EBook Gratis

APRENDIZAJE sqlalchemy

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlalchem

y

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sqlalchemy

It is an unofficial and free sqlalchemy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sqlalchemy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con sqlalchemy

Observaciones

LA FILOSOFIA DE LA SQLALQUIMA

Desde el sitio web de SQLAlchemy :

> Las bases de datos SQL se comportan menos como las colecciones de objetos, mientras más tamaño y rendimiento comienzan a importar; las colecciones de objetos se comportan menos como tablas y filas, mientras más abstracción comienza a importar. SQLAlchemy tiene como objetivo dar cabida a estos dos principios.
>
> SQLAlchemy considera que la base de datos es un motor de álgebra relacional, no solo una colección de tablas. Las filas se pueden seleccionar no solo de tablas sino también de uniones y otras declaraciones de selección; cualquiera de estas unidades se puede componer en una estructura más grande. El lenguaje de expresión de SQLAlchemy se basa en este concepto desde su núcleo.
>
> SQLAlchemy es más famoso por su mapeador de objetos relacional (ORM), un componente opcional que proporciona el patrón del mapeador de datos, donde las clases se pueden mapear en la base de datos de forma abierta, de múltiples maneras, lo que permite que el modelo de objetos y el esquema de la base de datos se desarrollen en una Manera limpiamente desacoplada desde el principio.
>
> El enfoque general de SQLAlchemy para estos problemas es completamente diferente del de la mayoría de las otras herramientas de SQL / ORM, enraizadas en el llamado enfoque orientado a la complementariedad; En lugar de esconder detalles relacionales de objetos y SQL detrás de un muro de automatización, todos los procesos están completamente expuestos dentro de una serie de herramientas compacta y transparente. La biblioteca asume el trabajo de automatizar tareas redundantes mientras el desarrollador mantiene el control de cómo está organizada la base de datos y cómo se construye SQL.
>
> El objetivo principal de SQLAlchemy es cambiar la forma en que piensas sobre las bases de datos y SQL.

Versiones

Versión	Estado de liberación	Registro de cambios	Fecha de lanzamiento
1.1	Beta	1.1	2016-07-26
1.0	Lanzamiento actual	1.0	2015-04-16
0.9	Mantenimiento	0.9	2013-12-30

Versión	Estado de liberación	Registro de cambios	Fecha de lanzamiento
0.8	Seguridad	0.8	2013-03-09

Examples

Instalación o configuración

```
pip install sqlalchemy
```

Para las aplicaciones más comunes, particularmente las aplicaciones web, generalmente se recomienda que los principiantes consideren usar una biblioteca complementaria, como `flask-sqlalchemy` .

```
pip install flask-sqlalchemy
```

¡Hola Mundo! (SQLAlchemy Core)

Este ejemplo muestra cómo crear una tabla, insertar datos y seleccionar de la base de datos usando SQLAlchemy Core. Para obtener información sobre: SQLAlchemy ORM, consulte aquí .

Primero, necesitaremos conectarnos a nuestra base de datos.

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

> El motor es el punto de partida para cualquier aplicación SQLAlchemy. Es una "base de operaciones" para la base de datos real y su DBAPI, entregada a una aplicación SQLAlchemy a través de un conjunto de conexiones y un dialecto, que describe cómo hablar con un tipo específico de combinación de base de datos / DBAPI. El motor hace referencia tanto a un dialecto como a un conjunto de conexiones, que juntos interpretan las funciones del módulo DBAPI, así como el comportamiento de la base de datos.

Después de crear nuestro motor, necesitamos definir y crear nuestras tablas .

```
from sqlalchemy import Column, Integer, Text, MetaData, Table

metadata = MetaData()
messages = Table(
    'messages', metadata,
    Column('id', Integer, primary_key=True),
    Column('message', Text),
)

messages.create(bind=engine)
```

Para seguir explicando el objeto MetaData, vea lo siguiente de la documentación:

> Una colección de objetos de tabla y sus objetos secundarios asociados se conoce como metadatos de base de datos
>
> Definimos nuestras tablas dentro de un catálogo llamado MetaData, usando la construcción de la Tabla, que se asemeja a las sentencias CREATE TABLE de SQL regulares.

Ahora que hemos definido y creado nuestras tablas, ¡podemos comenzar a insertar datos! La inserción implica dos pasos. Componer la construcción de inserción , y ejecutar la consulta final.

```
insert_message = messages.insert().values(message='Hello, World!')
engine.execute(insert_message)
```

Ahora que tenemos datos, podemos usar la función de selección para consultar nuestros datos. Los objetos de columna están disponibles como atributos con nombre del atributo c en el objeto Tabla, lo que facilita la selección de columnas directamente. La ejecución de esta declaración de selección devuelve un objeto `ResultProxy` que tiene acceso a algunos métodos, fetchone () , fetchall () y fetchmany () , todos los cuales devuelven un número de filas de la base de datos consultadas en nuestra declaración de selección.

```
from sqlalchemy import select
stmt = select([messages.c.message])
message, = engine.execute(stmt).fetchone()
print(message)
```

```
Hello, World!
```

¡Y eso es! Consulte el Tutorial de expresiones SQL de SQLAlchemy para obtener más ejemplos e información.

¡Hola Mundo! (SQLAlchemy ORM)

Este ejemplo muestra cómo crear una tabla, insertar datos y seleccionar de la base de datos utilizando el **ORM de SQLAlchemy** . Para información sobre: SQLAlchemy Core, vea aquí .

Lo primero es lo primero, necesitamos conectarnos a nuestra base de datos, que es idéntica a la forma en que nos conectamos usando SQLAlchemy Core (Core).

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://')
```

Después de conectar y crear nuestro motor, necesitamos definir y crear nuestras tablas. Aquí es donde el lenguaje ORM de SQLAlchemy comienza a diferir mucho del Core. En ORM, el proceso de creación y definición de tablas comienza definiendo las tablas y las clases que usaremos para asignar esas tablas. Este proceso se realiza en un paso en ORM, que SQLAlchemy llama el sistema Declarativo .

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Ahora que nuestro mapeador base está declarado, podemos hacer una subclase de él para construir nuestros mapeos o `models` declarativos .

```
from sqlalchemy import Column, Integer, String

class Message(Base):
    __tablename__ = 'messages'

    id = Column(Integer, primary_key=True)
    message = Column(String)
```

Usando la clase base declarativa, terminamos creando una `Table` y un objeto `Mapper` . De la documentación:

> El objeto Table es un miembro de una colección más grande conocida como MetaData. Cuando se utiliza Declarativo, este objeto está disponible con el atributo .metadata de nuestra clase base declarativa.

Teniendo esto en cuenta, para crear todas las tablas que aún no existen, podemos llamar al siguiente comando, que utiliza el registro de metadatos de SQLAlchemy Core.

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

Ahora que nuestras tablas están asignadas y creadas, ¡podemos insertar datos! La inserción se realiza mediante la creación de instancias del asignador .

```
message = Message(message="Hello World!")
message.message # 'Hello World!
```

En este punto, todo lo que tenemos es una instancia de mensaje en el nivel del nivel de abstracción ORM, pero aún no se ha guardado nada en la base de datos. Para hacer esto, primero necesitamos crear una sesión .

```
from sqlalchemy.orm import sessionmaker

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

Este objeto de sesión es nuestro manejador de base de datos. Según los documentos de SQLAlchemy:

> recupera una conexión de un grupo de conexiones mantenidas por el motor y la mantiene hasta que confirmamos todos los cambios y / o cerramos el objeto de la sesión.

Ahora que tenemos nuestra sesión, podemos agregar nuestro nuevo mensaje a la sesión y confirmar nuestros cambios en la base de datos.

```
session.add(message)
session.commit()
```

Ahora que tenemos datos, podemos aprovechar el lenguaje de consulta ORM para recuperar nuestros datos.

```
query = session.query(Message)
instance = query.first()
print (instance.message) # Hello World!
```

Pero eso es solo el comienzo! Hay muchas más funciones que se pueden usar para redactar consultas, como `filter` , `order_by` y muchas más. Consulte el tutorial de SQLAlchemy ORM para obtener más ejemplos e información.

Lea Empezando con sqlalchemy en línea:
https://riptutorial.com/es/sqlalchemy/topic/1697/empezando-con-sqlalchemy

Capítulo 2: Conectando

Examples

Motor

El motor se utiliza para conectarse a diferentes bases de datos mediante una URL de conexión:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://user:pass@localhost/test')
```

Sin embargo, tenga en cuenta que el motor no establece realmente una conexión hasta que se utiliza por primera vez.

El motor crea automáticamente un grupo de conexiones, pero abre nuevas conexiones de manera perezosa (es decir, SQLAlchemy no abrirá 5 conexiones si solo solicita una).

Usando una conexión

Puede abrir una conexión (es decir, solicitar una del grupo) utilizando un administrador de contexto:

```
with engine.connect() as conn:
    result = conn.execute('SELECT price FROM products')
    for row in result:
        print('Price:', row['price'])
```

O sin, pero debe cerrarse manualmente:

```
conn = engine.connect()
result = conn.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
conn.close()
```

Ejecución implícita

Si solo desea ejecutar una sola instrucción, puede usar el motor directamente y abrirá y cerrará la conexión por usted:

```
result = engine.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
```

Actas

Puede usar `engine.begin` para abrir una conexión y comenzar una transacción que se retrotraerá si se produce una excepción o si se confirma de otra manera. Esta es una forma implícita de usar una transacción, ya que no tiene la opción de revertir manualmente.

```
with engine.begin() as conn:
    conn.execute(products.insert(), price=15)
```

Más explícitamente, puedes comenzar una transacción usando una conexión:

```
with conn.begin() as trans:
    conn.execute(products.insert(), price=15)
```

Tenga en cuenta que todavía llamamos a `execute` en la conexión. Como antes, esta transacción se confirmará o revertirá si se produce una excepción, pero también tenemos acceso a la transacción, lo que nos permite revertir manualmente usando `trans.rollback()` .

Esto se podría hacer más explícitamente así:

```
trans = conn.begin()
try:
    conn.execute(products.insert(), price=15)
    trans.commit()
except:
    trans.rollback()
    raise
```

Lea Conectando en línea: https://riptutorial.com/es/sqlalchemy/topic/2025/conectando

Capítulo 3: El ormo

Observaciones

El SQLAlchemy ORM está construido sobre SQLAlchemy Core . Por ejemplo, aunque las clases modelo usan objetos de `Column` , son parte del núcleo y allí se encontrará documentación más relevante.

Las partes principales del ORM son la sesión , la consulta y las clases asignadas (normalmente utilizando la extensión declarativa en SQLAlchemy moderno).

Examples

Convertir un resultado de consulta a dict

Primero la configuración para el ejemplo:

```
import datetime as dt
from sqlalchemy import Column, Date, Integer, Text, create_engine, inspect
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Session = sessionmaker()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

engine = create_engine('sqlite://')
Base.metadata.create_all(bind=engine)
Session.configure(bind=engine)

session = Session()
session.add(User(name='Alice', birthday=dt.date(1990, 1, 1)))
session.commit()
```

Si está consultando columnas individualmente, la fila es un `KeyedTuple` que tiene un método `_asdict` . El nombre del método comienza con un solo guión bajo, para que coincida con la API de `namedtuple` (¡no es privado!).

```
query = session.query(User.name, User.birthday)
for row in query:
    print(row._asdict())
```

Cuando se utiliza el ORM para recuperar objetos, esto no está disponible de forma predeterminada. Se debe utilizar el sistema de inspección SQLAlchemy.

```
def object_as_dict(obj):
    return {c.key: getattr(obj, c.key)
            for c in inspect(obj).mapper.column_attrs}

query = session.query(User)
for user in query:
    print(object_as_dict(user))
```

Aquí, creamos una función para realizar la conversión, pero una opción sería agregar un método a la clase base.

En lugar de utilizar `declarative_base` como anteriormente, puede crearlo desde su propia clase:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base:
    def _asdict(self):
        return {c.key: getattr(self, c.key)
                for c in inspect(self).mapper.column_attrs}
```

Filtración

Dado el siguiente modelo

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

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)
```

Puede filtrar columnas en la consulta:

```
import datetime as dt
session.query(User).filter(User.name == 'Bob')
session.query(User).filter(User.birthday < dt.date(2000, 1, 1))
```

Para el primer caso, hay un atajo:

```
session.query(User).filter_by(name='Bob')
```

Los filtros se pueden componer utilizando una relación AND al encadenar el método de `filter` :

```
(session.query(User).filter(User.name.like('B%'))
                    .filter(User.birthday < dt.date(2000, 1, 1)))
```

O de forma más flexible, utilizando los operadores de bits sobrecargados `&` y `|` :

```
session.query(User).filter((User.name == 'Bob') | (User.name == 'George'))
```

No olvide los paréntesis internos para tratar con la precedencia del operador.

Ordenar por

Dado un modelo básico:

```
class SpreadsheetCells(Base):
    __tablename__ = 'spreadsheet_cells'

    id = Column(Integer, primary_key=True)
    y_index = Column(Integer)
    x_index = Column(Integer)
```

Puede recuperar una lista ordenada encadenando el método `order_by` .

```
query = session.query(SpreadsheetCells).order_by(SpreadsheetCells.y_index)
```

Esto podría ser encadenado después de un `filter` ,

```
query = session.query(...).filter(...).order_by(...)
```

o para componer una consulta existente.

```
query = session.query(...).filter(...)
ordered_query = query.order_by(...)
```

También puede determinar la dirección de clasificación de una de las dos maneras siguientes:

1. Accediendo a las propiedades del campo `asc` y `dsc` :

```
query.order_by(SpreadsheetCells.y_index.desc()) # desc
query.order_by(SpreadsheetCells.y_index.asc()) # asc
```

2. Usando las funciones de los módulos asc y desc:

```
from sqlalchemy import asc, desc

query.order_by(desc(SpreadsheetCells.y_index)) # desc
query.order_by(asc(SpreadsheetCells.y_index)) # asc
```

Accediendo a los resultados de la consulta.

Una vez que tenga una consulta, puede hacer más con ella que solo iterar los resultados en un bucle for.

Preparar:

```
from datetime import date

class User(Base):
```

```
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

# Find users who are older than a cutoff.
query = session.query(User).filter(User.birthday < date(1995, 3, 3))
```

Para devolver los resultados como una lista, use `all()` :

```
reslist = query.all() # all results loaded in memory
nrows = len(reslist)
```

Puedes obtener un conteo usando `count()` :

```
nrows = query.count()
```

Para obtener solo el primer resultado, use `first()` . Esto es más útil en combinación con `order_by()` .

```
oldest_user = query.order_by(User.birthday).first()
```

Para consultas que deberían devolver solo una fila, use `one()` :

```
bob = session.query(User).filter(User.name == 'Bob').one()
```

Esto genera una excepción si la consulta devuelve varias filas o si no devuelve ninguna. Si la fila aún no existe, use `one_or_none()` :

```
bob = session.query(User).filter(User.name == 'Bob').one_or_none()
if bob is None:
    create_bob()
```

Esto seguirá provocando una excepción si varias filas tienen el nombre 'Bob'.

Lea El ormo en línea: https://riptutorial.com/es/sqlalchemy/topic/2020/el-ormo

Capítulo 4: Frasco-SQLAlchemy

Observaciones

Flask-SQLAlchemy agrega algunas funciones adicionales, como la destrucción automática de la sesión, asumiendo algunas cosas para usted que a menudo no son lo que necesita.

Examples

Una aplicación mínima

Para el caso común de tener una aplicación Flask, todo lo que tiene que hacer es crear su aplicación Flask, cargar la configuración elegida y luego crear el objeto SQLAlchemy al pasarle la aplicación.

Una vez creado, ese objeto contiene todas las funciones y ayudantes de sqlalchemy y sqlalchemy.orm. Además, proporciona una clase llamada Modelo que es una base declarativa que se puede usar para declarar modelos:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

Lea Frasco-SQLAlchemy en línea: https://riptutorial.com/es/sqlalchemy/topic/4601/frasco-sqlalchemy

Capítulo 5: La sesión

Observaciones

Una sesión realiza un seguimiento de los objetos ORM y sus cambios, gestiona las transacciones y se utiliza para realizar consultas .

Examples

Crear una sesión

Generalmente, una sesión se obtiene usando `sessionmaker` , que crea una clase de `Session` única para su aplicación. Más comúnmente, la clase `Session` está vinculada a un motor, lo que permite que las instancias usen el motor de manera implícita.

```
from sqlalchemy.orm import sessionmaker

# Initial configuration arguments
Session = sessionmaker(bind=engine)
```

El `engine` y la `Session` solo deben crearse una vez.

Una sesión es una instancia de la clase que creamos:

```
# This session is bound to provided engine
session = Session()
```

`Session.configure()` se puede usar para configurar la clase más adelante, por ejemplo, el inicio de la aplicación en lugar del tiempo de importación.

```
Session = sessionmaker()

# later
Session.configure(bind=engine)
```

Los argumentos pasados a la `Session` anulan directamente los argumentos pasados al `sessionmaker` .

```
session_bound_to_engine2 = Session(bind=engine2)
```

Añadiendo instancias

Se pueden agregar objetos nuevos o separados a la sesión usando `add()` :

```
session.add(obj)
```

Se puede agregar una secuencia de objetos usando `add_all()` :

```
session.add_all([obj1, obj2, obj3])
```

Se emitirá un INSERT a la base de datos durante la siguiente descarga, lo que sucede automáticamente. Los cambios se mantienen cuando se confirma la sesión.

Borrando instancias

Para eliminar objetos persistentes usa `delete()` :

```
session.delete(obj)
```

La eliminación real de la base de datos se producirá en la próxima descarga .

Lea La sesión en línea: https://riptutorial.com/es/sqlalchemy/topic/2258/la-sesion

Capítulo 6: SQLAlchemy Core

Examples

Convertir el resultado a dict

En el núcleo de SQLAlchemy, el resultado es `RowProxy` . En los casos en que desee un diccionario explícito, puede llamar a `dict(row)` .

Primero la configuración para el ejemplo:

```
import datetime as dt
from sqlalchemy import (
    Column, Date, Integer, MetaData, Table, Text, create_engine, select)

metadata = MetaData()
users = Table(
    'users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', Text, nullable=False),
    Column('birthday', Date),
)

engine = create_engine('sqlite://')
metadata.create_all(bind=engine)

engine.execute(users.insert(), name='Alice', birthday=dt.date(1990, 1, 1))
```

Luego, para crear un diccionario a partir de una fila de resultados:

```
with engine.connect() as conn:
    result = conn.execute(users.select())
    for row in result:
        print(dict(row))

    result = conn.execute(select([users.c.name, users.c.birthday]))
    for row in result:
        print(dict(row))
```

Lea SQLAlchemy Core en línea: https://riptutorial.com/es/sqlalchemy/topic/2022/sqlalchemy-core

Creditos

S. No	Capítulos	Contributors
1	Empezando con sqlalchemy	adarsh, Community, Mattew Whitt, RazerM, Stephen Fuhry
2	Conectando	RazerM
3	El ormo	Ilja Everilä, Mattew Whitt, RazerM, Tom Hunt
4	Frasco-SQLAlchemy	Ilya Rusin
5	La sesión	Ilja Everilä, RazerM
6	SQLAlchemy Core	RazerM