

SQLAlchemy convierte bases de datos en objetos Python

# LA PIEDRA FILOSOFAL

La relación entre los lenguajes orientados a objetos y las bases de datos es bastante tensa, pero SQLAlchemy nos ayuda a sacar todo el partido de una base de datos sin dejar nuestro lenguaje favorito.

POR JOSÉ MARÍA RUÍZ

Todos usamos bases de datos, pero ¿debemos aprender todos SQL? Hay quienes defienden la base de datos como el sistema sobre el que debe girar el software y después están aquellos que creen que un buen diseño orientado a objetos es superior al de una base de datos. Incluso existe un término para definir los problemas que aparecen al intentar almacenar datos de objetos en una base de datos relacional: «impedancia objeto/relacional».

Este problema, que aparece continuamente cuando se crean programas, ha sido abordado desde muchos puntos de vista. SQLAlchemy,

(ver Recurso [1]), nació con el objetivo de crear una capa en Python que nos permita olvidarnos prácticamente del SQL. SQLAlchemy aún no ha alcanzado la versión 1.0, pero es un proyecto maduro y muy bien documentado (ver Recurso [2]).

SQLAlchemy nos permite trabajar con complejas bases de datos de forma simple o crear bases de datos a partir de nuestro código de forma automática. Python es un lenguaje especialmente útil para los administradores de sistemas, una de cuyas tareas es la gestión de los ficheros de log de los sistemas. Vamos a crear un pequeño programa que extraiga las

líneas de un fichero de log de Apache y las almacene en una base de datos para que podamos realizar consultas complejas.

## SQLAlchemy

SQLAlchemy es, según su propia página web, mucho más que un simple ORM («Object Relatio-

nal Mapper»). Nos permite olvidarnos casi completamente de la base de datos, aunque podemos hacer uso de las características especiales de la misma si las necesitáramos. De hecho, SQLAlchemy posee dos niveles: *ORM* y *SQL Expression*. El nivel *ORM* trabaja a nivel de clases y objetos, es ideal para despreocuparse completamente de la existencia de una base de datos; *SQL Expression*, en cambio, nos permite trabajar a un nivel más bajo, manipulando tablas y usando un lenguaje muy cercano a *SQL*. Esta es una de las grandes virtudes de SQLAlchemy: podemos comenzar trabajando a alto nivel con *ORM* y para consultas delicadas podemos emplear *SQL Expression*.

Una de las enormes ventajas que supone emplear SQLAlchemy es que aísla completamente nuestro código de la base de datos que empleemos. Con SQLAlchemy es posible comenzar a desarrollar una aplicación empleando una base de datos como *SQLite*, que no requiere instalación, y pasar después a trabajar con algo más grande, como por ejemplo *MySQL*, *Firebird* o *PostgreSQL*. Para conectar con una base de datos *SQLite* debemos emplear:

```
>>> from sqlalchemy import
create_engine
>>> engine = create_engine(
('sqlite:///memory:',
echo=True)
```

Para conectar con la base de datos necesitamos crear un *engine* («motor» en inglés). La función *create\_engine()* actúa como factoría, creando el objeto adecuado dependiendo de la cadena de conexión que proporcionemos como primer parámetro. En el primer caso conectamos con una base de datos virtual en memoria de *SQLite*. Esta es una opción muy buena para realizar pruebas, puesto que no llegamos a crear ningún fichero y por tanto es básicamente una base de datos de usar y tirar. Un ejemplo de conexión más complejo sería:

```
>>> engine = create_engine(
('postgres://josemaria:
contraseña@192.168.1.134:5323/
pedidos', echo=True)
```

En él conectamos con una base de datos *PostgreSQL*, llamada *usuarios*, que se encuentra en el ordenador con dirección de red *192.168.1.134* a través del puerto *5323* con el usuario *josemaria* y la clave *contraseña*. Si ejecutamos *create\_engine()* con el parámetro *echo=True* SQLAlchemy imprimirá por la salida estándar, generalmente el terminal de

```
Terminal - Konsole
Sesión Editar Vista Marcadores Preferencias Ayuda
./analiza.py
2009-04-29 18:36:38.898 INFO sqlalchemy.engine.base.Engine.0x...588c PRAGMA table_info('logs')
2009-04-29 18:36:38.899 INFO sqlalchemy.engine.base.Engine.0x...588c ()
Fecha      Visitas
-----
2009-04-29 18:36:38.904 INFO sqlalchemy.engine.base.Engine.0x...588c BEGIN
2009-04-29 18:36:38.905 INFO sqlalchemy.engine.base.Engine.0x...588c SELECT logs.fecha AS logs.fecha, count(*) AS count
_1
FROM logs GROUP BY logs.fecha ORDER BY logs.fecha
2009-04-29 18:36:38.905 INFO sqlalchemy.engine.base.Engine.0x...588c [**]
2009-03-02      158
2009-03-07      298
2009-03-16       2
2009-03-17      14
>
```

Figura 1: Ejemplo de salida con *echo=True*.

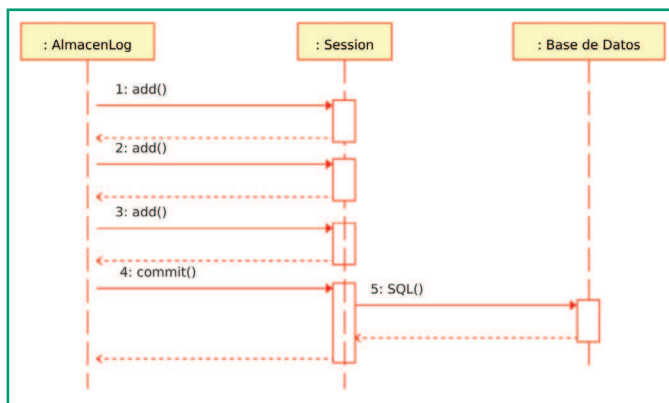


Figura 2: Esquema de trabajo con Session().

comandos, las sentencias SQL que ejecute. De esta forma podemos hacer pruebas y comprobar qué está haciendo exactamente SQLAlchemy con nuestra base de datos, ver Figura 1.

## La Clase LogApache

Apache nos permite seleccionar qué datos queremos que aparezcan en sus ficheros de log, pero la mayoría de las instalaciones emplean un formato estándar bien documentado. Para definir el objeto emplearemos el método más sencillo de SQLAlchemy: el estilo declarativo (ver Listado 1). Para ello

Lo primero que hacemos es establecer el nombre de la tabla que almacenará los datos. Para ello asignamos el nombre a la variable `__tablename__`. Posteriormente tenemos que definir las variables de la clase. Cada una de ellas contendrá un objeto de la clase `Column` que nos permite definir el nombre (que puede ser distinto al de la clase), el tipo y las propiedades de las columnas que poseerá la tabla `logs` en la base de datos. Recordemos que Python posee tipos, pero no podemos definir el tipo de una variable antes de usarla. Además, los tipos que emplean las bases de datos son diferentes a los que posee Python.

necesitamos crear una clase de la que heredarán nuestras propias clases mediante la función `declarative_base()`.

Almacenamos esta clase en una variable y heredamos de ella en todas las clases que representen objetos a almacenar.

En nuestro caso sólo crearemos una clase: `LogApache`.

SQLAlchemy nos permite emplear tipos genéricos de bases de datos (aquellos definidos por los estándares) o tipos específicos de alguna base de datos. Es mejor no hacer esto último a no ser que tengamos muy buenas razones (por ejemplo, que tengamos que emplear una base de datos predefinida que emplee tipos especiales). Las propiedades nos permiten establecer qué valores identifican a un objeto de la clase `LogApache` de forma única mediante `primary_key`. No pueden existir dos objetos con el mismo identificador. Como el fichero de log de Apache no nos proporciona un identificador único para cada línea, SQLAlchemy se hará cargo de asignar a la variable `id` un identificador único.

Podremos usar todas estas variables de forma normal y corriente, como podemos ver en el constructor `__init__` de `LogApache` en el Listado 1. SQLAlchemy trabaja de forma transparente. Cada vez que modifiquemos el valor de una de estas variables, SQLAlchemy tomará nota de ello. Por último, cuando tengamos definidas todas las clases que vayamos a almacenar, sólo tenemos que invocar a `metadata.create_all(engine)` y SQLAlchemy generará las tablas para nosotros. Si las tablas ya existieran, SQLAlchemy no haría nada,

## Listado 1: Constructores Estáticos (parte 1)

```

01 #!/usr/local/bin/python2.6
02 # -*- coding: utf-8 -*-
03
04 from datetime import datetime
05 import re
06 import urlparse
07 from sqlalchemy import *
08 from sqlalchemy.orm import *
09 from sqlalchemy.sql import *
10 from
11 sqlalchemy.ext.declarative
12 import *
13
14 engine =
15 create_engine("sqlite:///logs.
16 db", echo = False)
17
18 Session =
19 sessionmaker(bind=engine)
20
21 Base = declarative_base()
22 metadata = Base.metadata
23
24 class LogApache(Base):
25     __tablename__ = 'logs'
26
27     id = Column('id',
28 Integer, primary_key=True)
29
30 Ip1 = Column('ip1',
31 String, nullable = False)
32
33 Fecha = Column('fecha',
34 Date, nullable = False)
35
36 Hora = Column('hora',
37 Time, nullable = False)
38
39 Peticion = Column('peticion',
40 String, nullable = False)
41
42 Status =
43 Column('codigohttp', Integer,
44 nullable = False)
45
46 Tcpu = Column('tcpu',
47 Integer, nullable = False)
48
49 Referer = Column('referer',
50 String)
51
52 Ip2 = Column('ip2',
53 String)
54
55 Usuario = Column('usuario',
56 String)
57
58 Agente = Column('agente',
59 String)
60
61 self.Ip1 = ip1
62 self.Fecha = fecha
63 self.Hora = hora
64 self.Peticion = petition
65 self.Status = status
66 self.Tcpu = tcpu
67
68 self.Referer = referer if
69 referer != "" else None
70
71 self.Ip2 = ip2 if ip2 !=
72 "" else None
73
74 self.Usuario = usuario if
75 usuario != "" else None
76
77 self.Agente = agente if
78 agente != "" else None
79
80 metadata.create_all(engine)
81
82 class AlmacenLogs(object):
83
84     def __init__(self):
85         self.sesion = Session()
86         self.ReLinea =
87 re.compile("(?P<ip1>[\\d.]+)
88 (\\S+) (\\S+)
89 \\[(?P<fecha>[\\w:/]+\\s+[\\-]

```

por lo que es seguro dejar esta sentencia en nuestro programa.

### Sesiones y Metadatos

Para poder trabajar con una base de datos necesitamos conectar con ella empleando la clase *Session*. Esta clase actúa como intermediario entre la base de datos y nuestro código fuente, encargándose de crear los objetos que extraigamos de la base de datos o de generar el SQL a partir de ellos. *Session* almacena todas las operaciones que hayamos realizado y sólo las ejecuta cuando se lo indiquemos.

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

Siguiendo un patrón que ya hemos visto con *declarative\_base()*, debemos crear una clase *Session* a través de la factoría *sessionmaker()*, pasándole como parámetro el *engine* que estamos utilizando. Como resultado obtendremos la clase *Session* que nos permitirá crear sesiones de trabajo. No sólo eso, sino que además este objeto se encargará de reutilizar conexiones de forma que no

tengamos que estar abriendo y cerrando conexiones con la base de datos. Solo debería existir un objeto *Session* en nuestro programa, por lo que lo ideal es crearlo al arrancar el programa y posteriormente crear tantas sesiones como necesitemos a partir de él.

### Guardando Objetos LogApache

La clase *AlmacenLogs* es la encargada de realizar el trabajo duro en este programa. Su método *AddFichero()* acepta como parámetro la ruta de un fichero que contenga logs de Apache. Una vez abierto el fichero se itera sobre cada una de sus líneas, despedazándolas con la expresión regular almacenada en *self.ReLinea*. Esta expresión regular emplea una característica bastante interesante, a cada grupo (conjunto de letras reconocido por un patrón) que nos interese podemos asignarle un nombre mediante la sintaxis:

```
(?<NOMBRE>
EXPRESION_REGULAR)
```

De esta forma nos resultará mucho más sencillo identificar las distintas cadenas de texto reconocidas y podremos entender mejor la expresión regular. Pasamos cada una de las líneas del fichero al método *AddLinea()* que aplica sobre ella la expresión regular. Si la expresión regular reconoce la línea, devolverá un objeto *Match*, en caso contrario devolverá *None*.

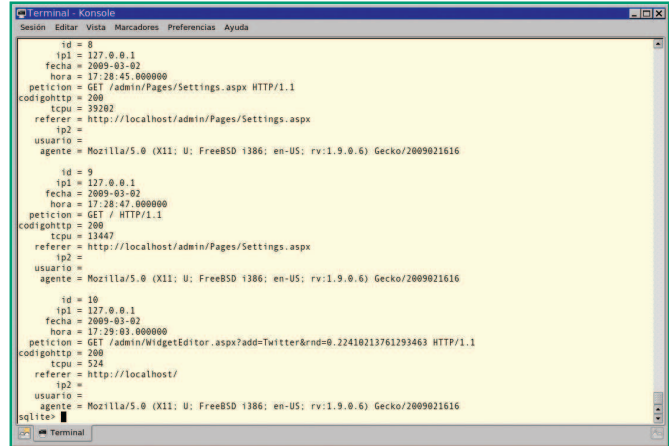


Figura 3: Contenido de nuestra base de datos.

### Listado 1: Constructores Estáticos (parte 2)

<pre> \d{4})\] \"(?P&lt;petition&gt;.+?)\" (?P&lt;status&gt;\d{3}) (?P&lt;cpu&gt;\d+) \"(?P&lt;referer&gt;[^\"]+)\" \"(?P&lt;agente&gt;[^\"]+)\" 55 56 def AddFichero(self, fichero): 57 for linea in open( fichero ): 58 self.AddLinea(linea) 59 self.sesion.commit() 60 61 def AddLinea (self,linea): 62 63 match = self.ReLinea.match(linea) 64 65 datos = match.groupdict() if match else {} 66 67 if datos: 68 fechaYHora = datetime.strptime(datos['fecha '].split(" ")[0], 69</pre>	<pre> "%d/%b/%Y:%H:%M:%S") 70 71 lineaLog = LogApache(ip1 = datos['ip1'], 72 73 fechaYHora, 74 75 fechaYHora, 76 77 datos['petition'], 78 79 datos['referer'], 80 81 datos['status'], 82 83 datos['agente'], 84 85 datos['cpu']) self.sesion.add(lineaLog) 81 82 def 83 ImprimeEstadisticas(self): 84 consulta = self.sesion.query(LogApache.Fe</pre>	<pre> cha, func.count('*') \ 85 .group_by(LogApache.Fecha) \ 86 .order_by(LogApache.Fecha) 87 88 print "Fecha Visitas" 89 print "===== 90 for resultado in consulta: 91 print resultado[0], ' ', resultado[1] 92 93 if __name__ == "__main__": 94 almacenLogs = AlmacenLogs() 95 almacenLogs.AddFichero("ficher o.log") 96 almacenLogs.ImprimeEstadistica s()</pre>
--	---	---

El objeto *Match* posee muchos métodos, pero el que nos interesa es *groupdict()*, que nos da un diccionario con las cadenas reconocidas y que emplea como índices los nombres que les dimos. Este método es mucho más seguro que utilizar las posiciones, sobre todo en expresiones regulares complejas. Sólo nos falta convertir al tipo *datetime*, la fecha y hora que aparecen en el log de Apache, puesto que hemos definido el tipo de ambas variables como *Date* y *Time*.

Ya sólo nos queda crear un objeto de tipo *LogApache* y pasárselo a *self.sesion.add()*. Este método añade un objeto a la base de datos, así de simple, ¡No hay que hacer nada más!. Cuando finalizamos el bucle en *AddFichero()*, ejecutamos *self.sesion.commit()* para asegurarnos de que todas las operaciones sobre la base de datos se llevan a cabo. Ejecutando el comando *sqlite3* desde el shell:

```
% sqlite3 logs.db
```

podemos comprobar la nueva base de datos, (ver Figura 3). De esta forma tan simple hemos almacenado el contenido del fichero en una base de datos que nos permitirá realizar consultas muy complejas sobre los datos de forma eficiente.

## Consultas

Nuestra base de datos está lista para trabajar. En los pasos anteriores hemos creado las tablas necesarias en la base de datos y hemos almacenado en ellas todos los objetos *LogApache* establecidos. Ahora vamos a realizar consultas sobre esos objetos. La clase *AlmacenLogs* posee un método que nos muestra cómo realizar una consulta sobre los datos, es *ImprimeEstadísticas()*. Este método recopila todas las líneas de la base de datos por fecha y las cuenta, imprimiendo por pantalla el resultado:

Fecha	Visitas
2009-03-02	225
2009-03-07	447
2009-03-16	3
2009-03-17	21

El sistema de consultas de SQLAlchemy es extremadamente potente. Nos ofrece mucha más flexibilidad que la que proporcionaría SQL, además de hacer el trabajo sucio. Por ejemplo, podemos realizar una consulta como la siguiente:

```
self.sesion.query(
    (LogApache.Fecha,
     LogApache).all()
```

Nos devolvería tuplas con cada una de las fechas de los objetos *LogApache* ¡Así como el propio objeto *LogApache*!. No existe nada igual en SQL, que sólo trabaja con filas y columnas. Pero nosotros no queremos todos los objetos *LogApache*, lo que queremos saber es cuántas visitas ha habido por día. La fecha de cada visita aparece reflejada en *LogApache.Fecha*. Pero ¿qué hay de la cantidad de visitas? Si revisamos la consulta:

```
self.sesion.query(
    (LogApache.Fecha,
     func.count('*').label(
        ('visitas')) \
        .group_by(LogApache.Fecha)\
        .order_by(LogApache.Fecha)
```

observaremos algo extraño en los parámetros de *self.sesion.query()*, y es que empleamos el método *func.count('\*')*. Las bases de datos cuentan con una serie de funciones denominadas *aggregate* que nos permiten realizar operaciones sobre los resultados, *func* contiene esas operaciones espaciales (entre las que aparecen *count()*, contar, y *sum()*, sumar). Si ejecutásemos sólo el *self.sesion.query()*, obtendríamos una lista con todas las fechas y un *1* al lado de cada una. Para obtener la cantidad de *LogApache.Fecha* diferentes debemos agrupar el resultado en base a algún parámetro, que en nuestro caso no es otro sino *LogApache.Fecha*. Por último ordenamos el resultado en base, de nuevo, a *LogApache.Fecha*.

## Alterando los Datos

Ya hemos realizado una consulta alto compleja, ahora necesitamos modificar los datos. Con SQLAlchemy es realmente sencillo, cada vez que modificamos un objeto *LogApache*, cambiando el valor de algunas de sus variables, SQLAlchemy toma nota de ello en la sesión. La sesión se encarga de controlar el estado de los objetos e intercepta cualquier intento de modificarlos. Cuando así sucede, la sesión marca el objeto como «sucio» y listo para ser actualizado en el siguiente *commit()*. Veamos cómo funciona este proceso con un ejemplo. Imaginemos que queremos ajustar la hora de algunos logs porque Apache los registró mal. Para ello deberemos recuperarlos con un *query()* y cambiarles la hora uno a uno:

```
logErroneo.Hora =
datetime.now()
```

En ese preciso instante ese objeto será marcado como «sucio», y la sesión que nos lo dio, a través de *sesion.query()*, lo incluirá en una lista de actualizaciones. Podemos ver la lista con *sesion.dirty*:

```
>>> sesion.dirty
IdentitySet
([<LogApache(...)>])
```

Cada objeto modificado aparecerá en *sesion.dirty*, pero SQLAlchemy no hará nada hasta que se lo indiquemos o le forcemos a ello. Podemos forzarle a ejecutar los cambios si intentamos realizar una consulta, un *query()*. Si no los efectuase obtendríamos datos erróneos. Si queremos asegurarnos de que los cambios se guarden, siempre podemos ejecutar *sesion.commit()*.

¿Qué pasa cuando queremos borrar un objeto? Digamos que queremos eliminar todos los objetos cuya *Ip1* corresponda con la de un ordenador que empleamos para hacer pruebas. Para ello deberíamos realizar una consulta como la siguiente:

```
>>> for log in sesion.query(
    (LogApache).filter_by(Ip1 =
    '192.168.1.1'):
    ... sesion.delete(log)
>>>
```

Si volviésemos a ejecutar *sesion.dirty* comprobaríamos que esos objetos han sido marcados para ser borrados y, de nuevo, los cambios sólo se ejecutarán cuando forcemos a SQLAlchemy a ello o cuando se lo indiquemos con *sesion.commit()*.

## Conclusión

SQLAlchemy permite hacer uso de una base de datos sin saber nada de SQL, aunque el que sepa SQL puede sacar todo el partido a su base de datos sin dejar Python. El diseño de SQLAlchemy nos permite obviar la tarea de interactuar con la base de datos, permitiéndonos llevar a cabo tareas muy complejas como realizar transacciones que involucren diferentes bases de datos a la vez. ■

## RECURSOS

- [1] SQLAlchemy: <http://www.sqlalchemy.org>
- [2] Documentación: <http://www.sqlalchemy.org/docs/05/index.html>