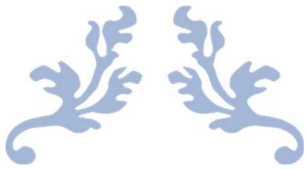


SQLALCHEMY

Por Damian R. Molín

1. Qué es SQLAlchemy?

SQLAlchemy es el kit de herramientas SQL de Python y el asignador relacional de objetos que brinda a los desarrolladores de aplicaciones todo el poder y la flexibilidad de SQL.

Proporciona un conjunto completo de patrones de persistencia de nivel empresarial bien conocidos, diseñados para un acceso a bases de datos eficiente y de alto rendimiento, adaptados a un lenguaje de dominio simple y Pythonic.

2. Instalación.

La instalación de SQLAlchemy se realiza a través de metodologías estándar de Python que se basan en herramientas de configuración, ya sea consultando setup.py directamente o utilizando pip u otros enfoques compatibles con herramientas de configuración.

2.1 Instalar a través de pip.

Cuando pip está disponible, la distribución se puede descargar desde PyPI e instalar en un solo paso:

```
pip install SQLAlchemy
```

Este comando descargará la última versión lanzada de SQLAlchemy de Python Cheese Shop y la instalará en su sistema. Para las plataformas más comunes, se descargará un archivo Python Wheel que proporciona extensiones Cython/C nativas preconstruidas.

Para instalar la versión preliminar más reciente, como 2.0.0b1, pip requiere que se use el indicador --pre:

```
pip install --pre SQLAlchemy
```

2.2 Instalación manual desde la distribución de origen.

Cuando no se instala desde pip, la distribución de origen se puede instalar mediante el script *setup.py*:

```
python setup.py install
```

La instalación de origen es independiente de la plataforma y se instalará en cualquier plataforma, independientemente de si las herramientas de compilación Cython / C están instaladas o no. Como se detalla en la siguiente sección Construyendo las extensiones de Cython, setup.py intentará construir usando Cython / C si es posible, pero de lo contrario recurrirá a una instalación de Python puro.

2.3 Instalación de una API de base de datos.

SQLAlchemy está diseñado para operar con una implementación de DBAPI creada para una base de datos en particular e incluye soporte para las bases de datos más populares. Las secciones de bases de datos individuales en Dialectos enumeran las DBAPI disponibles para cada base de datos, incluidos los enlaces externos.

2.3.1 Comprobación de la versión de SQLAlchemy instalada.

Esta documentación cubre SQLAlchemy versión 2.0. Si está trabajando en un sistema que ya tiene SQLAlchemy instalado, verifique la versión desde su indicador de Python de esta manera:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
```

Notas:
El soporte asyncio de SQLAlchemy depende del proyecto greenlet. Esta dependencia se instalará de manera predeterminada en plataformas de máquinas comunes; sin embargo, no se admite en todas las arquitecturas y es posible que no se instale de manera predeterminada en arquitecturas menos comunes.

3. Trabajando con SQLAlchemy sin Flask.

Utilizaremos el gestor de base de datos Postgres, asegúrate de disponer de el en tu equipo. No entraremos a explicar en detalle como instalarlo o utilizarlo ya que el alcance de este tutorial pretende limitarse a SQLALchemy. En caso de tener algún problema contacta con tu supervisor.

Lo primero que haremos será definir nuestro primer modelo. Un modelo no es más que la representación de nuestra tabla en la base de datos, y para poder crear un modelo lo haremos a través de una clase.

3.1 Definimos un modelo.

Utilizaremos como ejemplo la clase *User*(). Para que la clase pueda considerarse un modelo debe heredar de la clase *Base* y la clase *Base* la vamos a obtener a partir de la función *declarative_base.*

La función *Base = declarativa_base()* nos va a devolver nuestra clase *Base*, a partir de la cual seremos capaces de generar la cantidad de modelos que deseemos.

Mediante el atributo *__tablename__* definimos un nombre para nuestra tabla, en el caso del ejemplo "*Users*".

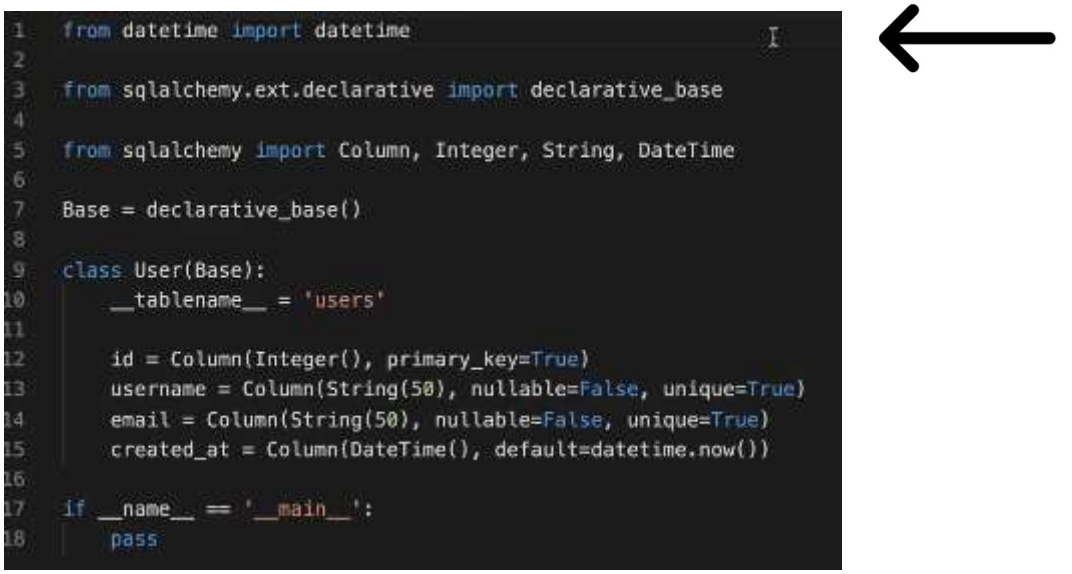
El siguiente paso sería definir los atributos para nuestra clase, que corresponderán con cada una de las columnas de nuestra tabla.

```python
class User(Base):
    __tablename__ = 'users'

    id =
    username =
    email =
    created_at =

if __name__ == '__main__':
    pass
```

Para definir el tipo de dato utilizaremos objetos de tipo *column*.

```python
from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy import Column, Integer, String, DateTime

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer(), primary_key=True)
    username = Column(String(50), nullable=False, unique=True)
    email = Column(String(50), nullable=False, unique=True)
    created_at = Column(DateTime(), default=datetime.now())

if __name__ == '__main__':
    pass
```

Si queremos utilizar la fecha exacta en la que se creó el registro deberemos importar el modulo *datetime*. Con esto ya tendremos listos los cuatro atributos de nuestro modelo.



```python
from datetime import datetime

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy import Column, Integer, String, DateTime

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer(), primary_key=True)
    username = Column(String(50), nullable=False, unique=True)
    email = Column(String(50), nullable=False, unique=True)
    created_at = Column(DateTime(), default=datetime.now())

if __name__ == '__main__':
    pass
```

Y para terminar con esta parte del tutorial sobre escribimos el modelo *username*.

```python
from datetime import datetime

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy import Column, Integer, String, DateTime

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer(), primary_key=True)
    username = Column(String(50), nullable=False, unique=True)
    email = Column(String(50), nullable=False, unique=True)
    created_at = Column(DateTime(), default=datetime.now())

    def __str__(self):
        return self.username

if __name__ == '__main__':
    pass
```

3.2 Conexión entra la base de datos y el modelo.

Nuestra base de datos no contiene ninguna tabla, así que crearemos nuestra primera tabla a partir del modelo. Para ello deberemos establecer la conexión entre nuestra aplicación y el gestor.

Lo haremos mediante la función engine que recibe como argumento un String que corresponde con la dirección a partir de la cual nos conectaremos con el servidor.

```python
from sqlalchemy import create_engine

engine = create_engine('postgresql://damian:@localhost/pythondb')
```

Para establecer una relación entre una conexión y el modelo, lo haremos a través de sesiones. El argumento que pasaremos por paréntesis será la conexión.

```python
    def __str__(self):
        return self.username

Session = sessionmaker(engine)

if __name__ == '__main__':
    pass
```

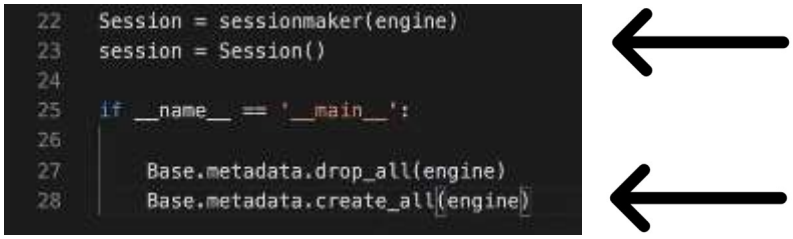
Una sesión no es más que un puente entre la conexión y nuestros modelos, y a través de las sesiones podremos modificar nuestra base de datos, ya sea para insertar registros, actualizarlos, eliminarlos o realizar consultas.

Gracias a la clase *Session* podremos generar la cantidad de sesiones que queramos. Gracias a esto SQLAlchemist nos permite trabajar con múltiples sesiones, muy útil por ejemplo en el caso de trabajar de forma concurrente.

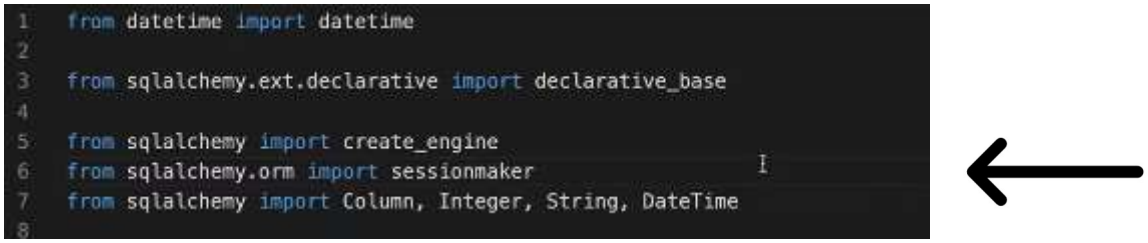
```
22   Session = sessionmaker(engine)
```

Instanciamos Session y añadimos el método *drop* que nos permitirá borrar absolutamente todo lo que haya en la base de datos y el método *create*. En ambos casos pasamos como argumento la conexión.

Gracias a estos dos métodos podremos eliminar y crear todas nuestras tablas de nuestra base de datos.



Por último importamos la función *sessionmaker.*



3.3 Probamos la conexión.

Abrimos una consola CMD, nos colocamos en la carpeta donde está el archivo y lo ejecutamos.

```
python main.py
```

Comprobamos la base de datos y podremos ver que la tabla se ha creado de forma correcta.

```
\dt;
```

3.4 Resumen hasta este punto.

Nuestras clases deben heredar de *Base*.

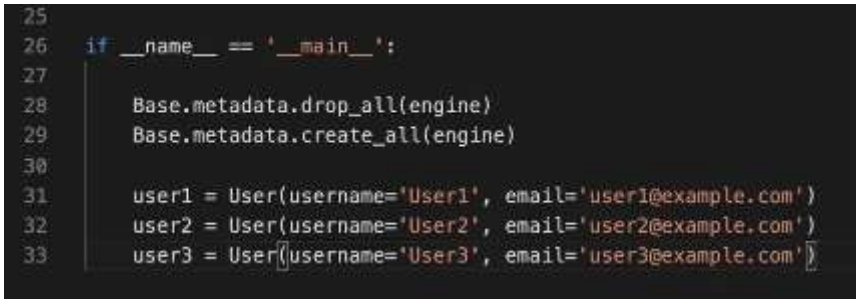
Mediante el atributo *__tablename__* indicamos el nombre de nuestra tabla.

Definimos los atributos para nuestra clase, o dicho de otra forma las futuras columnas que tendrá la tabla.

Mediante la clase Base con los métodos *drop_all* y *create_all* tendremos acceso a las tablas.

3.5 Persistencia

Hagamos tres instancias de nuestra tabla.



```
25
26  if __name__ == '__main__':
27
28      Base.metadata.drop_all(engine)
29      Base.metadata.create_all(engine)
30
31      user1 = User(username='User1', email='user1@example.com')
32      user2 = User(username='User2', email='user2@example.com')
33      user3 = User(username='User3', email='user3@example.com')
```

Si ejecutamos el script no habrá ningún error, pero no podremos hacer ninguna consulta porque nuestra tabla está totalmente vacía. Para poder conseguir que los datos persistan nos apoyaremos de la sesión.

```
26  if __name__ == '__main__':
27
28      Base.metadata.drop_all(engine)
29      Base.metadata.create_all(engine)
30
31      user1 = User(username='User1', email='user1@example.com')
32      user2 = User(username='User2', email='user2@example.com')
33      user3 = User(username='User3', email='user3@example.com')
34
35      session.add(user1)
36      session.add(user2)
37      session.add(user3)
38
```

De esta forma, al añadir las instancias en la sesión, haremos que los cambios se añadan a una pila o stack y después se pueda ejecutar aplicando todos los cambios definidos, de esta forma si podremos ver los cambios reflejados en la base de datos.

Y por último ejecutamos el método *commit()*

```
26  if __name__ == '__main__':
27
28      Base.metadata.drop_all(engine)
29      Base.metadata.create_all(engine)
30
31      user1 = User(username='User1', email='user1@example.com')
32      user2 = User(username='User2', email='user2@example.com')
33      user3 = User(username='User3', email='user3@example.com')
34
35      session.add(user1)
36      session.add(user2)
37      session.add(user3)
38
39      session.commit()
```

3.6 Consultas

Como ejemplo listaremos en consola todos los usuarios guardados en la tabla users.

A través de la *session* es como conectaremos el modelo con el gestor, y para hacer la consulta deberemos apoyarnos en dicho método.

Ejecutamos el método *query* que recibe como argumento la tabla en la que haremos la consulta. Y ejecutamos el método *all*() para que obtenga todos los registros de la tabla.

Esto nos devuelve una lista, un objeto que iteraremos e imprimiremos en consola.

```
39      session.commit()
40
41      # SELECT * FROM users;
42      users = session.query(User).all()
43
44      for user in users:
45          print(user)
```

Si queremos establecer una condición utilizaremos *filter*, por ejemplo usuario que tengan un ID mayor o igual a 2 y por ejemplo que el nombre sea User3.

```
41      # SELECT * FROM users;
42      # users = session.query(User).all()
43
44      # SELECT * FROM users WHERE id >= 2 and username = 'User3'
45      users = session.query(User).filter(
46          User.id >= 2
47      ).filter(
48          User.username == 'User3'
49      )
50
51      for user in users:
52          print(user)
```

Si queremos que solo nos muestre columnas determinadas, trabajaremos con el método query, poniendo entre paréntesis lo que queramos que se muestre.

```
45      users = session.query(User.id, User.username, User.email).filter(
46          User.id >= 2
47      )
48
```

Si ejecutamos veremos una diferencia importante ya que cuando al método *query* le colocamos una clase, el método nos devuelve instancias de la clase, pero si colocamos argumentos el método nos devolverá tuplas.

4. SQL ALchemy y Flask.

A diferencia del punto anterior, para facilitar la utilización del código en este apartado se proporcionará el mismo en un formato que se pueda copiar mediante CTRL+C y pegarlo con CTRL+V.

Antes de empezar será necesaria una nueva instalación para trabajar con Flask y SQLAlchemy.

```
pip install flask-sqlalchemy
```

4.1 Primeros pasos.

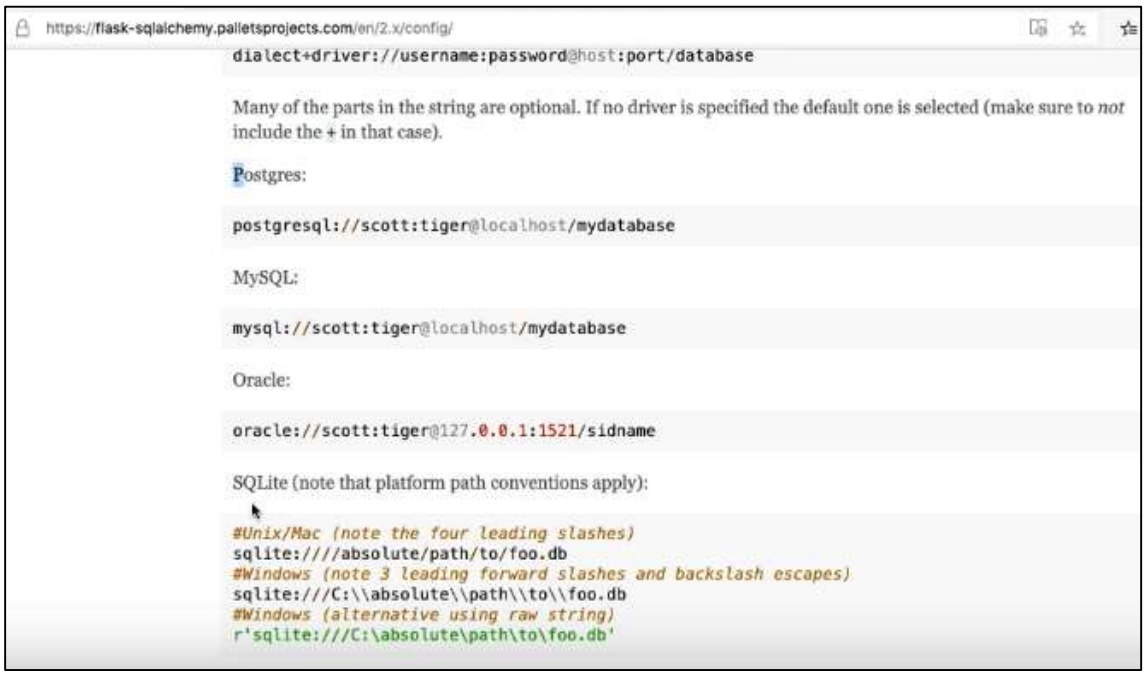
Escribimos en un archivo con nombre app.py, el nombre es un ejemplo cualquiera, lo incluyo en la descripción porque se debe hacer referencia en el código.

La primera línea de app.config indica nuestra base de datos y la última línea une la base de datos a nuestra aplicación en Flask.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)
app.config[“SQLALCHEMY_DATABASE_DAMIAN] =
app.config[“SQLALCHEMY_TRACK_MODIFICATIONS”] = False
db = SQLAlchemy(app)
```

4.2 Establecer la configuración de la base de datos.

En la documentación buscamos como establecer la conexión, en el ejemplo de uso anterior utilicé postgres, en este caso utilizaré sqlite.




```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)
app.config[“SQLALCHEMY_DATABASE_DAMIAN] = “sqlite:///C:\\Desktop\miBBDD.db”
app.config[“SQLALCHEMY_TRACK_MODIFICATIONS”] = False
db = SQLAlchemy(app)
```

4.3 Creamos el modelo

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_DAMIAN] = "sqlite:///C:\\Desktop\miBBDD.db"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db = SQLAlchemy(app)

class Delegacion(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String)
    país = db.Column(db.String(30))

    def __repr__(self):
        return "<Delegacion %r >" % self.nombre

if __name__ = "__main__":
     app.run(debug=True)
```

El método __repr__ nos permite identificar de forma sencilla los objetos.

4.4 Creamos la base de datos.

Abrimos un Python Shell en la terminal e importamos nuestra base de datos y creamos las tablas con los comandos import db y create_all.

```
>>> Python
  …….
>>> from app import db
>>> db.create_all()
```

4.5 Agregamos datos.

Para ello vamos a crear instancias

```
>>> from app import Delegacion
>>> Barcelona = Delegacion(nombre="bcn", pais="Esp")
>>> Berlin = Delegacion(nombre="ber", pais="Ale")
```

Ahora agregaremos datos en la base de datos valga la redundancia. Muy importante el commit del final o no se agregarán los datos.

```
>>> from app import Delegacion
>>> Barcelona = Delegacion(nombre="bcn", pais="Esp")
>>> Berlin = Delegacion(nombre="ber", pais="Ale")

>>> db.session.add(Barcelona)
>>> db.session.add(Berlin)
>>> db.session.commit()
```

4.6 Modificamos algunos valores.

Para hacer cualquier modificación en los datos es imprescindible referirnos a la instancia, por ejemplo cambiemos Berlin por Londres.

```
>>> Berlin.nombre = "Londres"
>>> db.session.commit()
```

4.7 Consultas básicas.

Para hacer consultas la metodología es muy similar a lo que ya hemos visto en el punto tres. Por ejemplo:

4.7.1 Consultar todos los nombres de los valores de la tabla.

```
>>> result = Delegacion.query.all()
>>> for r in result:
…       print(r.nombre)
```

4.7.2 Buscando un valor en particular. Con el first() devolverá tan solo el primer valor que encuentre, si utilizásemos all() nos devolvería todos los datos.

```
>>> bcn = Delegacion.query.filter_by(nombre="Barcelona").first()
>>> print (bcn.nombre)
```

4.7.3 Devolver todas las delegaciones que no estén en España.

```
>>> result = Delegacion.query.filter(Delegacion.pais!="esp").all()
```

4.8 Borrar datos.

En este caso borraremos la delegación de Barcelona. Referenciamos la instancia y hacemos el commit final como siempre.

```
>>> db.session.delete.(bcn)
>>> db.session.commit()
```