

1. Introducción

Hasta ahora, todos los programas que hemos escrito comparten una característica común: cuando el programa termina, los datos que estaba manipulando **desaparecen**. Las variables, los objetos y las estructuras de datos viven únicamente en la memoria del proceso y se liberan cuando este finaliza. Esto es aceptable para programas pequeños o para cálculos puntuales, pero resulta inviable para cualquier aplicación que necesite conservar información entre ejecuciones: un juego que guarda puntajes, un sistema que registra usuarios, una tienda que mantiene un catálogo.

A la capacidad de conservar datos más allá del tiempo de vida del programa se la denomina **persistencia**. Existen muchas formas de lograr persistencia —archivos de texto, archivos binarios, bases de datos—, pero en el desarrollo profesional la más utilizada es, por amplio margen, la **base de datos relacional**. Este apunte introduce los conceptos básicos para acceder a una base de datos desde **Python**, poniendo especial énfasis en las prácticas de seguridad que todo desarrollador debe conocer.

2. Bases de datos relacionales

Una **base de datos relacional** organiza la información en **tablas**. Cada tabla está compuesta por **filas** (también llamadas registros) y **columnas** (también llamadas campos). Cada fila representa una entidad del mundo real, y cada columna representa un atributo de esa entidad.

Por ejemplo, una tabla **jugadores** podría verse así:

id	nombre	puntaje
1	Arturo	9500
2	Lancelot	8200
3	Merlin	7800

Cada tabla debe tener una **clave primaria** (*primary key*): una columna (o conjunto de columnas) que identifica de manera única a cada fila. En el ejemplo, la columna **id** cumple ese rol. Las claves primarias garantizan que no haya dos filas duplicadas y permiten referirse a una fila específica de manera inequívoca.

Además de las claves primarias, existen las **claves foráneas** (*foreign keys*): columnas que referencian la clave primaria de otra tabla, estableciendo una relación entre ambas. Por ejemplo, si tuviéramos una tabla **partidas** y quisiéramos indicar qué jugador participó en cada una, podríamos tener una columna **jugador_id** que apunte al **id** correspondiente en **jugadores**.

El lenguaje estándar para consultar y manipular bases de datos relacionales es **SQL** (*Structured Query Language*). SQL permite crear tablas, insertar datos, consultarlos, modificarlos y eliminarlos. Los cuatro tipos de operación básicos se conocen con el acrónimo **CRUD**:

- Create: **INSERT**
- Read: **SELECT**
- Update: **UPDATE**
- Delete: **DELETE**

Existen múltiples motores de bases de datos relacionales, tanto comerciales como de código abierto. Entre los más utilizados se encuentran MySQL, PostgreSQL, SQLite y SQL Server. En este curso trabajaremos con **MySQL**, que es software libre, ampliamente utilizado y con excelente soporte desde **Python**.

3. Conexión desde Python

Para conectarse a una base de datos MySQL desde **Python** se necesita un **conector**: una biblioteca que implementa el protocolo de comunicación con el motor. Existen varios conectores disponibles; en este curso utilizaremos **mysql-connector-python**, mantenido por el propio equipo de MySQL¹.

El uso básico del conector sigue una secuencia muy clara:

1. Abrir una **conexión** al motor, especificando host, usuario, contraseña y base de datos.

¹Documentación oficial: <https://dev.mysql.com/doc/connector-python/en/>

2. Crear un **cursor**, que es el objeto que permite enviar consultas y recibir resultados.
3. Ejecutar la consulta deseada a través del cursor.
4. Recuperar los resultados (en el caso de las consultas de lectura).
5. Confirmar los cambios (en el caso de escrituras).
6. Cerrar el cursor y la conexión.

Un ejemplo mínimo se ve así:

```
1 import mysql.connector
2
3 conexion = mysql.connector.connect(
4     host="localhost",
5     user="root",
6     password="***",
7     database="juego"
8 )
9
10 cursor = conexion.cursor()
11 cursor.execute(
12     "SELECT nombre, puntaje FROM jugadores ORDER BY puntaje DESC")
13
14 for nombre, puntaje in cursor.fetchall():
15     print(f"{nombre}: {puntaje}")
16
17 cursor.close()
18 conexion.close()
```

El método `fetchall()` devuelve una lista con todas las filas del resultado. Cada fila es una tupla cuyos elementos se corresponden con las columnas seleccionadas en la consulta.

Para las operaciones de escritura (**INSERT**, **UPDATE**, **DELETE**) es necesario invocar `conexion.commit()` para que los cambios se persistan efectivamente en la base de datos. Si no se ejecuta `commit()`, los cambios se pierden al cerrar la conexión.

```
1 cursor = conexion.cursor()
2 cursor.execute(
3     "INSERT INTO jugadores (nombre, puntaje) VALUES (%s, %s)",
4     ("Galahad", 7500)
5 )
6 conexion.commit()
```

Notar el uso de `%s` como marcador de posición en la consulta. Este detalle es fundamental y lo analizamos en la próxima sección.

4. El problema de la inyección SQL

Supongamos que queremos permitir que el usuario ingrese un nombre y consultar todos los jugadores con ese nombre. Una primera aproximación, ingenua, consistiría en construir la consulta concatenando cadenas:

```
1 nombre = input("Ingrese el nombre a buscar: ")
2 consulta = f"SELECT * FROM jugadores WHERE nombre = '{nombre}'"
3 cursor.execute(consulta)
```

Este código parece funcionar, pero contiene una vulnerabilidad crítica conocida como **inyección SQL** (*SQL injection*). Si el usuario ingresa como nombre el texto:

```
' OR '1'='1'
```

la consulta final queda así:

```
SELECT * FROM jugadores WHERE nombre = '' OR '1'='1'
```

La condición `'1'='1'` es siempre verdadera, por lo que la consulta devuelve **todas** las filas de la tabla, revelando información que no debería estar accesible. Peor aún: con una entrada adecuadamente construida, un atacante puede llegar a ejecutar consultas destructivas como `DROP TABLE`, o acceder a datos de otras tablas. La inyección SQL es, desde hace décadas, una de las vulnerabilidades más explotadas en aplicaciones reales².

4.1. La solución: bind values

La manera correcta de construir consultas con datos variables es usar **valores vinculados** (*bind values*): marcadores de posición que el conector reemplaza de manera segura. La consulta se escribe con `%s` en los lugares donde irán los datos, y estos se pasan como segundo argumento de `execute()`:

```
1 nombre = input("Ingrese el nombre a buscar: ")
2 cursor.execute(
3     "SELECT * FROM jugadores WHERE nombre = %s",
4     (nombre,)
5 )
```

²OWASP Foundation. *SQL Injection*. https://owasp.org/www-community/attacks/SQL_Injection. OWASP mantiene una lista de las diez vulnerabilidades más críticas en aplicaciones web, y la inyección SQL aparece recurrentemente en ella.

La diferencia es sutil en la sintaxis pero enorme en seguridad. Cuando se usan *bind values*, el conector se encarga de **escapar** los caracteres especiales del valor antes de insertarlo en la consulta. El texto `' OR '1'='1'` se trata como un valor literal, no como código SQL, y la consulta devuelve cero filas en lugar de todas.

Forma	Seguridad
Concatenación con f-strings o +	Vulnerable a inyección SQL.
Uso de %s con tupla	Seguro. El conector escapa los valores.

Como regla general, **nunca** se deben concatenar datos provenientes del usuario directamente en una consulta SQL. Siempre se usan *bind values*, sin excepciones.

5. Manejo de credenciales

En los ejemplos anteriores aparecen las credenciales de la base de datos (**host**, **user**, **password**, **database**) directamente escritas en el código fuente. Esto es aceptable en un ejemplo didáctico, pero constituye una mala práctica en cualquier proyecto real.

Si las credenciales están en el código, cualquiera que acceda al repositorio —un compañero de equipo, un colaborador externo, un atacante que comprometió el sistema— tiene acceso a la base de datos. Peor aún: si el repositorio es público (por ejemplo, en GitHub), las credenciales quedan expuestas a todo internet.

La práctica recomendada es almacenar las credenciales en **variables de entorno**: valores que se configuran fuera del código, en el sistema operativo o en un archivo que no se versiona. En **Python** se accede a ellas a través del módulo **os**:

```
1 import os
2 import mysql.connector
3
4 conexion = mysql.connector.connect(
5     host=os.environ["DB_HOST"],
6     user=os.environ["DB_USER"],
7     password=os.environ["DB_PASSWORD"],
8     database=os.environ["DB_NAME"]
9 )
```

Para facilitar la carga de estas variables durante el desarrollo, es común usar la biblioteca **python-dotenv**, que lee un archivo **.env** ubicado en la raíz del proyecto. Este archivo

contiene las credenciales en formato **CLAVE=valor** y **debe agregarse al .gitignore**, para que nunca se suba al repositorio.

```
# archivo .env (NO se versiona)
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=miclave
DB_NAME=juego
```

```
1 from dotenv import load_dotenv
2 load_dotenv() # carga las variables del archivo .env
```

De esta manera, el código puede subirse a un repositorio sin exponer información sensible, y cada entorno (desarrollo, pruebas, producción) puede tener sus propias credenciales.

6. Hash de contraseñas

Un aspecto relacionado con la seguridad es el tratamiento de las contraseñas de los usuarios. Cuando una aplicación permite que los usuarios se registren y se autenticquen, surge la pregunta: ¿cómo se almacenan las contraseñas en la base de datos?

La respuesta **nunca** debe ser “tal cual las escribió el usuario”. Guardar contraseñas en texto plano es una de las peores prácticas posibles: si la base de datos se ve comprometida, todas las contraseñas quedan expuestas, y como muchos usuarios reutilizan la misma contraseña en otros servicios, el daño se multiplica.

La solución es aplicar una **función de hash**: un algoritmo que transforma una cadena de longitud arbitraria en una cadena de longitud fija, de manera que:

- Sea imposible recuperar el valor original a partir del hash (función unidireccional).
- Dos entradas distintas produzcan hashes distintos (con altísima probabilidad).
- La misma entrada produzca siempre el mismo hash.

Cuando un usuario se registra, se calcula el hash de su contraseña y se guarda ese hash. Cuando intenta iniciar sesión, se calcula el hash de la contraseña ingresada y se compara con el guardado. Si coinciden, las credenciales son correctas. La contraseña original nunca se almacena.

Sin embargo, no cualquier función de hash sirve para contraseñas. Algoritmos rápidos como MD5 o SHA-1 son inadecuados porque un atacante con acceso al hash puede probar millones de combinaciones por segundo. Para contraseñas se usan algoritmos diseñados para ser **lentos y costosos** computacionalmente, como `bcrypt`, `scrypt` o `argon2`. En **Python** se recomienda la biblioteca `bcrypt`:

```
1 import bcrypt
2
3 # Al registrar un usuario
4 clave = "clave_secreta".encode("utf-8")
5 hash_guardado = bcrypt.hashpw(clave, bcrypt.gensalt())
6 # Se almacena hash_guardado en la base de datos
7
8 # Al iniciar sesion
9 clave_ingresada = "clave_secreta".encode("utf-8")
10 if bcrypt.checkpw(clave_ingresada, hash_guardado):
11     print("Acceso concedido")
12 else:
13     print("Credenciales incorrectas")
```

El uso de `gensalt()` añade un valor aleatorio (el *salt*) al cálculo del hash, de modo que dos usuarios con la misma contraseña obtengan hashes distintos. Esto evita ataques basados en tablas precomputadas (*rainbow tables*).

7. El rol de una clase de acceso a datos

A medida que una aplicación crece, dispersar las operaciones de base de datos por todo el código se vuelve problemático: las consultas quedan repetidas, las credenciales aparecen en múltiples lugares y los cambios en el esquema obligan a modificar decenas de archivos. La solución es agrupar todas las operaciones de acceso a datos en una **clase especializada**.

Esta clase, la cual podríamos llamar `AccesoDB`³, tiene una única responsabilidad: **gestionar la comunicación con la base de datos**. Internamente maneja la conexión, los cursores, la ejecución de consultas, el cierre de recursos y la confirmación de cambios. Hacia afuera expone dos niveles de métodos:

- **Métodos de bajo nivel:** ejecutan una consulta SQL escrita explícitamente por el cliente, pero **siempre con bind values**. Son útiles cuando la operación no encaja

³La implementación completa de esta clase se desarrolla en las guías prácticas de la materia.

en la interfaz de alto nivel: consultas con **JOIN**, funciones de agregación, **ORDER BY** complejos, etc.

- **Métodos de alto nivel:** construyen la consulta a partir de estructuras de datos simples (listas y diccionarios), ocultando por completo el SQL. Por ejemplo, `insertar("jugadores", {"nombre": "Arturo", ...})` arma internamente el **INSERT** y ejecuta los bind values correspondientes.

Una decisión de diseño importante es que **las filas se devuelven como diccionarios**, no como tuplas. En el código, la conexión se configura para que sus cursores devuelvan diccionarios: `conn.cursor(dictionary=True)`. Esto tiene dos ventajas: por un lado, el código cliente accede a los campos por nombre (`fila["nombre"]`) en lugar de por posición, lo cual es mucho más legible y menos propenso a errores. Esto tiene dos ventajas: por un lado, el código cliente accede a los campos por nombre (`fila["nombre"]`) en lugar de por posición, lo cual es mucho más legible y menos propenso a errores. Por otro, un diccionario se serializa directamente a JSON sin necesidad de conversiones adicionales, lo cual será útil cuando construyamos APIs (**apunte 13**).

AccesoDB
-conn : Connection
+AccesoDB(host : String, user : String, password : String, database : String) -cursor() : Cursor
-construirWhere(filtro : Dictionary[]) : Tuple(String, List)
+ejecutarConsulta(query : String, valores : Tuple) : Dictionary[] +ejecutarComando(query : String, valores : Tuple) : Integer
+consultar(tabla : String, columnas : String[], filtro : Dictionary[]) : Dictionary[] +buscarUno(tabla : String, columnas : String[], filtro : Dictionary[]) : Dictionary
+insertar(tabla : String, datos : Dictionary) : Integer +insertarVarios(tabla : String, registros : Dictionary[]) : Integer
+actualizar(tabla : String, datos : Dictionary, filtro : Dictionary[]) : Integer
+eliminar(tabla : String, filtro : Dictionary[]) : Integer
+cerrar()

Además, la clase puede usarse como **context manager**, de modo que la conexión se cierre automáticamente al finalizar un bloque **with**, incluso si ocurre una excepción:

```
1 with AccesoDB(host, user, password, database) as db:
2     db.insertar("jugadores", {"nombre": "Arturo", "puntaje": 9500})
3     # ... mas operaciones ...
4 # al salir del bloque, la conexion se cierra sola
```

Lo cual se implementa con las funciones especiales `__enter__()` que debe devolver la instancia `self` y `__exit__()` que debe hacer lo necesario para limpiar cualquier proceso pendiente. En este caso ejecutar `self.cerrar()` para dejar cerrada la conexión.

Lo que esta clase **no** debe hacer:

- **Lógica de negocio.** No decide si un usuario puede registrarse, ni valida reglas del dominio. Solo persiste y recupera datos.
- **Presentación.** No formatea salidas para el usuario, ni genera HTML, ni imprime mensajes. Devuelve datos estructurados y quien la use decidirá qué hacer con ellos.
- **Decidir qué preguntar.** No decide cuándo consultar la base. Eso lo decide la capa que la usa.
- **Gestionar credenciales ni hashing.** Recibe la conexión ya resuelta y los datos ya procesados. El manejo de `.env` y el hashing de contraseñas son responsabilidad de otras capas, como vimos en las secciones anteriores.

Sobre esta clase genérica se puede construir una **fachada específica del dominio**, tal como vimos en el apunte de patrones de diseño. Por ejemplo, un `RepositorioJugadores` que internamente use un `AccesoDB` y exponga métodos con nombres del negocio: `registrar()`, `buscar_por_nombre()`, `actualizar_puntaje()`, `top_10()`. La composición entre ambas clases permite separar dos responsabilidades: `AccesoDB` sabe de MySQL pero no del dominio; el repositorio sabe del dominio pero no de SQL ni de cursores.

Este diseño se alinea con lo que ya vimos en el apunte de principios de diseño: aplicar **SRP** (una única responsabilidad por clase), **DIP** (la lógica de negocio no depende de la implementación concreta de la base de datos) y aprovechar el patrón **Façade** que estudiamos en el apunte de patrones.

Aun así, escribir esta clase a mano implica un trabajo considerable: cada modelo de dominio requiere métodos específicos, cada cambio en el esquema obliga a revisar las consultas, y la conversión entre filas y objetos se hace manualmente. En un próximo apunte sobre veremos que existen herramientas pensadas específicamente para automatizar el mapeo entre objetos y tablas, integrándose de forma natural con el paradigma orientado a objetos que venimos trabajando desde el comienzo del curso.

8. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial de MySQL Connector/Python:
<https://dev.mysql.com/doc/connector-python/en/>
- OWASP SQL Injection:
https://owasp.org/www-community/attacks/SQL_Injection
- Documentación de bcrypt:
<https://pypi.org/project/bcrypt/>
- Documentación de python-dotenv:
<https://pypi.org/project/python-dotenv/>