



POLITÉCNICA

Programación contra bases de datos en python

Bases de datos

Departamento de Sistemas Informáticos

E.T.S.I. de Sistemas Informáticos

Universidad Politécnica de Madrid



Índice

1. Arquitectura cliente-servidor
2. Drivers nativos
3. SQLAlchemy

ARQUITECTURA CLIENTE-SERVIDOR

Arquitectura cliente-servidor

- Las bases de datos funcionan de acuerdo con una arquitectura cliente-servidor.
- El servidor, que contiene los datos, escucha las peticiones de los clientes.
- Los clientes solicitan al servidor que realicen operaciones sobre los datos: creación, actualización, borrado y consulta de los datos.
- Habitualmente, el servidor y los clientes se ejecutan en dispositivos físicos diferentes.

MySQL Client/Server Protocol

Para comunicarse, el servidor y los clientes necesitan “hablar” el mismo idioma.

MySQL dispone de un protocolo que implementan tanto el servidor como los clientes para establecer la comunicación:

- Se denomina MySQL Client/Server Protocol.
- Se ejecuta sobre TCP.
- El cuerpo de los mensajes incluye sentencias SQL.

Más información en [la documentación de MySQL](#).

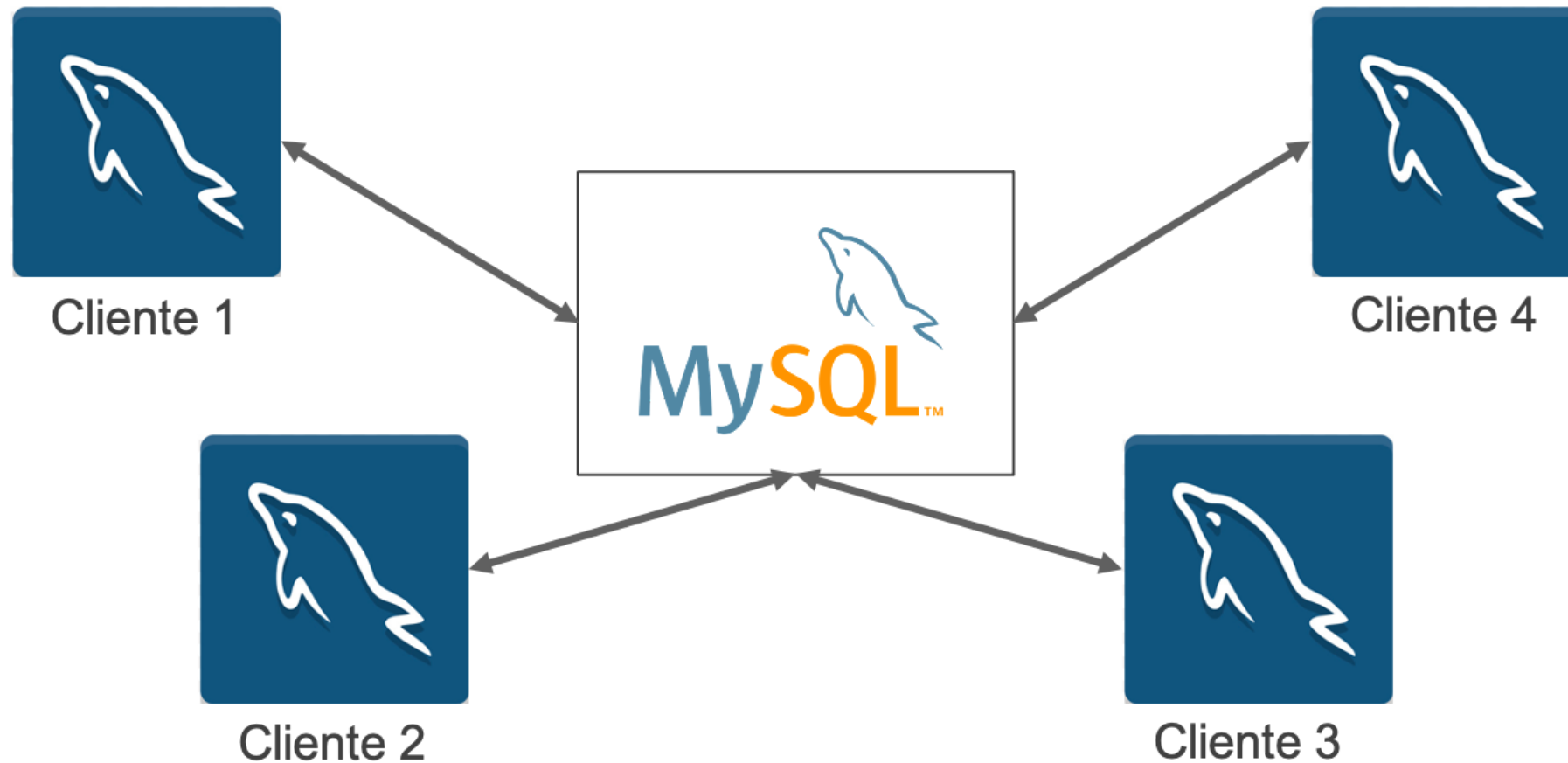
Esquema

Esquema básico de la arquitectura:



Esquema con varios clientes

Lo habitual es que un mismo servidor reciba conexiones de diferentes clientes:



Cientes de MySQL

- El cliente no tiene por qué ser **MySQL Workbench**.
- El cliente puede ser cualquier software que implemente el protocolo **MySQL Client/Server Protocol**.
- La mayoría de lenguajes de programación incorporan librerías (extensiones) para comunicarse con *MySQL* a través de clases y funciones de alto nivel.
- La base de datos es común para todos los programas. Cada programa se comunica con la base de datos a través de su conector:

Drivers nativos

Open Database Connectivity

Open DataBase Connectivity (ODBC) es un estándar de acceso a las bases de datos.

El objetivo de ODBC es permitir el acceso a cualquier dato desde cualquier aplicación.

- Se crea una capa intermedia entre la aplicación y el SGBD.
- Esta capa actúa de traductor entre ODBC y el SGBD.
- Permite utilizar diferentes bases de datos sin cambiar la aplicación.



pyodbc

`pyodbc` es un módulo de código abierto de Python que facilita el acceso a las bases de datos **ODBC**. Implementa la especificación *DB API 2.0*, pero incluye además funciones y características adicionales para facilitar el acceso a la información.

La forma más fácil de instalarlo es usar pip:

```
pip install pyodbc
```

Consulta la [documentación de instalación](#) para información adicional.

MySQL Connector/ODBC

Para conectarnos a una base de datos **MySQL** mediante el protocolo ODBC es necesario instalar el driver específico que nos proporciona el fabricante.

Este driver será dependiente del sistema operativo desde el cual nos queramos conectar al sistema gestor de bases de datos y, por tanto, su [instalación](#) es diferente para cada SO.

Aunque usemos una librería específica de Python para conectarnos (**pyodbc**), esta se basa en el conector oficial de **MySQL** para funcionar.

pyodbc: cadena de conexión

Las conexiones a las bases de datos suelen apoyarse en la **cadena de conexión** para especificar los parámetros.

En el caso de ODBC, una cadena de conexión incluye información como:

- Versión del driver que se va a usar
- Host y puerto de comunicaciones
- Usuario y contraseña
- Codificación y otras opciones de la base de datos

Conector oficial de MySQL para Python

En lugar de usar el conector ODBC, bastante complejo de instalar y configurar, vamos a usar una **librería propia** de MySQL para conectarnos al SGBD desde Python.

Podemos instalar el conector directamente desde `pip`:

```
pip install mysql-connector-python
```

La principal ventaja de este conector es que es autocontenido, no requiere descargarse el conector oficial ni librerías adicionales de Python.

Demo `mysql.connector`

```
import mysql.connector  
from datetime import date, datetime, timedelta
```

Demo `mysql.connector`

```
# Nos conectamos a la base de datos, creando un objeto para gestionar la conexión
mydb = mysql.connector.connect(
    host="localhost",
    user="root",
    password="root"
)

# Las operaciones se ejecutan con un objeto de tipo cursor:
cursor = mydb.cursor()
```


Demo `mysql.connector`

```
# Vamos a crear un esquema de prueba  
cursor.execute("CREATE DATABASE prueba DEFAULT CHARACTER SET 'utf8'")
```

Demo `mysql.connector`

```
# Podemos usar strings multilínea de Python para las consultas SQL
tabla_empleado = """
CREATE TABLE prueba.employees (
    emp_no      INT          NOT NULL AUTO_INCREMENT,
    birth_date   DATE         NOT NULL,
    first_name   VARCHAR(14)  NOT NULL,
    last_name    VARCHAR(16)  NOT NULL,
    gender       ENUM ('M', 'F') NOT NULL,
    hire_date    DATE         NOT NULL,
    PRIMARY KEY (emp_no)
);
"""

cursor.execute(tabla_empleado)
```

Demo `mysql.connector`

```
# Podemos definirnos una función para crear un empleado y almacenarlo en la base de datos
def crear_empleado(fecha_nac, nombre, apellidos, genero, fecha_alta):
    # Vamos a utilizar las consultas parametrizadas
    add_employee = ("INSERT INTO prueba.employees "
                    "(first_name, last_name, hire_date, gender, birth_date) "
                    "VALUES (%s, %s, %s, %s, %s)")
    cursor.execute(add_employee, (nombre, apellidos, fecha_alta, genero, fecha_nac))
    # Nos aseguramos de confirmar la transacción
    mydb.commit()
```

Demo `mysql.connector`

```
# Vamos a solicitar los datos del empleado por pantalla
fecha_nac = datetime.strptime(input("Fecha de nacimiento dd/mm/aa: "), r"%d/%m/%y")
nombre = input("Nombre: ")
apellidos = input("Apellidos: ")
genero = input("Género M/F: ")
fecha_alta = datetime.strptime(input("Fecha de alta dd/mm/aa: "), r"%d/%m/%y")

# Añadimos al usuario
crear_empleado(fecha_nac, nombre, apellidos, genero, fecha_alta)
```

Demo `mysql.connector`

```
# Vamos a realizar una consulta para obtener todos los empleados de la tabla
consulta = """
SELECT emp_no, first_name, last_name, hire_date
FROM prueba.employees
"""

cursor.execute(consulta)

for (id_emp, nombre, apellidos, fecha_alta) in cursor:
    print(f"Empleado {id_emp} - {apellidos}, {nombre} dado de alta el {fecha_alta}")
```

Demo `mysql.connector`

```
# Limpiamos la base de datos
cursor.execute("DROP SCHEMA prueba")
cursor.close()
mydb.close()
```

SQLAlchemy

SQLAlchemy

SQLAlchemy es el conjunto de herramientas SQL de Python y un ORM¹ que ofrece a los desarrolladores de aplicaciones toda la potencia y flexibilidad de SQL.

Proporciona un conjunto completo de patrones de persistencia de nivel empresarial bien conocidos, diseñados para un acceso a la base de datos eficiente y de alto rendimiento, adaptados a un lenguaje de dominio sencillo y pitónico.

¹ *Object Relational Mapping*

SQLAlchemy 2.x

Introduce cambios muy profundos en la API y el comportamiento de la biblioteca

- Un único estilo de consultas basado en `select()` y `Session.execute()`
- *Declarative Mapping* modernizado, con soporte de tipado
 - `DeclarativeBase`, `Mapped[T]` y `mapped_column()`
- Manejo de transacciones más seguro con *context managers*
 - `with Session(engine) as session:`
 - Al cerrar el bloque `with` se hace `close()`, ¡¡pero no `commit()`!!
 - Si hay excepción, se ejecuta un `rollback()` y todo se queda como estaba
- Compatibilidad futura garantizada (la rama 1.x ahora es de mantenimiento)

Instalación:

```
pip install pymysql sqlalchemy # o `sqlalchemy>=2.0`, pero no es necesario
```

¿Quién usa SQLAlchemy?

- Yelp!
- Reddit
- DropBox
- The OpenStack Project
- Survey Monkey

Es una de las librerías para trabajar con bases de datos más usada en Python.

Creación del motor y de sesiones

SQLAlchemy se basa en el concepto de **motor** y **sesión**

- El **motor** es el objeto que se encarga de gestionar la conexión con el SGBD

```
from sqlalchemy import create_engine
engine = create_engine("mysql+pymysql://user:password@host:3306/database")
```

- La conexión se configura mediante una **cadena de conexión** con el siguiente formato:

```
[DB_TYPE]+[DB_CONNECTOR]://[USERNAME]:[PASSWORD]@[HOST]:[PORT]/[DB_NAME]
```

- La **sesión** es el objeto que se encarga de gestionar las transacciones con el SGBD

```
from sqlalchemy.orm import Session
with Session(engine) as session:
    # Consultas, inserciones, etc.
```

Ejecutando consultas

Podemos usar SQLAlchemy de forma similar al conector oficial de MySQL que ya hemos visto con respecto a la ejecución de consultas:

```
res = engine.execute("SELECT * FROM tabla")
for row in res.fetchall():
    # Cosís
```

La principal diferencia es que SQLAlchemy abre la conexión, ejecuta la consulta y cierra la conexión de manera automática.

Aunque interesante, este modo de funcionamiento es similar al conector nativo, por lo que vamos a centrarnos en el ORM.

Object Relational Mapping

- Los ORM se encargan de representar las tablas de la base de datos mediante una estructura de clases
- Toda la interacción entre el programa y la base de datos se realiza a través del ORM:
 - Creación de tablas
 - Inserción y modificación de datos
 - Consultas
- El desarrollador no necesita conocer SQL, aunque es recomendable entender su funcionamiento

Definiendo tablas como modelos de SQLAlchemy

Modelos → clases que representan una tabla SQL en nuestra base de datos

- Atributos del modelo → columnas de una tabla.
- Instancias del modelo → filas de una tabla.

Creamos modelos definiendo clases de Python que extienden una clase base²:

```
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass
```

La parte más crítica es la de cómo modelar las relaciones

- Afortunadamente, con un ejemplo queda muy claro cómo hacerlo.

² Lo más habitual es usar un `DeclarativeBase` propio, y por convenio se le llama `Base`. Usar directamente `DeclarativeBase` no es recomendable por la abstracción interna que hace de la base de datos.

Ejemplo: Problema de las series

Vamos a estudiar el funcionamiento de **SQLAlchemy** mediante el siguiente ejemplo:

- Se quiere desarrollar una aplicación para gestionar las visualizaciones de series por parte de los usuarios.
- Un usuario estará definido por su alias y podrá ver todos los capítulos de las series que quiera.
- Un capítulo, que dispondrá de un título y una duración, pertenecerá a una serie.
- Una serie estará caracterizada por su título y género y dispondrá de un número indeterminado de capítulos.

Ejemplo: Diagrama de entidad-relación



Ejemplo: Diagrama relacional



Ejemplo: Modelando los usuarios

```
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import String, DateTime, func

class Usuario(Base):
    __tablename__ = "usuarios"

    id: Mapped[int] = mapped_column(primary_key=True) # ← autoincrement implícito
    alias: Mapped[str] = mapped_column(String(255), unique=True, nullable=False)
    fecha_alta: Mapped[DateTime] = mapped_column(DateTime, server_default=func.now())

    # N:M con capítulos
    capitulos: Mapped[List["Capitulo"]] = relationship(
        secondary="visualiza", back_populates="usuarios"
    )

    def __repr__(self) -> str:
        return f"<Usuario {self.alias}>"
```

Ejemplo: Modelando las series

```
class Serie(Base):
    __tablename__ = "series"

    id: Mapped[int] = mapped_column(primary_key=True)
    titulo: Mapped[str] = mapped_column(String(500), nullable=False)
    genero: Mapped[str] = mapped_column(String(150), nullable=False)
    fecha_alta: Mapped[DateTime] = mapped_column(server_default=func.now())

    # 1:M con capítulos
    capitulos: Mapped[List["Capitulo"]] = relationship(
        back_populates="serie",
        cascade="all, delete-orphan"
    )

    def __repr__(self) -> str:
        return f"<Serie '{self.titulo}'>"
```

Ejemplo: Modelando los capítulos

```
class Capitulo(Base):
    __tablename__ = "capitulos"

    id: Mapped[int] = mapped_column(primary_key=True)
    titulo: Mapped[str] = mapped_column(String(500), nullable=False)
    duracion: Mapped[int] = mapped_column(Integer, nullable=False)
    id_serie: Mapped[int] = mapped_column(ForeignKey("series.id"), index=True)

    # 1:N con serie
    serie: Mapped["Serie"] = relationship(back_populates="capitulos")
    # N:M con usuarios
    usuarios: Mapped[List["Usuario"]] = relationship(
        secondary="visualiza", back_populates="capitulos"
    )

    def __repr__(self) -> str:
        return f"<Capítulo '{self.titulo}' de {self.serie.titulo}>"
```

Ejemplo: Tabla de asociación (N:M)

Para las relaciones N:M es necesario definir una tabla de asociación que contenga las claves foráneas de las tablas involucradas:

```
visualiza = Table(  
    "visualiza",  
    Base.metadata,  
    mapped_column("id_usuario", ForeignKey("usuarios.id"), primary_key=True),  
    mapped_column("id_capitulo", ForeignKey("capitulos.id"), primary_key=True),  
)
```

Esta tabla es la que llevaría atributos adicionales en caso de ser necesarios

Ejemplo: Creación del modelo de datos

Una vez hemos definido nuestros modelos es el momento de crear las tablas correspondientes en la base de datos

- Para ello basta con ejecutar el método `create_all()` de nuestra clase `Base`:

```
Base.metadata.create_all(engine)
```

SQLAlchemy **generará** las **sentencias SQL** correspondientes a la creación de las tablas **y las ejecutará** contra la base de datos

- Se debe usar una sola vez en el *bootstrap* de la aplicación

Ejemplo: Creación de registros

Aprovechando el ORM, vamos a añadir un usuario, una serie con algunos de sus capítulos y a relacionar al usuario con los capítulos que ha visto.

```
with Session(engine) as session:
    u1 = Usuario(alias="Katee Sackhoff")
    battlestar = Serie(titulo="Battlestar Galactica (2004)", genero="Science Fiction")
    p1 = Capitulo(titulo="Part/Night 1", duracion=118, serie=battlestar)
    p2 = Capitulo(titulo="Part/Night 2", duracion=118, serie=battlestar)

    u1.capitulos.append(p1)

    session.add_all([u1, battlestar, p1, p2])
    session.commit()
```

Si todo ha ido bien, la sesión se habrá cerrado con un `commit()` y los cambios se habrán propagado a la base de datos.

- Y si no, se habrá hecho un `rollback()` y no se habrá modificado nada

Ejemplo: Recuperando registros de la BD

No tiene sentido contar la API aquí, para eso está su [documentación oficial](#)

- Capítulos de más de 40 minutos:

```
caps = session.scalars(
    select(Capitulo).where(Capitulo.duracion > 40)
).all() # Scalars devuelve sólo la primera columna (el id)
```

- Número de series por género

```
res = session.execute(
    select(Serie.genero, func.count())
    .group_by(Serie.genero)
).all()
```

Los **resultados** de las consultas son ya **tipos de Python**, no hace falta convertirlos

Ejemplo: Actualización y borrado de registros

Actualizar una instancia implica la propagación de cambios cuando finaliza la transacción

```
with Session(engine) as session:
    cap = session.get(Capitulo, 1)
    cap.duracion += 5 # Editamos
    session.flush()   # Forzamos la propagación

    serie = session.get(Serie, 1)
    session.delete(serie) # Eliminamos la serie (y capítulos huérfanos)
    session.commit()      # Propagamos los cambios (obligatorio)
```

La función `flush()` fuerza la propagación de los cambios a la base de datos, pero no hace `commit()`

Ejemplo: Relaciones 1:1

En ocasiones podemos querer modelar relaciones 1:1 entre tablas.

```
class Perfil(Base):  
    __tablename__ = "perfiles"  
  
    id: Mapped[int] = mapped_column(ForeignKey("usuarios.id"), primary_key=True)  
    bio: Mapped[str] = mapped_column(String(500))  
  
    usuario: Mapped[Usuario] = relationship(back_populates="perfil", uselist=False)  
  
Usuario.perfil = relationship(  
    "Perfil", back_populates="usuario", uselist=False, cascade="all, delete-orphan"  
)
```

La clave está en el atributo `uselist=False`, que indica que esa relación es 1:1.

Licencia

**Esta obra está licenciada bajo una licencia Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 Internacional.**

**Puede encontrar su código en el siguiente enlace:
<https://github.com/bbddetsisi/material-docente>**