Kostenloses eBook

LERNEN sqlalchemy

Free unaffiliated eBook created from
Stack Overflow contributors.

#sqlalchem

y

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit der sqlalchemy

Bemerkungen

DIE PHILOSOPHIE DER SQLALCHEMIE

Von der SQLAlchemy-Website :

> SQL-Datenbanken verhalten sich weniger wie Objektsammlungen, je mehr Größe und Leistung von Bedeutung sind. Objektsammlungen verhalten sich weniger wie Tabellen und Zeilen, je mehr Abstraktion an Bedeutung gewinnt. SQLAlchemy zielt darauf ab, diese beiden Prinzipien zu berücksichtigen.
>
> SQLAlchemy betrachtet die Datenbank als relationale Algebra-Engine und nicht nur als eine Sammlung von Tabellen. Zeilen können nicht nur aus Tabellen, sondern auch aus Joins und anderen Select-Anweisungen ausgewählt werden. Jede dieser Einheiten kann zu einer größeren Struktur zusammengesetzt werden. Die Ausdruckssprache von SQLAlchemy baut auf diesem Konzept auf.
>
> SQLAlchemy ist am bekanntesten für seinen Object-Relational-Mapper (ORM), eine optionale Komponente, die das Data-Mapper-Muster bereitstellt. Dabei können Klassen auf verschiedene Weise auf die Datenbank abgebildet werden. Dies ermöglicht die Entwicklung des Objektmodells und des Datenbankschemas Von Anfang an sauber entkoppelt.
>
> Der allgemeine Ansatz von SQLAlchemy für diese Probleme unterscheidet sich grundlegend von dem der meisten anderen SQL / ORM-Tools, die auf einem sogenannten Komplimentaritäts-Ansatz basieren. Anstatt die relationalen Details von SQL und Objekten hinter einer Automatisierungswand zu verbergen, werden alle Prozesse vollständig in einer Reihe zusammenstellbarer, transparenter Tools dargestellt. Die Bibliothek übernimmt die Aufgabe, redundante Aufgaben zu automatisieren, während der Entwickler die Organisation der Datenbank und den Aufbau von SQL kontrolliert.
>
> Das Hauptziel von SQLAlchemy ist es, Ihre Denkweise über Datenbanken und SQL zu ändern!

Versionen

Ausführung	Freigabestatus	Änderungsprotokoll	Veröffentlichungsdatum
1.1	Beta	1.1	2016-07-26
1,0	Aktuelle Version	1,0	2015-04-16
0,9	Instandhaltung	0,9	2013-12-30

Ausführung	Freigabestatus	Änderungsprotokoll	Veröffentlichungsdatum
0,8	Sicherheit	0,8	2013-03-09

Examples

Installation oder Setup

```
pip install sqlalchemy
```

Für die meisten gebräuchlichen Anwendungen, insbesondere Webanwendungen, wird normalerweise empfohlen, dass Anfänger die Verwendung einer `flask-sqlalchemy` Betracht ziehen, z. B. `flask-sqlalchemy` .

```
pip install flask-sqlalchemy
```

Hallo Welt! (SQLAlchemy Core)

Dieses Beispiel zeigt, wie Sie eine Tabelle erstellen, Daten einfügen und mit SQLAlchemy Core aus der Datenbank auswählen. Informationen zum SQLAlchemy-ORM finden Sie hier .

Zuerst müssen wir eine Verbindung zu unserer Datenbank.

```
from sqlalchemy import create_engine

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

> Die Engine ist der Ausgangspunkt für jede SQLAlchemy-Anwendung. Es ist eine "Basis" für die eigentliche Datenbank und ihre DBAPI, die über einen Verbindungspool und einen Dialekt an eine SQLAlchemy-Anwendung übermittelt wird. Dort wird beschrieben, wie mit einer bestimmten Art von Datenbank / DBAPI-Kombination gesprochen wird. Die Engine verweist auf einen Dialekt und einen Verbindungspool, die zusammen die Modulfunktionen der DBAPI sowie das Verhalten der Datenbank interpretieren.

Nachdem wir unsere Engine erstellt haben, müssen wir unsere Tabellen definieren und erstellen .

```
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)
```

Weitere Informationen zum MetaData-Objekt finden Sie in den folgenden Dokumenten:

> Eine Sammlung von Table-Objekten und ihren zugehörigen untergeordneten Objekten wird als Datenbank-Metadaten bezeichnet
>
> Wir definieren unsere Tabellen alle in einem Katalog namens MetaData. Verwenden Sie dazu das Table-Konstrukt, das regulären SQL-Anweisungen CREATE TABLE ähnelt.

Nachdem wir unsere Tabellen definiert und erstellt haben, können wir mit dem Einfügen von Daten beginnen! Das Einfügen umfasst zwei Schritte. Einfügen des Einfügungskonstrukts und Ausführen der letzten Abfrage.

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

Jetzt, da wir Daten haben, können wir die select- Funktion verwenden, um unsere Daten abzufragen. Spaltenobjekte sind als benannte Attribute des Attributs c für das Table-Objekt verfügbar, sodass Sie Spalten direkt auswählen können. Wenn Sie diese select-Anweisung `ResultProxy` ein `ResultProxy` Objekt zurückgegeben, das Zugriff auf einige Methoden, fetchone () , fetchall () und fetchmany () , hat, die alle eine Anzahl von Datenbankzeilen zurückgeben, die in unserer select-Anweisung abgefragt wurden.

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

```
Hello, World!
```

Und das ist es! Weitere Beispiele und Informationen finden Sie im SQLAlchemy SQL Expressions Tutorial .

Hallo Welt! (SQLAlchemy ORM)

Dieses Beispiel zeigt, wie Sie eine Tabelle erstellen, Daten einfügen und mit dem **SQLAlchemy-ORM** aus der Datenbank auswählen. Informationen zu SQLAlchemy Core finden Sie hier .

Zuerst müssen wir uns mit unserer Datenbank verbinden, was identisch ist mit der, die wir mit SQLAlchemy Core (Core) verbinden würden.

```
from sqlalchemy import create_engine

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

Nachdem wir unsere Engine angeschlossen und erstellt haben, müssen wir unsere Tabellen definieren und erstellen. Hier beginnt sich die Sprache von SQLAlchemy ORM stark von Core zu unterscheiden. In ORM beginnt der Tabellenerstellungs- und -definitionsprozess mit der Definition der Tabellen und der Klassen, die wir zur Zuordnung zu diesen Tabellen verwenden. Dieser

Prozess wird in einem Schritt in ORM ausgeführt, das von SQLAlchemy als deklaratives System bezeichnet wird.

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

Jetzt, da unser Basis-Mapper deklariert ist, können wir daraus eine Unterklasse erstellen, um unsere deklarativen Mappings oder `models` zu erstellen.

```
from sqlalchemy import Column, Integer, String

class Message(Base):
    __tablename__ = 'messages'

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

Mit der deklarativen Basisklasse erstellen wir schließlich ein `Table` und `Mapper` Objekt. Aus den Dokumenten:

> Das Table-Objekt ist Mitglied einer größeren Sammlung, die als MetaData bezeichnet wird. Bei Verwendung von deklarativ ist dieses Objekt mit dem .metadata-Attribut unserer deklarativen Basisklasse verfügbar.

Um alle Tabellen zu erstellen, die noch nicht vorhanden sind, können Sie den folgenden Befehl aufrufen, der die MetaData-Registrierung von SQLAlchemy Core verwendet.

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

Jetzt, da unsere Tabellen abgebildet und erstellt sind, können wir Daten einfügen! Das Einfügen erfolgt durch die Erstellung von Mapper-Instanzen .

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

An dieser Stelle haben wir nur eine Instanz einer Nachricht auf der Ebene der ORM-Abstraktion, aber es wurde noch nichts in der Datenbank gespeichert. Dazu müssen wir zunächst eine Sitzung erstellen.

```
from sqlalchemy.orm import sessionmaker

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

Dieses Session-Objekt ist unser Datenbankhandler. Wie in den SQLAlchemy-Dokumenten beschrieben:

> Es ruft eine Verbindung aus einem Pool von Verbindungen ab, die von der Engine verwaltet werden, und behält es bei, bis alle Änderungen festgeschrieben und / oder das Sitzungsobjekt geschlossen werden.

Nun, da wir unsere Sitzung haben, können wir unsere neue Nachricht zur Sitzung hinzufügen und die Änderungen in die Datenbank übernehmen.

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

Jetzt, da wir über Daten verfügen, können wir die ORM-Abfragesprache nutzen, um unsere Daten abzurufen.

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

Aber das ist erst der Anfang! Es gibt viele weitere Funktionen, die zum `order_by` Abfragen verwendet werden können, wie z. B. `filter` , `order_by` und vieles mehr. Weitere Beispiele und Informationen finden Sie im SQLAlchemy ORM-Tutorial .

Erste Schritte mit der sqlalchemy online lesen:
https://riptutorial.com/de/sqlalchemy/topic/1697/erste-schritte-mit-der-sqlalchemy

Kapitel 2: Anschließen

Examples

Motor

Die Engine wird verwendet, um über eine Verbindungs-URL eine Verbindung zu verschiedenen Datenbanken herzustellen:

```
from sqlalchemy import create_engine

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

Beachten Sie jedoch, dass die Engine erst nach ihrer ersten Verwendung eine Verbindung aufbaut.

Das Modul erstellt automatisch einen Verbindungspool, öffnet aber neue Verbindungen träge (dh SQLAlchemy öffnet keine 5 Verbindungen, wenn Sie nur nach einem fragen).

Verbindung verwenden

Sie können eine Verbindung mit einem Kontextmanager öffnen (dh eine Verbindung aus dem Pool anfordern):

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

Oder ohne, aber es muss manuell geschlossen werden:

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

Implizite Ausführung

Wenn Sie nur eine einzelne Anweisung ausführen möchten, können Sie die Engine direkt verwenden, und die Verbindung wird für Sie geöffnet und geschlossen:

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

Transaktionen

Sie können `engine.begin` , um eine Verbindung zu öffnen und eine Transaktion zu beginnen, die zurückgesetzt wird, wenn eine Ausnahme `engine.begin` oder anderweitig festgeschrieben wird. Dies ist eine implizite Methode für die Verwendung einer Transaktion, da Sie nicht die Möglichkeit haben, einen manuellen Rollback durchzuführen.

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

Genauer gesagt, können Sie eine Transaktion über eine Verbindung beginnen:

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

Beachten Sie, dass wir auf der Verbindung immer noch `execute` aufrufen. Wie zuvor wird diese Transaktion festgeschrieben oder rückgängig gemacht, wenn eine Ausnahme `trans.rollback()` . Wir haben jedoch auch Zugriff auf die Transaktion, sodass wir mithilfe von `trans.rollback()` manuell `trans.rollback()` .

Dies könnte explizit wie folgt gemacht werden:

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

Anschließen online lesen: https://riptutorial.com/de/sqlalchemy/topic/2025/anschlie-en

Kapitel 3: Der ORM

Bemerkungen

Der SQLAlchemy-ORM basiert auf SQLAlchemy Core . Beispiel: Modellklassen verwenden zwar `Column` , sie sind jedoch Bestandteil des Kerns, und relevantere Dokumentation wird dort gefunden.

Die Hauptbestandteile des ORM sind die Session- , Query- und Mapping-Klassen (normalerweise unter Verwendung der deklarativen Erweiterung in der modernen SQLAlchemy.)

Examples

Konvertieren eines Abfrageergebnisses in Dict

Zuerst das Setup für das Beispiel:

```
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()
```

Wenn Sie Spalten einzeln abfragen, ist die Zeile ein `KeyedTuple` das eine `_asdict` Methode hat. Der Methodenname beginnt mit einem einzelnen Unterstrich, damit er mit der `namedtuple` API `namedtuple` (er ist nicht privat!).

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

Wenn Sie ORM zum Abrufen von Objekten verwenden, ist diese standardmäßig nicht verfügbar. Das SQLAlchemy- Inspektionssystem sollte verwendet werden.

```
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))
```

Hier haben wir eine Funktion für die Konvertierung erstellt. Eine Möglichkeit wäre jedoch, der Basisklasse eine Methode hinzuzufügen.

Anstatt `declarative_base` wie oben beschrieben zu verwenden, können Sie es aus Ihrer eigenen Klasse erstellen:

```
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}
```

Filterung

Gegeben das folgende Modell

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

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

Sie können Spalten in der Abfrage filtern:

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

Für den ersten Fall gibt es eine Verknüpfung:

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

Filter können mithilfe einer UND-Verknüpfung erstellt werden, indem die `filter` verkettet wird:

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

Oder flexibler mit den überladenen bitweisen Operatoren `&` und `|` :

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

Vergessen Sie nicht die inneren Klammern, um den Vorrang des Bedieners zu behandeln.

Sortieren nach

Gegeben ein Grundmodell:

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

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

Sie können eine geordnete Liste abrufen, indem Sie die `order_by` Methode `order_by` .

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

Dies könnte nach einem `filter` angekettet `filter` ,

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

oder um eine vorhandene Abfrage weiter zu erstellen.

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

Sie können die Sortierrichtung auch auf zwei Arten festlegen:

1. Zugriff auf die Feldeigenschaften `asc` und `dsc` :

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

2. Mit den Modulfunktionen asc und desc:

```
from sqlalchemy import asc, desc

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

Zugriff auf Abfrageergebnisse

Sobald Sie eine Abfrage haben, können Sie mehr als nur die Ergebnisse einer for-Schleife durchlaufen.

Konfiguration:

```
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))
```

Um die Ergebnisse als Liste zurückzugeben, verwenden Sie `all()` :

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

Sie können mit `count()` eine Zählung erhalten:

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

Um nur das erste Ergebnis zu erhalten, verwenden Sie `first()` . Dies ist in Kombination mit `order_by()` am nützlichsten.

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

Verwenden Sie für Abfragen, die nur eine Zeile zurückgeben sollen, `one()` :

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

Dies führt zu einer Ausnahme, wenn die Abfrage mehrere Zeilen oder keine zurückgibt. Wenn die Zeile möglicherweise noch nicht vorhanden ist, verwenden Sie `one_or_none()` :

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

Dies führt immer noch zu einer Ausnahme, wenn mehrere Zeilen den Namen 'Bob' haben.

Der ORM online lesen: https://riptutorial.com/de/sqlalchemy/topic/2020/der-orm

Kapitel 4: Die Sitzung

Bemerkungen

In einer Sitzung werden ORM-Objekte und ihre Änderungen nachverfolgt, Transaktionen verwaltet und Abfragen ausgeführt .

Examples

Sitzung erstellen

Eine Sitzung wird in der Regel unter Verwendung erhalten `sessionmaker` , die eine schafft `Session` Klasse einzigartig auf Ihre Bewerbung. Meistens ist die `Session` Klasse an eine Engine gebunden, sodass Instanzen die Engine implizit verwenden können.

```
from sqlalchemy.orm import sessionmaker

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

Die `engine` und die `Session` sollten nur einmal erstellt werden.

Eine Sitzung ist eine Instanz der von uns erstellten Klasse:

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

`Session.configure()` kann verwendet werden, um die Klasse später zu konfigurieren, z. B. Anwendungsstart statt Importzeitpunkt.

```
Session = sessionmaker()

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

An `Session` Argumente überschreiben direkt die an `sessionmaker` Argumente.

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

Instanzen hinzufügen

Neue oder getrennte Objekte können mit `add()` zur Sitzung hinzugefügt werden:

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

Eine Folge von Objekten kann mit `add_all()` hinzugefügt werden:

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

Ein INSERT wird während des nächsten Spülvorgangs an die Datenbank gesendet, was automatisch erfolgt. Änderungen werden beibehalten, wenn die Sitzung festgeschrieben wird.

Instanzen löschen

Löschen Sie persistierte Objekte mit `delete()` :

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

Das tatsächliche Löschen aus der Datenbank wird beim nächsten Flush ausgeführt .

Die Sitzung online lesen: https://riptutorial.com/de/sqlalchemy/topic/2258/die-sitzung

Kapitel 5: Flask-SQLAlchemy

Bemerkungen

Flask-SQLAlchemy fügt einige zusätzliche Funktionen hinzu, z. B. die automatische Zerstörung der Sitzung, wobei einige Dinge für Sie angenommen werden, die häufig nicht das sind, was Sie brauchen.

Examples

Eine minimale Anwendung

Für den üblichen Fall einer Flask-Anwendung müssen Sie lediglich Ihre Flask-Anwendung erstellen, die gewünschte Konfiguration laden und dann das SQLAlchemy-Objekt erstellen, indem Sie die Anwendung übergeben.

Sobald das Objekt erstellt wurde, enthält es alle Funktionen und Helfer von sqlalchemy und sqlalchemy.orm. Außerdem stellt es eine Klasse namens Model bereit, die eine deklarative Basis ist, die zur Deklaration von Modellen verwendet werden kann:

```
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
```

Flask-SQLAlchemy online lesen: https://riptutorial.com/de/sqlalchemy/topic/4601/flask-sqlalchemy

Kapitel 6: SQLAlchemy Core

Examples

Ergebnis in Dikt konvertieren

Im SQLAlchemy-Kern ist das Ergebnis `RowProxy` . Wenn Sie ein explizites Wörterbuch wünschen, können Sie `dict(row)` aufrufen.

Zuerst das Setup für das Beispiel:

```
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))
```

Dann erstellen Sie ein Wörterbuch aus einer Ergebniszeile:

```
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))
```

SQLAlchemy Core online lesen: https://riptutorial.com/de/sqlalchemy/topic/2022/sqlalchemy-core

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit der sqlalchemy	adarsh, Community, Mattew Whitt, RazerM, Stephen Fuhry
2	Anschließen	RazerM
3	Der ORM	Ilja Everilä, Mattew Whitt, RazerM, Tom Hunt
4	Die Sitzung	Ilja Everilä, RazerM
5	Flask-SQLAlchemy	Ilya Rusin
6	SQLAlchemy Core	RazerM