Python y BD: sqlite

1. Intro

> SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. More Information...

Python incluye soporte para SQLite nativo. En nuestro caso usaremos inicialmente sqlite como ejemplo de acceso desde Python a bases de datos. Y mostraremos un ataque de inyección SQL (SQLi) y como evitarlo.

2. Instalar sqlite

Puede ser necesario instalar `sqlite3`:

```
alu@xdebian11:~/pps-clase/src_sqlite$ sqlite3
bash: sqlite3: orden no encontrada
alu@xdebian11:~/pps-clase/src_sqlite$
```

En ese caso

```
alu@xdebian11:~/pps-clase/src_sqlite$ su
Contraseña:
root@xdebian11:/home/alu/pps-clase/src_sqlite# apt update
root@xdebian11:/home/alu/pps-clase/src_sqlite# apt install sqlite3
Leyendo lista de paquetes... Hecho
...
```

```
Configurando sqlite3 (3.34.1-3) ...
Procesando disparadores para man-db (2.9.4-2) ..
root@xdebian11:/home/alu/pps-clase/src_sqlite# exit
exit
alu@xdebian11:~/pps-clase/src_sqlite$ sqlite3 --version
3.34.1 2021-01-20 14:10:07 ....
alu@xdebian11:~/pps-clase/src_sqlite$
```

3. Un poco de SQL

No es el objetivo de este tema aprender SQL. Usaremos consultas sencillas para básicamante crear tablas simples, insertar información inicial o hacer búsquedas. Si desea un resumen rápido use este enlace: https://diego.com.es/sql-principios-basicos

4. Comandos sqlite3

Si quiere acceder a una BD desde línea de comandos puede usar el comando `sqlite3`. Desde esa aplicación podemos consultar el schema de la BD, las tablas o hacer consultas SQL. En el ejemplo se aprecia como abrir una BD de datos (`ejemplo-basico-02.db`), ver su esquema, sus tablas y hacer una consulta sobre la tabla `users`:

```
alu@xdebian11:~/pps-clase/src_sqlite$ sqlite3
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ejemplo-basico-02.db
sqlite> .schema
CREATE TABLE users(
    id INTEGER PRIMARY KEY,
    email TEXT,
    username TEXT,
    password TEXT);
sqlite> .tables
users
sqlite> SELECT * FROM users;
1|artist@local.com|artist|1234
2|boss@local.com|boss|123456
3|carpet@local.com|carpet|123478

sqlite> .quit
(venv) @tos:~/python-sqlite$
```

Puede invocar sqlite3 con el nombre de la BD y no sería preciso el comando `.open`:

```
alu@xdebian11:~/pps-clase/src_sqlite$ sqlite3 ejemplo-basico-02.db
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
sqlite> .tables
```

```
users
sqlite> .quit
alu@xdebian11:~/pps-clase/src_sqlite$
```

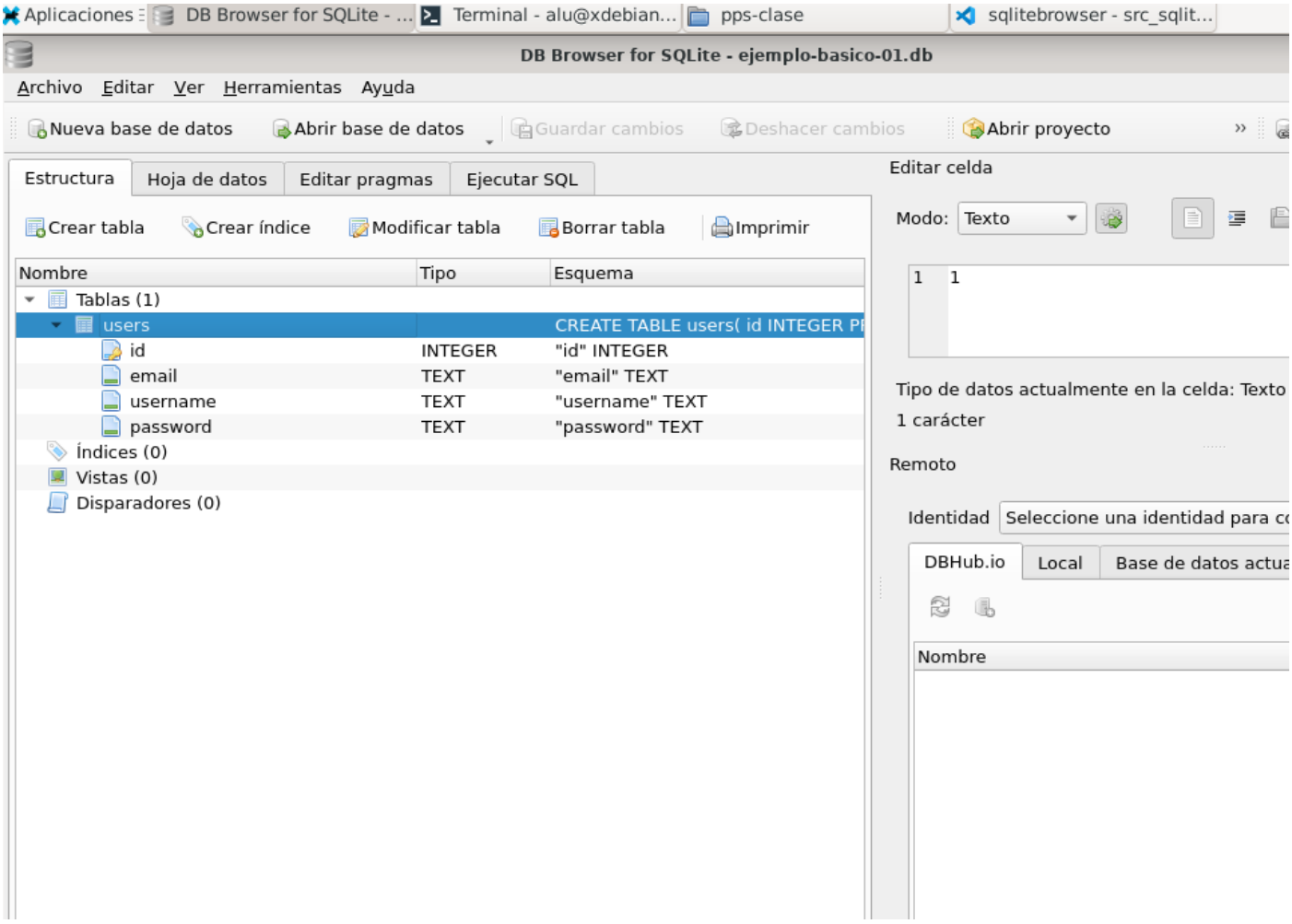
4.1. Entorno gráfico

Aunque no es imprescindible en esta práctica puede usar un interfaz gráfico para acceder a las BD, por ejemplo `sqlitebrowser`

```
alu@xdebian11:~/pps-clase/src_sqlite$ sudo apt install sqlitebrowser
...
alu@xdebian11:~/pps-clase/src_sqlite$
```

Y para invocarlo:

```
...
@tos:~/python-sqlite$ sqlitebrowser ejemplo-basico-01.db
```



5. Ejemplo inicial

Todo el código de este apartado se encuentra en el fichero `ejemplo-basico-01.py`. Vamos a analizarlo paso a paso.

Para acceder a Bases de Datos desde nuestro programa Python necesitamos importar el módulo `sqlite`. Además nos conectamos a la BD usando el método `connect`, que nos devuelve un objeto del tipo `Connection`. A partir de ese objeto obtenemos un puntero del tipo `Cursor` que será el que se utiliza para acceder a la BD y hacer consultas:

```
import sqlite3
# print("sqlite version: ", sqlite3.version)

con = sqlite3.connect('ejemplo-basico-01.db')
cursor = con.cursor()
```

Para ejecutar una consulta usamos el objeto `cursor` y el método `execute` al que le pasamos la cadena SQL. Tras la consulta, si esta implica cambios en la BD (añadir, modificar o borrar) usamos el método `commit` para que los datos se registren en la BD.

En este fragmento de código la consulta SQL borra la tabla `users` de la Base de Datos (si existía).

```
# pasamos directamente string con SQL a ejecutar
cursor.execute("DROP TABLE IF EXISTS users")
con.commit()
```

Para crear una tabla hacemos algo similar. En este ejemplo se utiliza una variable auxiliar (`sql_crea_tabla`) para almacenar la consulta utiliza `execute`

```
sql_crea_tabla = '''
          CREATE TABLE IF NOT EXISTS users(
              id INTEGER PRIMARY KEY,
              email TEXT,
              username TEXT,
              password TEXT)
          '''

cursor.execute(sql_crea_tabla)
con.commit()
```

Para introducir datos iniciales en la tabla `users` usamos otra consulta del tipo `INSERT`

```
# pasamos directamente string con SQL a ejecutar
cursor.execute('''
          INSERT INTO users (id, email, username, password)
                VALUES
                (1,'artist@local.com', 'artist', '1234'),
                (2,'boss@local.com', 'boss', '123456'),
```

```
                (3,'carpet@local.com', 'carpet', '123478')
            ''')
con.commit()
```

Para hacer búsquedas de información se utilizan consultas SQL del tipo `SELECT`. Los resultados se obtienen con el método `fetchall()`, que devuelve una lista que podemos recorrer con un `for`.

```
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

print("Variable rows que retorna fetchall")
print(rows)
# print(type(rows))

print("Recorremos rows con for")
for row in rows:
    print("Fila: ", row)
```

Si ejecutamos el fichero `ejemplo-basico-01.py`.

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-01.py
Variable rows que retorna fetchall
[(1, 'artist@local.com', 'artist', '1234'), (2, 'boss@local.com', 'boss',
'123456'), (3, 'carpet@local.com', 'carpet', '123478')]
Recorremos rows con for
Fila:  (1, 'artist@local.com', 'artist', '1234')
Fila:  (2, 'boss@local.com', 'boss', '123456')
Fila:  (3, 'carpet@local.com', 'carpet', '123478')
alu@xdebian11:~/pps-clase/src_sqlite$
```

6. Consultas SQL desde fichero

Todo el código de este apartado se encuentra en el fichero `ejemplo-basico-02.py`. Vamos a analizarlo paso a paso.

En este caso hacemos una modificación al programa para leer la información de las consultas desde ficheror de texto. Tenemos dos ficheros:

- uno con la consulta de creación de la tabla
- y otro con la consulta que rellena con información

Ficheros `ejemplo-basico-02_crea_tabla.sql` y `ejemplo-basico-02_rellena_tabla.sql`:

```
alu@xdebian11:~/pps-clase/src_sqlite$ cat ejemplo-basico-02_crea_tabla.sql
CREATE TABLE IF NOT EXISTS users(
    id INTEGER PRIMARY KEY,
    email TEXT,
    username TEXT,
```

```
    password TEXT);

alu@xdebian11:~/pps-clase/src_sqlite$ cat ejemplo-basico-02_rellena_tabla.sql
INSERT INTO users (id, email, username, password)
    VALUES
    (1,'artist@local.com', 'artist', '1234'),
    (2,'boss@local.com', 'boss', '123456'),
    (3,'carpet@local.com', 'carpet', '123478');
alu@xdebian11:~/pps-clase/src_sqlite$
```

Y modificamos el código para que las consultas necesarias se hagan utilizando el contenido de dichos ficheros:

- leemos el contenido
- se lo asignamos a variables.
- con esas variables solo resta hacer `execute` y `commit`.

La lectura desde fichero se hace en un bloque `with`, y se usan los métodos `open` y `read`:

```
try:
    with open("ejemplo-basico-02_crea_tabla.sql") as sql_file:
        crea_sql_as_string = sql_file.read()
    with open("ejemplo-basico-02_rellena_tabla.sql") as sql_file:
        rellena_sql_as_string = sql_file.read()
except:
    print("error al abrir alguno de los ficheros")
    exit(1)

cursor.execute(crea_sql_as_string)
cursor.execute(rellena_sql_as_string)
con.commit()
```

El resto del código se puede mantener igual. Si ejecutamos:

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-02.py
Variable rows que retorna fetchall
[(1, 'artist@local.com', 'artist', '1234'), (2, 'boss@local.com', 'boss',
'123456'), (3, 'carpet@local.com', 'carpet', '123478')]
Recorremos rows con for
Fila:  (1, 'artist@local.com', 'artist', '1234')
Fila:  (2, 'boss@local.com', 'boss', '123456')
Fila:  (3, 'carpet@local.com', 'carpet', '123478')
alu@xdebian11:~/pps-clase/src_sqlite$
```

7. Buscar en BD a partir de datos entrada

Todo el código de este apartado se encuentra en el fichero `ejemplo-basico-03.py`. Vamos a analizarlo paso a paso.

La idea de este ejemplo es que el usuario introduzca un usuario y su clave para mostrar la información de dicho usuario almacenada en la tabla `users`. Para que se muestre la información el usuario y clave introducidad deben coincidir con las de la tabla.

Para ello solicitamos por entrada estándar esos dos valores, y construimos una consulta SQL a partir de esos valores:

```
cadena = input("username a buscar: ")
password = input("password de ese usuario: ")

sql_buscar = "SELECT * FROM users WHERE username='"+cadena + "' AND
password='" + password +"'"
cursor.execute(sql_buscar)
print("Resultados con SQLi: ")
print(cursor.fetchall())
```

Puede probar el programa con usuario y clave correctos, y con usuario y claves incorrectos:

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: artist
password de ese usuario: 1234
Resultados con SQLi:
[(1, 'artist@local.com', 'artist', '1234')]
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: artist
password de ese usuario: abcd
Resultados con SQLi:
[]
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: artista
password de ese usuario: 1234
Resultados con SQLi:
[]
alu@xdebian11:~/pps-clase/src_sqlite$
```

8. Inyección SQL

Para comprobar si en el ejemplo anterior la consulta es vulnerable a SQLi puede introducir como usuario la cadena `"' OR 1=1 --"`:

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: ' OR 1=1 -- '
password de ese usuario: cualquiera
Resultados con SQLi:
[(1, 'artist@local.com', 'artist', '1234'), (2, 'boss@local.com', 'boss',
'123456'), (3, 'carpet@local.com', 'carpet', '123478')]
alu@xdebian11:~/pps-clase/src_sqlite$
```

Observe que a pesar de que ni el usuario ni la clave introducidas son correctos se han mostrado todos los registros de la tabla con usuarios, claves y correos.

9. Solución: consultas parametrizadas

Como hemos comprobado, la concatenación de strings en una consulta puede ser susceptible de ataques de inyección SQL. Para solucionarlo se usan las consultas **parametrizadas**.

Para ello:

- Se prepara la consulta en forma de plantilla, indicando dónde irían los "parámetros" con el carácter `?`.
- Se construyen los `data` que formarán los parámetros
- Se invoca `execute` que recibe ahora dos parámetros: la consulta parametrizada y su "parámetros":

```
# Solución con placeholder
## Se construye la plantilla con ?
sql_buscar = "SELECT * FROM users WHERE username=? AND password=?"
## Se construye `data` con  los parámetros como un tupla
data = (cadena, password)
print(type(data))

## Se invoca execute con plantilla y parámetros
cursor.execute(sql_buscar, data)
print("Resultados con SQLi y consulta parametrizada: ")
print(cursor.fetchall())
```

Si ejecutamos nuestro programa añadiendo esas sentencias y con valores "normales" vemos que los resultados son los mismos:

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: artist
password de ese usuario: 1234
Resultados con SQLi:
[(1, 'artist@local.com', 'artist', '1234')]
<class 'tuple'>
Resultados con SQLi y consulta parametrizada:
[(1, 'artist@local.com', 'artist', '1234')]
alu@xdebian11:~/pps-clase/src_sqlite$
```

Pero al intentar la inyección SQL vemos que con esta solución ya no es vulnerable:

```
alu@xdebian11:~/pps-clase/src_sqlite$ python3 ejemplo-basico-03.py
username a buscar: ## Se invoca execute con plantilla y parámetros
password de ese usuario: cualquiera
Resultados con SQLi:
[(1, 'artist@local.com', 'artist', '1234'), (2, 'boss@local.com', 'boss',
'123456'), (3, 'carpet@local.com', 'carpet', '123478')]
<class 'tuple'>
```

```
Resultados con SQLi y consulta parametrizada:
[]
alu@xdebian11:~/pps-clase/src_sqlite$
```

Tambien se puede parametrizar usando *placeholders* con nombre `:namePlaceHolder` en vez del comodín `?`

```
# Solución con placeholder con nombres

## Se construye la plantilla con :nombres
sql_buscar = "SELECT * FROM users \
              WHERE username=:usuario AND password=:clave"

## Se construye `data` con  los parámetros como diccionario
data = {
    'usuario': cadena,
    'clave': password
}
print(type(data))

## Se invoca execute con plantilla y parámetros
cursor.execute(sql_buscar, data)

print("Resultados con SQLi y consulta parametrizada con nombres: ")
print(cursor.fetchall())
```

Puede comprobar que el resultado es el mismo.

10. Ejercicio: CRUD

> CRUD: Create, Read, Update and Delete.

Se propone como ejercicio realizar un UPDATE y un DELETE. Además se propone el uso del módulo de python `Faker` y una pequeña mejora en la creación de la tabla.

10.1. Apartado 1: añadir al programa el borrado de un elemento

El SQL necesario sería algo similar a

```
DELETE FROM users WHERE id =?
```

10.2. Apartado 2: añadir al programa la actualización de un elemento

Por ejemplo, en el caso del email podría ser:

```
UPDATE users SET email = ?  WHERE id = ?
```

10.3. Apartado 3: mejorando nuestra BD

En la creación de la tabla no nos hemos complicado mucho:

```
CREATE TABLE IF NOT EXISTS users(
    id INTEGER PRIMARY KEY,
    email TEXT,
    username TEXT,
    password TEXT);
```

Modifique la creación de la tabla para que

- el campo email no pueda estar vacío
- el campo email debe ser único
- limite la lóngitud máxima del username y el password usando el tipo varchar.

10.4. Apartado 4. Rellenar datos de usuario con Faker.

El módulo `faker` puede ser útil para generar datos aleatorios al rellenar listas, bases de datos, etc.

> Faker is a Python package that generates fake data for you. Whether you need to bootstrap your database, create good-looking XML documents, fill-in your persistence to stress test it, or anonymize data taken from a production service, Faker is for you.
>
> https://faker.readthedocs.io/en/master/

Es necesario instalarlo con `pip` (y mejor en un entorno virtual). Observe un ejemplo de uso del módulo:

```
from faker import Faker

fake = Faker()
for i in range(1, 10):
    print(f"{i}: {fake.user_name()} {fake.company_email()} ")
```

Tiene más ejemplos de uso de `faker` en este enlace: https://zetcode.com/python/faker/

10.5. Ejemplo de ejecución

Observe una posible ejecución del ejercicio propuesto:

```
(venv) alu@xdebian11:~/pps-clase/src_sqlite/sol$ python ejercicio-sqlite.py
(1, 'artist@local.com', 'artist', '1234')
(2, 'boss@local.com', 'boss', '123456')
(3, 'carpet@local.com', 'carpet', '123478')
# Ejercicio parte 1: delete
id a borrar: 2
(1, 'artist@local.com', 'artist', '1234')
(3, 'carpet@local.com', 'carpet', '123478')
# Ejercicio parte 2: update
```

```
id a modificar: 3
Nueva password: qwerty
(1, 'artist@local.com', 'artist', '1234')
(3, 'carpet@local.com', 'carpet', 'qwerty')
# Ejercicio parte 3: mejora tabla creada: muestra campos DESDE PYTHON con
PRAGMA
(0, 'id', 'INTEGER', 0, None, 1)
(1, 'email', 'TEXT', 1, None, 0)
(2, 'username', 'VARCHAR(25)', 0, None, 0)
(3, 'password', 'VARCHAR(32)', 0, None, 0)
# Ejercicio parte 4: uso faker, añado 10 usuarios
(1, 'artist@local.com', 'artist', '1234')
(3, 'carpet@local.com', 'carpet', 'qwerty')
(4, 'whitekristin@example.org', 'thompsonchristopher', 'aFAzZmNl#5')
(5, 'amydouglas@example.net', 'hartjennifer', '0$R8z8Nq1N')
(6, 'michelle57@example.org', 'farmerheidi', '1^24DGa1Va')
(7, 'qalexander@example.com', 'michelle50', 'c_#N%$)u$2')
(8, 'gibsonbradley@example.org', 'seanwilliams', 'Ywf*2DoqYB')
(9, 'watkinschristopher@example.net', 'patrick07', 'Gl02Ls0Qc@')
(10, 'creynolds@example.org', 'agarcia', 'rI^Zf3WjdL')
(11, 'rosematthew@example.net', 'hbush', '5kDr5h$i+z')
(12, 'erinmeyer@example.net', 'cnguyen', 'S_h4gRenSl')
(venv) alu@xdebian11:~/pps-clase/src_sqlite/sol$
```

> Nota: Para mostrar la estructura de la tabla desde Python puede ejecutar este código:
>
> ```
> print("# Muestra campos tabla users")
> schema = cursor.execute("PRAGMA table_info('users')")
> for r in schema:
>    print(r)
> ```

11. Enlaces adicionales

- Python SQLite3
  - Tutorial: https://www.devdungeon.com/content/python-sqlite3-tutorial
  - Python programming tutorial for the SQLite database: https://zetcode.com/python/sqlite/
- SQlite web: https://www.sqlite.org/index.html
- SQL:
  - Tutorial W3schools: https://www.w3schools.com/sql/
  - Tutorial SQL en Español interactivo con intérprete SQL: https://www.sql-easy.com/es/tutorial/