connection, create_posts_table)
```

4. Paste records

In the previous section, we covered how to deploy tables in SQLite, MySQL, and PostgreSQL databases using various Python modules. In this section, we will learn how to insert records.

PostgreSQL

In the previous subsection, we saw two approaches for inserting records into MySQL database tables. Psycopg2 uses the second approach: we pass an SQL query with placeholders and a list of records to the execute() method. Each entry in the list must be a tuple whose values correspond to the values of a column in the database table. This is how we can insert user records into the users table:

```
users = [
("James",25,"male",USA),
("Leila",32,"female",France),
(Brigitte,35,"female","England"),
(Mike,40,"male","Denmark"),
("Elizabeth",21,"female",Canada),
]

user_records =", ".join(["%s"] * len(users))

insert_query = (
 f"INSERT INTO users (name, age, gender, nationality) VALUES {user_records}"
)

connection.autocommit =True
cursor = connection.cursor()
cursor.execute(insert_query, users)
```

The users list contains five user entries as tuples. We then create a string with five placeholder elements (%s) corresponding to the five user entries. The placeholder string is combined with a query that inserts records into the users table. Finally, the query string and user entries are passed to the execute() method.

The following script inserts records into the posts table:

```
posts = [
("Happy","I am feeling very happy today",one),
("Hot Weather","The weather is very hot today",2),
(Help,"I need some help with my work",2),
("Great News","I'm getting married",one),
("Interesting Game","It was a fantastic game of tennis",5),
("Party","Anyone up for a late-night party today?",3),
]

post_records =", ".join(["%s"] * len(posts))

insert_query = (
 f"INSERT INTO posts (title, description, user_id) VALUES {post_records}"
)

connection.autocommit =True
cursor = connection.cursor()
cursor.execute(insert_query, posts)
```

Using the same technique, you can insert records into the comments and likes tables.

5. Extracting data from records

PostgreSQL

The process of selecting records from a PostgreSQL table using the psycopg2 module is also similar to SQLite and MySQL. We use cursor.execute() again, then the fetchall() method to select records from the table. The following script selects all records from the users table:

```
def execute_read_query(connection, query):
cursor = connection.cursor()
result =None
 try:
cursor.execute(query)
result = cursor.fetchall()
 returnresult
 exceptOperationalErrorase:
print(f"The error '{e}' occurred")

select_users ="SELECT * FROM users"
users = execute_read_query(connection, select_users)

foruserinusers:
print(user)
```

6. Update table entries

PostgreSQL

PostgreSQL update query is similar to SQLite and MySQL.

7. Deleting table entries

SQLite

As an example, let's remove a comment with an id of 5:

```
delete_comment ="DELETE FROM comments WHERE id = 5"
execute_query(connection, delete_comment)
```

Now, if we retrieve all the records from the comments table, we can see that the fifth comment has been removed. The delete process in MySQL and PostgreSQL is identical to SQLite:

Conclusion

In this tutorial, we figured out how to use three common Python libraries to work with relational databases. Having learned to work with one of the sqlite3, mysql-connector-python and psycopg2 modules, you can easily transfer your knowledge to other modules and operate on any of the SQLite, MySQL and PostgreSQL databases.

There are also libraries for working with SQL andobject-relational mappings, such asSQLAlchemyandDjango ORM, which automate the tasks of Python interacting with databases.

If you are interested in working with databases using Python, write about it in the comments - we will prepare additional materials.