Introducción a SQLAlchemy[1]

Mapeo Objeto-Relacional

El ORM de *SQLAlquemy* presenta un método para asociar las clases de Python definidas por el usuario con las tablas de la base de datos, y las instancias de esas clases (es decir, los objetos) con las filas en su tabla correspondiente.

Además, el ORM incluye un sistema que sincroniza de forma transparente para el usuario todos los cambios de estado entre los objetos y sus correspondientes filas, llamado *unit of work*, así como un sistema para expresar consultas a la base de datos en términos de las clases definidas por el usuario y las relaciones entre cada una.
Declarar un mapeo

Cuando se utiliza el ORM, el proceso de configuración comienza por describir las tablas de la base de datos que vamos a utilizar y luego definiendo nuestras propias clases que serán mapeadas a esas tablas.

Sin embargo, en las nuevas versiones de *SQLAlchemy* estas dos tareas se realizan generalmente juntas, utilizando un sistema conocido como ***declarativo***, que nos permite crear clases que incluyen directivas para describir la tabla actual de la base de datos que será mapeada.

Las clases mapeadas utilizando el *sistema declarativo* son definidas en términos de una clase base que mantiene un catálogo de clases y tablas relativas a esta base, esto es conocido como la clase base declarativa.

[1] https://docs.sqlalchemy.org

Nuestra aplicación generalmente tendrá solo una instancia de esta base declarativa en un módulo común importado.

Para crear la clase base declarativa utilizamos la función *declarative_base().*

Para crear un mapeo de modelo, tenemos que crear una clase que herede de la base creada con la función *declarative_base* y como mínimo tenemos que definir el nombre de la tabla y al menos una columna que sea primary key.

En el siguiente ejemplo se crea la clase **Author** que tendrá las columnas *id* de tipo entero, *firstname* de tipo String y *lastname* de tipo String.

Además, la columna *id* es clave primaria y tiene definida una secuencia. La secuencia hará que dicho campo no tenga que ser definido por el usuario, sino que la base de datos al insertar un registro lo autocomplete con un valor secuencial.

```
# -*- coding: utf-8 -*-

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Sequence

Base = declarative_base()



class Author(Base):
    __tablename__ = 'author'

    id = Column(Integer, Sequence('author_id_seq'), primary_key=True)
    firstname = Column(String)
    lastname = Column(String)
```

```
def __repr__(self):
    return "{} {}".format(self.firstname, self.lastname)
```

Cuando nuestra clase es construida, el sistema declarativo reemplaza todos los objetos de tipo **Column** por un atributo especial conocido como *descriptores*[2]; este es un proceso conocido como *instrumentación*[3]. La clase mapeada "instrumentada" nos proveerá de los medios para referir a nuestra tabla en un contexto SQL, así como también persistir o cargar valores de las columnas desde la base de datos.

Fuera del proceso de mapeo, la clase sigue siendo, en su mayoría, una clase normal de Python, donde podemos definir los atributos y métodos que necesitemos para nuestra aplicación.

Crear un esquema

Una vez que tenemos construida nuestra clase a través del sistema declarativo, tenemos definida la información sobre la tabla de nuestro modelo, conocida como tabla metadato.

El objeto que utiliza *SQLAlchemy* para representar esta información para una tabla específica se llama **Table**, y el sistema declarativo la hace por nosotros. Podemos ver este objeto inspeccionado el atributo __table__:

```
>>> Author.__table__
Table('author', MetaData(bind=None),
            Column('id', Integer(), table=<author>, primary_key=True, nullable=False,
                        default=Sequence('author_id_seq',
                                                metadata=MetaData(bind=None))),
            Column('firstname', String(), table=<author>),
```

[2] https://docs.sqlalchemy.org/en/13/glossary.html#term-descriptors
[3] https://docs.sqlalchemy.org/en/13/glossary.html#term-instrumentation

```
Column('lastname', String(), table=<author>), schema=None)
```

El objeto Table es miembro de una colección conocida como **MetaData**. Cuando usamos el sistema declarativo, este objeto está disponible usando el tributo metadata de nuesta clase base declarativa.

MetaData es un registro que incluye la posibilidad de generar esquemas en la base de datos. Con el método *create_all* se crearán todas las tablas que aún no existen en la base de datos manejada por el motor especificado.

```
>>> engine = create_engine('sqlite:///:memory:')
>>> Base.metadata.create_all(engine)
```

En este ejemplo se crea la tabla *author* en una base de datos SQLite en memoria.

Crear una instancia de una clase mapeada

Una vez que el mapeo está completo podemos crear e inspeccionar objetos de tipo Author:

```
>>> author = Author(firstname='Joanne', lastname='Rowling')
>>> author.firstname
'Joanne'
>>> author.lastname
'Rowling'
>>> str(author.id)
‘None’
```

En este caso se crea un autor con nombre “Joanne” y apellido “Rowling”. Cuando accedo a los atributos *firstname* y *lastname,* vemos que el funcionamiento es el mismo que con cualquier objeto de Python. Notar que el *id* quedará en None hasta que el objeto se guarde en la base de datos ya que el mismo se completa con la secuencia cuando el objeto se guarda en la base.

Creando una sesión

Una vez creado el esquema, estamos en condiciones de comunicarnos con la base de datos. La comunicación con la base de datos se hace a través de una sesión.

Cuando iniciamos la aplicación al mismo nivel que la sentencia *create_engine*, definimos una clase **Session** que nos servirá para crear sesiones.

```
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
```

Cuando utilizamos la sesión por primera vez se recupera una conexión de un *pool* de conexiones que mantiene el motor de base de datos. Esta conexión la tendrá la sesión hasta que se haga un *commit* o se cierre la sesión.

Agregando y actualizando objetos

Para guardar un objeto en la base de datos vamos a agregarlo en la sesión.

```
>>> author = Author(firstname='Joanne', lastname='Rowling')
>>> session.add(author)
>>> our_author = session.query(Author).filter_by(firstname='Joanne').first()
>>> author is our_author
```

En este punto todavía no se insertó una fila en la base de datos. La sesión persistirá el objeto en la base de datos tan pronto como lo necesite, utilizando un proceso conocido como **flush**. Por ejemplo, si consulto a la base de datos por el objeto recién creado, primero se hará un *flush* e inmediatamente después se hace la consulta a la base de datos.

En el ejemplo, hacemos una consulta de un autor con nombre Joanne a la base de datos, y nos quedamos con el primero. Esta consulta nos va a traer el objeto autor correspondiente a “Joanne Rowling”, pero además veremos que es la misma instancia.

Esto es así porque la sesión la reconoce en su mapa interno de objetos y por eso devuelve exactamente la misma instancia que acabamos de insertar.

También podemos agregar una lista de objetos a la base de datos. Por ejemplo:

```
session.add_all([
    Author(firstname='John Ronald Reuel', lastname='Tolkien'),
    Author(firstname='Jose', lastname='Hernandez')])

>>> session.new
IdentitySet([John Ronald Reuel Tolkien, Jose Hernandez])
```

Además, Si modificamos un objeto, la sesión lo registra.

```
>>> author.firstname = 'Joanne K.'
>>> session.dirty
IdentitySet([Joanne K. Rowling])

>>> session.commit()
```

Luego haciendo *commit* de la sesión envíamos todos los cambios a la base y se harán los dos *insert* y el *update* correspondientes. Y como mencionamos se devuelve la conexión al *pool* de conexiones.

Una vez insertados los objetos en la base de datos, todos los identificadores y valores por defecto generados por la base de datos están disponibles en la instancia.

Rollback

Como la sesión trabaja dentro de una transacción, entonces podemos hacer rollback de todos los cambios no incluidos en un commit.

```
>>> author.firstname = 'Joanne'
>>> another_author = Author(firstname='Gabriel', lastname='Garcia Marquez')
>>> session.add(another_author)
>>> session.query(Author).filter(Author.firstname.in_(['Joanne', 'Gabriel'])).all()
>>> session.rollback()
>>> author.firstname
'Joanne K.'
>>> another_author in session
False
```

Luego de hacer llamar al método *rollback* de la sesión se revierten todos los cambios realizados después del último *commit*.

Buscando objetos

Para hacer consultas a la base de datos se utilizan objetos de la clase **Query** que se obtienen con el método *query()* de la sesión.

A continuación, se listan algunos ejemplos:

```
# Devuelve los autores ordenados por id. Imprimimos el nombre y el apellido de cada uno.
for instance in session.query(Author).order_by(Author.id):
    print(instance.firstname, instance.lastname)

# Devuelve el nombre y el apellido de cada autor y los imprimimos en pantalla.
for firstname, lastname in session.query(Author.firstname, Author.lastname):
    print(firstname, lastname)

# Devuelve el autor y el primer nombre del mismo. Imprimimos cada uno de estos datos.
for row in session.query(Author, Author.firstname).all():
    print(row.Author, row.firstname)

# Devuelve los autores, pero le asignamos una etiqueta al campo primer nombre.
# Luego imprimimos el valor de esa etiqueta.
for row in session.query(Author.firstname.label('firstname_label')).all():
    print(row.firstname_label)

# Devuelve el autor y el primer nombre del mismo. La diferencia con la Query 3 es que
# definimos un alias de la tabla. Imprimimos el autor.
author_alias = aliased(Author, name='author_alias')

for row in session.query(author_alias, author_alias.firstname).all():
    print(row.author_alias)

# Buscamos todos los autores, pero nos quedamos con los de las posiciones 2 y 3.
```

```
# La notación es como el slicing de las listas en Python.
for an_author in session.query(Author).order_by(Author.id)[1:3]:
    print(an_author)

# Filtramos los autores por los que tienen como nombre Joanne y nos quedamos con el
# nombre del mismo.
for name, in session.query(Author.firstname).filter_by(firstname='Joanne'):
    print(name)

# Igual a la consulta anterior, solo que filtramos por apellido del autor
for name, in session.query(Author.firstname).filter(Author.lastname=='Rowling'):
    print(name)



# Filtramos autor por nombre y apellido
for an_author in session.query(Author).\
        filter(Author.firstname=='Joanne').\
        filter(Author.lastname=='Rowling'):
    print(an_author)

# Nos quedamos con la cantidad de autores que tienen como primer nombre Joanne
>>> session.query(Author).filter(Author.firstname=='Joanne').count()
```

Construyendo una relación entre objetos

Para relacionar dos modelos se realizan los siguientes pasos:

1. Se crea una columna será la clave foránea hacia la tabla del modelo que queremos relacionar.
2. Por otro lado, se crea una relación con el modelo utilizando el constructor de relaciones llamado *relationship*.
3. La relación también se puede crear en el modelo relacionado para tener un acceso desde el objeto también. De esta manera la relación es bidireccional.

A continuación, se muestra una relación uno a muchos (un autor muchos libros):

```
# -*- coding: utf-8 -*-

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Sequence, ForeignKey
from sqlalchemy.orm import sessionmaker, relationship

Base = declarative_base()



class Author(Base):
   __tablename__ = 'author'

   id = Column(Integer, Sequence('author_id_seq'), primary_key=True)
   firstname = Column(String)
   lastname = Column(String)

   books = relationship("Book", order_by="Book.id", back_populates="author")
```

```
    def __repr__(self):
        return "{} {}".format(self.firstname, self.lastname)



class Book(Base):
    __tablename__ = 'book'
    id = Column(Integer, Sequence('book_id_seq'), primary_key=True)
    isbn = Column(String)
    title = Column(String)
    description = Column(String)
    author_id = Column(Integer, ForeignKey('author.id'))

    author = relationship("Author", back_populates="books")

    def __repr__(self):
        return "{}".format(self.title)
```

Trabajando con objetos relacionados

Una vez que creamos los modelos relacionados, podemos pedirle tanto a los libros su autor o al autor sus libros.

En el siguiente ejemplo se crea un autor y se le piden los libros que aún no tiene ninguno asociado. Después de le agregan dos libros y por último se agrega el autor en la sesión (que se agrega con sus libros) y se hace un *commit*.

```
>>> engine = create_engine('sqlite:///:memory:')
>>>  Base.metadata.create_all(engine)

>>> Session = sessionmaker(bind=engine)
>>> session = Session()

>>> j_rowling = Author(firstname='Joanne', lastname='Rowling')
>>> j_rowling.books
[]

>>> j_rowling.books = [Book(isbn='9788498387087',
                        title='Harry Potter y la Piedra Filosofal',
                        description='La vida de Harry Potter cambia para siempre el ...'),
                    Book(isbn='9788498382679',
                        title='Harry Potter y la camara secreta',
                        description='Tras derrotar una vez mas a lord Voldemort, ...')]

>>> j_rowling.books[1]
'Harry Potter y la camara secreta'

>>> j_rowling.books[1].title
'Harry Potter y la camara secreta'
>>> session.add(j_rowling)
```

```
>>> session.commit()


>>> j_rowling = session.query(Author).filter_by(firstname='Joanne').one()
>>> j_rowling.books
```

Consultando objetos relacionados

A continuación, se listan algunos ejemplos de cómo hacer consultas de objetos que están relacionados:

```
# Devuelve el autor y libro para este isbn. Los imprimimos en pantalla.
for an_author, a_book in session.query(Author, Book).\
        filter(Author.id==Book.author_id).\
        filter(Book.isbn=='9788498387087').\
        all():
   print(an_author)
   print(a_book)

# Devuelve el autor del libro con este isbn.
>>> session.query(Author).join(Book).\
        filter(Book.isbn=='9788498387087').\
        all()

# Devuelve los autores de los libros poniendo la condición explícitamente.
>>> session.query(Author).join(Book, Author.id==Book.author_id).all()

# Devuelve los autores de los libros especificando la relación de izquierda a derecha.
>>> session.query(Author).join(Author.books).all()

# Devuelve los autores de los libros para una relación específica.
>>> session.query(Author).join(Book, Author.books).all()

# Devuelve los autores de los libros utilizando un string
>>> session.query(Author).join('books').all()

# Devuelve el nombre del autor del libro utilizando un filtro exists.
# Es decir, si existe algún libro del autor
```

```
>>> stmt = exists().where(Book.author_id==Author.id)
>>> for name, in session.query(Author.firstname).filter(stmt):
        print(name)

# Devuelve el nombre del autor del libro utilizando un filtro any.
# Es decir, si el autor tiene algún libro
for name, in session.query(Author.firstname).filter(Author.books.any()):
   print(name)
# Devuelve el nombre del autor del libro utilizando un filtro like,
# parecido a any solo que se le puede definir una condición,
# en este caso que el apellido del autor contenga la subcadena de texto Row.
for name, in session.query(Author.firstname).\
        filter(Author.books.any(Author.lastname.like('%Row%'))):
   print(name)

# Devuelve los libros donde el autor no se llame Joanne
>>> session.query(Book).filter(~Book.author.has(Author.firstname=='Joanne')).all()
```

Borrando objetos

En esta sección veremos cómo borrar objetos de la base de datos. Tal como creamos las relaciones de los modelos, si borro el autor los libros no se borrarán. Con lo cual, quedarán libros sin autor. En este caso se dice que esos objetos están huérfanos.

```
>>> session.delete(j_rowling)
>>> session.query(Author).filter_by(firstname='Joanne').count()
0
>>> session.query(Book).filter(Book.isbn.in_(['9788498387087',
'9788498382679'])).count()
2
>>> session.rollback()
```

Si queremos que al borrar el objeto padre se borren los objetos hijos, debemos definir a la relación el atributo *cascade* con la opción *"all, delete, delete-orphan".* De esta manera al borrar un autor se borrarán sus libros también.

```
class Author(Base):
    __tablename__ = 'author'

    id = Column(Integer, Sequence('author_id_seq'), primary_key=True)
    firstname = Column(String)
    lastname = Column(String)

    books = relationship("Book", order_by="Book.id", back_populates="author",
                    cascade="all, delete, delete-orphan")

    def __repr__(self):
        return "{} {}".format(self.firstname, self.lastname)
```

Realizando este cambio en la relación, se puede ver que ahora si se borran todos los libros del autor cuando se borra el autor.

```
>>> del j_rowling.books[1]
>>> session.query(Book).filter(Book.isbn.in_(['9788498387087',
'9788498382679'])).count()
1

>>> session.delete(j_rowling)
>>> session.query(Author).filter_by(firstname='Joanne').count()
0

>>> session.query(Book).filter(Book.isbn.in_(['9788498387087',
'9788498382679'])).count()
0
```