Programación contra bases de datos en python

Departamento de Sistemas Informáticos

E.T.S.I. de Sistemas Informáticos

Universidad Politécnica de Madrid



Programación contra bases de datos

Índice

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

Bases de datos 2 / 44

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.

Bases de datos 4 / 44

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:

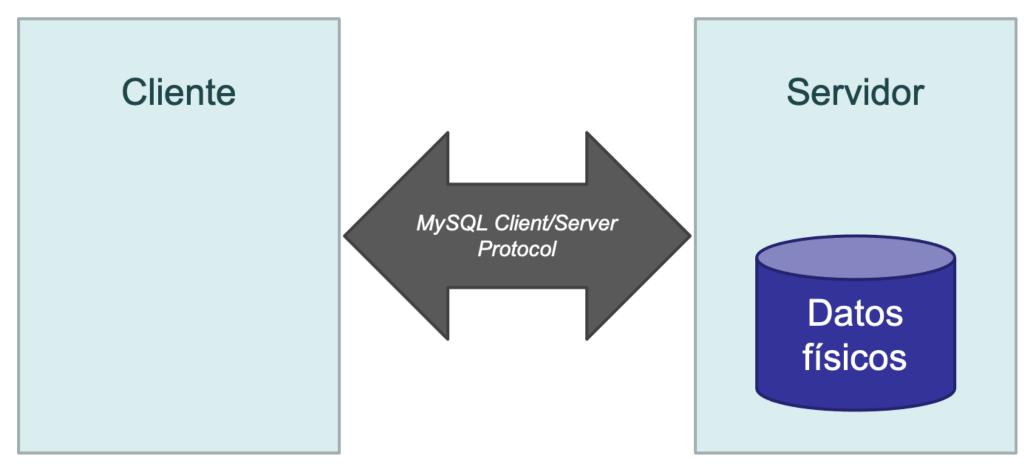
- Se denomina MySQL Client/Server Protocol.
- Se ejecuta sobre TCP.
- El cuerpo de los mensajes incluye sentencias SQL.

Más información en la documentación de MySQL.

Bases de datos 5 / 44

Esquema

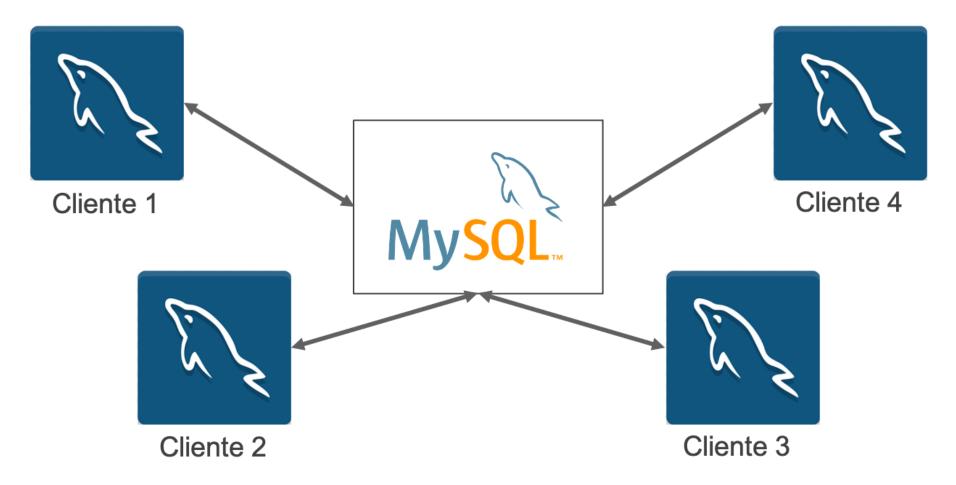
Esquema básico de la arquitectura:



Bases de datos 6 / 44

Esquema con varios clientes

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



Bases de datos 7 / 44

Clientes 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:

Bases de datos 8 / 44

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.



Bases de datos 10 / 44

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 innformación adicional.

Bases de datos 11 / 44

MySQL Connector/ODBC

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

Este driver será dependiende 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.

Bases de datos 12 / 44

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

Bases de datos 13 / 44

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.

Bases de datos 14 / 44

Programación contra bases de datos

Demo mysql.connector

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

Bases de datos 15 / 44

```
# 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()
```

Bases de datos 16 / 44

Programación contra bases de datos

Demo mysql.connector

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

Bases de datos 17 / 44

```
# Podemos usar strings multilínea de Python para las consultas SQL
tabla_empleado =
CREATE TABLE prueba.employees (
            INT
                               NOT NULL AUTO_INCREMENT,
   emp_no
   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)
11 11 11
cursor.execute(tabla_empleado)
```

Bases de datos 18 / 44

Bases de datos 19 / 44

```
# 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)
```

Bases de datos 20 / 44

```
# 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}")
```

Bases de datos 21 / 44

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

Bases de datos 22 / 44

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.

Para instalar:

pip install sqlalchemy pymysql

¹: Object Relational Mapper

Bases de datos 24 / 44

¿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.

Bases de datos 25 / 44

Conexión con el SGBD

Las conexiones con el SGBD se gestionan mediante un **motor (engine)**, que no es más que un objeto que representa una base de datos.

```
"""Conexión con la BBDD."""
from sqlalchemy import create_engine
engine = create_engine(
    "mysql+pymysql://user:password@host:3600/database",
)
```

Como se puede observar, la conexión se configura mediante una cadena de conexión con el siguiente formato:

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

Bases de datos 26 / 44

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():
...
```

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.

Bases de datos 27 / 44

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

Bases de datos 28 / 44

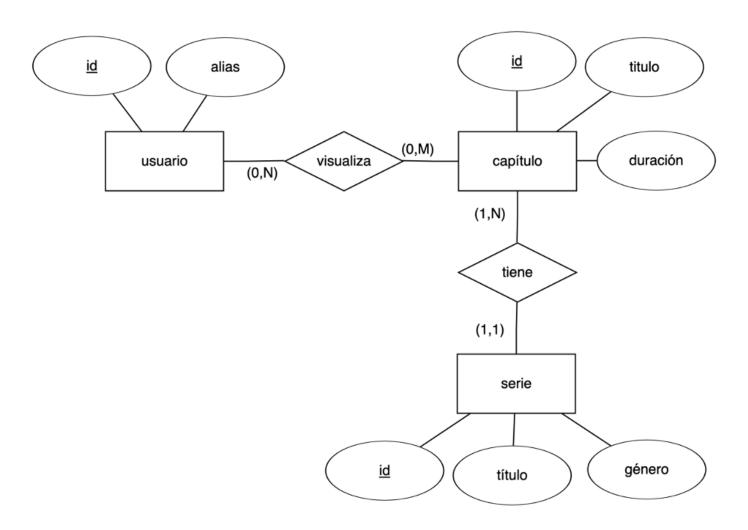
SQLAlchemy: ejemplo de uso

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.

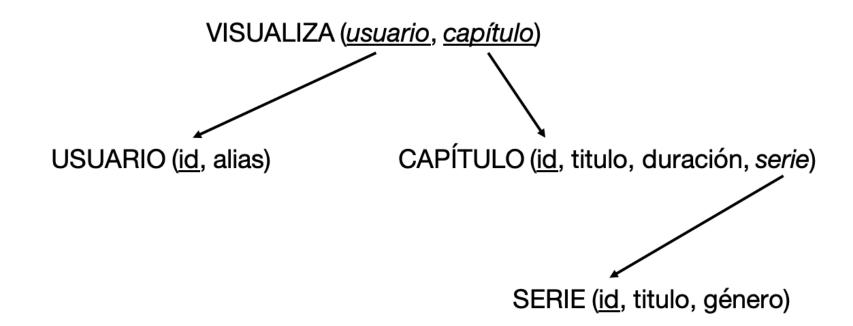
Bases de datos 29 / 44

SQLAlchemy: ejemplo de uso



Bases de datos 30 / 44

SQLAlchemy: ejemplo de uso



Bases de datos 31 / 44

Definiendo las tablas como modelos de SQLAlchemy

Los modelos de datos son clases de Python que representan una tabla SQL en nuestra base de datos, donde los atributos de un modelo se traducen en columnas de una tabla.

Al trabajar con ORMs, la creación de instancias de nuestros modelos se traduce en la creación de filas en una tabla SQL.

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

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

Nuestros modelos deben extender la clase Base.

Bases de datos 32 / 44

Modelando los usuarios

Empezamos nuestros modelos con los *Usuarios*:

```
from sqlalchemy import Column, Table, ForeignKey
from sqlalchemy.types import Integer, String, DateTime
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
class Usuario(Base):
  # Nombre de la tabla que se creará en la BD
   __tablename__ = "usuarios"
  # Atributos del modelo (columnas de la tabla)
   id = Column(Integer, primary_key=True, autoincrement="auto")
   alias = Column(String(255), unique=True, nullable=False)
   fecha_alta = Column(DateTime, server_default=func.now())
   def __repr__(self):
      return f"<Usuario: {self.alias}>"
```

Bases de datos 33 / 44

Modelando las series

El modelo para las *Series* se construye de manera similar a los usuarios:

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

id = Column(Integer, primary_key=True, autoincrement="auto")
    titulo = Column(String(500), nullable=False)
    genero = Column(String(150), nullable=False)
    fecha_alta = Column(DateTime, server_default=func.now())

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

Bases de datos 34 / 44

Relaciones 1:N

Para los *capítulos* necesitamos establecer una relación 1:N con la serie a la que pertenecen. Definimos la clave externa en la columna:

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

id = Column(Integer, primary_key=True, autoincrement="auto")
titulo = Column(String(500), nullable=False)
duracion = Column(Integer, nullable=False)
id_serie = Column(Integer, ForeignKey("series.id")) # Referencia a nombre de tabla
serie = relationship("Serie", backref="capitulos") # Referencia a nombre de la clase

def __repr__(self):
    return f"<Capitulo '{self.titulo}' ({self.serie})>"
```

Bases de datos 35 / 44

Relaciones N:M

Para las relaciones N:M se define una **tabla de asociación** en SQLAlchemy:

```
tabla_asoc = Table('visualiza', Base.metadata,
    Column('id_usuario', ForeignKey('usuarios.id'), primary_key=True),
    Column('id_capitulo', ForeignKey('capitulos.id'), primary_key=True)
)
```

Ahora solo queda añadir las relaciones a los modelos **Usuario**:

```
capitulos = relationship("Capitulo", secondary=tabla_asoc, backref="usuarios")
```

Bases de datos 36 / 44

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)

Será SLQAlchemy quien generará las sentencias SQL correspondientes para crear en la base de datos los modelos definidos.

Bases de datos 37 / 44

Creación de registros

Vamos a añadir un usuario a nuestra base de datos, pero en lugar de usar una sentencia INSERT vamos a aprovechar el ORM de SQLAlchemy. Lo primero es crearnos una sesión:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
```

A partir de ahora tenemos una conexión abierta con la BD que se corresponde con una transacción abierta, por lo que tendremos que confirmar los cambios commit() de manera manual.

Bases de datos 38 / 44

Creación de registros

Vamos a crear una nueva instancia de un usuario utilizando nuestro modelo Usuario para ello:

```
u1 = Usuario(
alias="Carlos Boyero"
)
```

Para que nuestro usuario se propague a la BD bastará con añadir la instancia a nuestra sesión:

```
session.add(u1)
session.commit()
```

Bases de datos 39 / 44

Registros relacionados: serie y capítulos

Creamos una nueva serie:

```
perdidos = Serie(titulo="Lost", genero="Ciencia Ficción")
session.add(perdidos)
session.commit()
```

Vamos a añadir dos capítulos de la serie que acabamos de crear:

```
s01e01 = Capitulo(titulo="Pilot, Part 1", duracion=42, id_serie=perdidos.id)
s01e02 = Capitulo(titulo="Pilot, Part 2", duracion=41, id_serie=perdidos.id)
session.add(s01e01)
session.add(s01e02)
session.commit()
```

Bases de datos 40 / 44

Visalizaciones del usuario

Recordemos que nuestro modelo Usuario disponía de un atributo capitulos vinculado a la tabla que modela nuestra relación N:M.

Para decir que un usuario ha visto un capítulo en concreto, basta con añadir la instancia del capítulo a este atributo:

```
u1.capitulos.append(s01e01)
session.commit()
```

Bases de datos 41 / 44

Recuperando registros de la BD

Para realizar consultas a la BD y obtener el objeto correspondiente a nuestro modelo usaremos la función query() de la sesión abierta.

Por ejemplo, podemos obtener todos los capítulos de la base de datos:

```
caps = session
  .query(Capitulo) # FROM Capitulos
  .all() # SELECT *
```

La variable caps contendrá una colección de instancias de capítulos:

```
for c in caps:
  print(c)
```

Bases de datos 42 / 44

API de consultas

La API de SLQAlchemy para realizar consultas es bastante extensa y permite realizar búsquedas más complejas. Por ejemplo, podemos aplicar filtros sobre valores de columnas:

```
res = session
.query(Capitulos)
.filter_by(duracion >= 25)
.all()
```

e incluso aplicar funciones de agregación y agrupamiento:

```
res = session
  .query(func.Count(Serie.id))
  .group_by(Serie.genero)
  .all()
```

Bases de datos 43 / 4

Actualización y eliminación de registros

La actualización de una instancia implica que los cambios se propagan a la base de datos cuando finalice la transacción. Podemos forzar la propagación de cambios con:

```
session.flush()
```

Para eliminar un registro bastará con pasar la instancia del elemento que se quiere eliminar al método delete() de la sesión que tengamos abierta:

```
session.delete(s01e02)
session.commit()
```

Bases de datos 44 / 44