SQLAlchemy

The Python SQL Toolkit and Object Relational Mapper

SQLAlchemy

Muhammet S. AYDIN

Python Developer @ Metglobal

@mengukagan

SQLAlchemy

- No ORM Required
- Mature
- High Performing
- Non-opinionated
- Unit of Work
- Function based query construction
- Modular

SQLAlchemy

- Seperation of mapping & classes
- Eager loading & caching related objects
- Inheritance mapping
- Raw SQL

SQLAlchemy

Drivers:

PostgreSQL
MySQL
MSSQL
SQLite
Sybase
Drizzle
Firebird
Oracle

SQLAlchemy

Core

Engine
Connection
Dialect
MetaData
Table
Column

SQLAlchemy

Core

Engine

Starting point for SQLAlchemy app.

Home base for the database and it's API.

SQLAlchemy

Core

Connection

Provides functionality for a wrapped DB-API connection.

Executes SQL statements.

Not thread-safe.

SQLAlchemy

Core

Dialect

Defines the behavior of a specific database and DB-API combination.

Query generation, execution, result handling, anything that differs from other dbs is handled in Dialect.

SQLAlchemy

Core

MetaData

Binds to an Engine or Connection.

Holds the Table and Column metadata in itself.

SQLAlchemy

Core

Table

Represents a table in the database.
Stored in the MetaData.

SQLAlchemy

Core

Column

Represents a column in a database table.

SQLAlchemy

Let's get started.

SQLAlchemy

Core

Creating an engine:

```
4
5 engine = create_engine('postgresql://username:pwd@dbhost/dbname', echo=True)
6
```

SQLAlchemy

Core

Creating tables

Register the Table with MetaData.

Define your columns.

Call metadata.create_all(engine)
or
table.create(engine)

SQLAlchemy

Core

Creating tables

```
1  from sqlalchemy import create_engine, MetaData, Table, Column
2  from sqlalchemy.types import Integer, Unicode, UnicodeText
3  from sqlalchemy.schema import ForeignKey
4
5  engine = create_engine('postgresql://postgres:@localhost/test',
6                         echo=True)
7  metadata = MetaData()
8
9  countries_table = Table('countries', metadata,
10                          Column('id', Integer, primary_key=True),
11                          Column('name', Unicode(255)),
12                          Column('code', Unicode(255)))
13
14 indicators_table = Table('indicators', metadata,
15                           Column('id', Unicode(255), primary_key=True),
16                           Column('name', Unicode(255)))
17
18 data_table = Table('data', metadata,
19                     Column('country_id', ForeignKey("countries.id")),
20                     Column('indicator', ForeignKey("indicators.id")),
21                     Column('data', UnicodeText))
22
23 metadata.create_all(engine)
```

SQLAlchemy

Core

More on Columns

Columns have some important parameters.

index=bool, nullable=bool, unique=bool,
primary_key=bool, default=callable/scalar,
onupdate=callable/scalar,
autoincrement=bool

SQLAlchemy

Core

Column Types

Integer, BigInteger, String, Unicode, UnicodeText, Date, DateTime, Boolean, Text, Time

and

All of the SQL std types.

SQLAlchemy

Core

Insert

```
insert = countries_table.insert().values(
code='TR', name='Turkey')

conn.execute(insert)
```

SQLAlchemy

Core

Select

```
select([countries_table])
select([ct.c.code, ct.c.name])
select([ct.c.code.label('c')])
```

SQLAlchemy

Core

Select

```
select([ct]).where(ct.c.region == 'Europe & Central Asia')
```

```
select([ct]).where(or_(ct.c.ilike('%europe%', ctc.c.ilike('%asia%')))
```

SQLAlchemy

Core

Select A Little Bit Fancy

```
select([func.count(ct.c.id).label('count'),
ct.c.region]).group_by(ct.c.region).order_by(
'count DESC')
```

```
SELECT count(countries.id) AS count,
countries.region
FROM countries GROUP BY countries.region
ORDER BY count DESC
```

SQLAlchemy

Core

Update

```
ct.update().where(ct.c.id ==
1).values(name='Turkey', code='TUR')
```

SQLAlchemy

Core

Cooler Update

```
case_list = [(pt.c.id == photo_id, index+1) for
index, photo_id in enumerate(order_list)]

pt.update().values(photo_order=case(case_list))
```

```
UPDATE photos SET photo_order=CASE WHEN
(photos.id = :id_1) THEN :param_1 WHEN
(photos.id = :id_2) THEN :param_2 END
```

SQLAlchemy

Core

Delete

```
ct.delete().where(ct.c.id_in([60,71,80,97]))
```

SQLAlchemy

Core

Joins

```
select([ct.c.name,
dt.c.data]).select_from(ct.join(dt)).where(ct.
c.code == 'TRY')
```

SQLAlchemy

Core

Joins

```
select([ct.c.name,
dt.c.data]).select_from(join(ct, dt, ct.c.id ==
dt.c.country_id)).where(ct.c.code == 'TRY')
```

SQLAlchemy

Core

Func

A SQL function generator with attribute access.

simply put:

func.count() becomes COUNT().

SQLAlchemy

Core

Func

```
select([func.concat_ws(“ -> “, ct.c.name,
ct.c.code)])
```

```
SELECT concat_ws(%(concat_ws_2)s,
countries.name, countries.code) AS
concat_ws_1
FROM countries
```

SQLAlchemy

ORM

- Built on top of the core

- Applied usage of the Expression Language

- Class declaration

- Table definition is nested in the class

SQLAlchemy

ORM

Definition

```
1  from sqlalchemy import Column
2  from sqlalchemy.types import Integer, Unicode, UnicodeText
3  from sqlalchemy.schema import ForeignKey
4  from sqlalchemy.ext.declarative import declarative_base
5
6  Base = declarative_base()
7
8
9  class Country(Base):
10
11     __tablename__ = 'countries'
12
13     id = Column(Integer, primary_key=True)
14     code = Column(Unicode(3))
15     name = Column(Unicode(100))
16     region = Column(Unicode(100))
17
```

SQLAlchemy

ORM

Session

Basically it establishes all connections to the db.

All objects are kept on it through their lifespan.

Entry point for Query.

SQLAlchemy

ORM

Master / Slave Connection?

```
master_session =
sessionmaker(bind=engine1)
slave_session =
sessionmaker(bind=engine2)

Session = master_session()
SlaveSession = slave_session()
```

SQLAlchemy

ORM

Querying

Session.query(Country).filter(Country.name.
startswith('Tur')).all()

Session.query(func.count(Country.id)).one()

Session.query(Country.name,
Data.data).join(Data).all()

SQLAlchemy

ORM

Querying

```
Session.query(Country).filter_by(id=1).update({“name”: “USA”})
```

```
Session.query(Country).filter(~Country.region.in_('Europe & Central Asia')).delete()
```

SQLAlchemy

ORM

Relationships: One To Many

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

SQLAlchemy

ORM

Relationships: One To One

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False, backref="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

SQLAlchemy

ORM

Relationships: Many To Many

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

SQLAlchemy

ORM

Relationships: Many To Many

```
class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=True)
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=True)
    extra_data = Column(String(50))
    child = relationship("Child")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Association")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

SQLAlchemy

ORM

Relationship Loading

```
# set children to load lazily
session.query(Parent).options(lazyload('children')).all()

# set children to load eagerly with a join
session.query(Parent).options(joinedload('children')).all()

# set children to load eagerly with a second statement
session.query(Parent).options(subqueryload('children')).all()
```

SQLAlchemy

ORM

Relationship Loading

```
# load the 'children' collection using a second query which
# JOINS to a subquery of the original
class Parent(Base):
    __tablename__ = 'parent'

    id = Column(Integer, primary_key=True)
    children = relationship("Child", lazy='subquery')
```

SQLAlchemy

ORM

More?

http://sqlalchemy.org

http://github.com/zzzeek/sqlalchemy

irc.freenode.net #sqlalchemy