



SqlAlchemy: A Python ORM

So powerful, it brought me from PHP to Python in a hurry.

- Rick Harding
- [@mitechie](#)
- blog.mitechie.com
- *Richard Harding* on Google+

Anyone using an ORM?

ORM

Object Relational Mapper

(Remember the mapper part)

So who da what?

Turn a relational datastore (SQL) into pretty Python code

Time to thin the crowd a bit

ORM != NOT KNOWING SQL

SqlAlchemy Bad Reputation

- Hard to setup
- Poor/Confusing Documentation
- More than I need

SqlAlchemy is like an onion...layers

- Raw Sql

```
session.execute('SELECT * FROM users;')
```

- Sql Expression Language (Level 1)

```
select([users]).all()
```

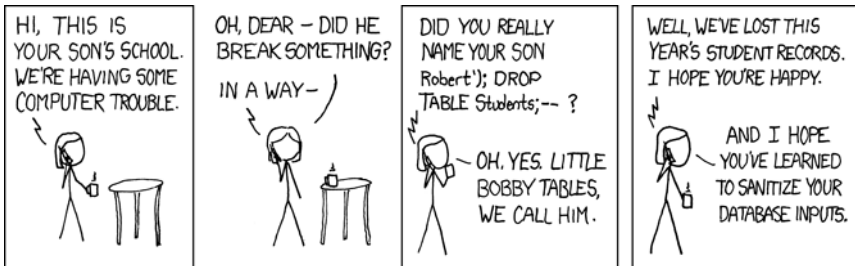
- ORM (Level 2)

```
Session.query(User).all()
```

When to use: Raw Sql

- one off scripts
- super performance
- No one in the office can figure out how to write query in ORM

Sql Injection, don't let it happen



```
session.execute(  
    text("DELETE FROM students WHERE id = :id", {id: 3})  
)
```

<http://xkcd.com/327/>

When to use: Sql Expression Language

- DB abstracted code
- Reusable Table models
- Simpler data types vs objects
- Basis for the higher level ORM, build your own!
- I use in migrations, db abstract, but don't depend on my ORM models

When to use: ORM

- Always!
- Start here, fall backwards
- You want pretty code

You say ORM I say declarative

Old style mapping (still works)

```
users_table = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', Unicode),
    Column('fullname', Unicode),
)

class User:
    pass

mapper(User, users_table)
```

Newer declarative style

- class extending Base
- table name (anything we want)
- columns, must have PK

```
class User(Base):  
    __tablename__ = "users"  
  
    id = Column(Integer, primary_key=True)  
    name = Column(Unicode)  
    fullname = Column(Unicode)
```

Advantages of the class

Add ons!

```
class User(base):  
    ...  
  
    username_min_length = 4  
  
    def __init__(self, username, fullname):  
        if len(username) < self.username_min_length:  
            raise ValueError  
  
        self.username = username  
        self.fullname = fullname
```

Advantages Cont'd

Models are just Python, code at will

```
def has_fullname(self):  
    if self.fullname:  
        return True  
    else:  
        return False  
  
rick = User.query.find(13)  
if rick.has_fullname():  
    print 'Yay!'
```

Python code works

```
filter_on = 'username'  
filter_val = 'rick'  
User.query.filter(getattr(User, filter_on) == filter_val).first()
```

You're convinced: back to basics

Connecting the powerful engine

engine == window to connection pool* and dialect* for your db

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://rick:passwd@localhost:5432/sa_demo')

result = engine.execute("select username from users")
for row in result:
    print "username:", row['username']
```

Session: ever read Patterns?

Unit of Work?

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

<http://martinfowler.com/eaCatalog/unitOfWork.html>

Everything in a transaction (or nested transactions)

Session: making Unit of Work cool

Let's pretend

```
rick = User.query.get(13)
rick.fullname = "Bob"

... elsewhere in the galaxy "Codebase"

logged_in = User.query.get(13)
print logged_in.fullname
>>> Bob
```

Session: let's make some

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine(...)

# create a configured "Session" class
Session = sessionmaker(bind=some_engine)

# create a Session
session = Session()

rick = User('rick', 'Rick Harding')
session.add(rick)
session.commit()
```

Session: Starting to put it together

```
Session = sessionmaker(bind=engine)
Base = declarative_base()
Base.metadata.bind = engine

# turns docs Session.query(User) into User.query
Base.query = Session.query_property(Query)

class User(Base):
    ...
```

Query Time: Users

Reminder of our object

```
class User(Base):  
    __tablename__ = "users"  
  
    id = Column(Integer, primary_key=True)  
    username = Column(Unicode(255))  
    fullname = Column(Unicode)  
    age = Column(Integer, default=10)  
    bio = Column(UnicodeText)  
    registered = Column(DateTime, default=datetime.now)
```

Query Time: .get

Based on PK (one or more bwuhahaha)

```
rick = User.query.get(13)

# what if multiple, tuple it
name = "Staples"
branch_id = 13
store = Store.query.get((name, branch_id))
```

Query Time: .filter

Chainable clauses, printable

```
print User.id == 23  
>>> users.id = :users_id_1
```

```
User.query.filter(User.username == 'rick')
```

```
User.query.filter(User.username != 'rick').\  
    filter(User.age > someage)
```

```
User.query.filter(User.username.in_('rick', 'bob')).\  
    filter(User.bio.contains('science'))
```

```
User.query.filter(or_(User.username == 'rick', User.username == 'bob'))
```

Query Time: building queries

```
def get_students(since=None, order_col=None):  
    qry = User.query  
  
    if since:  
        qry = qry.filter(User.registered >= since)  
  
    if order_col:  
        qry = qry.order_by(getattr(User, order_col))  
    else:  
        qry = qry.order_by(User.registered.desc())  
  
    return qry.all()
```

Query Time: getting results

Firing off the query

- `.one()` - exception
- `.first()` - None
- `.all()` - empty list

Query Time: Other query accessories

```
.group_by()  
.count()  
.order_by()  
.limit()  
.having()
```

Relations: How many of what?

Remember: you need to know sql

- One -> One
- One -> Many
- Many -> Many

Relations: A related object

One -> Many

```
class Email(Base):  
    __tablename__ = 'emails'  
  
    id = Column(Integer, primary_key=True)  
    user_id = Column(Integer, ForeignKey('users.id'))  
    addr = Column(String, unique=True, nullable=False)
```

Relations: Tie them together

Let User know about Email

```
class User(Base):  
    ...  
  
    emails = relation(Email,  
                      backref="user")  
  
rick = User.query.get(13)  
email.send(rick.emails[0])  
  
first_mail = rick.emails[0]  
print first_mail.user.username
```

Relations: Points of interest

- Only defined on one side, backref takes care of the rest
- defaults to lazy load, accessing rick.emails == another query

Lots of kwargs!

lazy, order_by, post_update, primaryjoin, secondaryjoin, uselist, viewonly, secondary, backref, back_populates, cascade, doc, foreign_keys, inner_join, join_depth,

Relations: One to One

Change it to one email per user

```
email = relation(Email,  
                  uselist=False,  
                  backref="user")
```

```
...
```

```
email.send(rick.email)
```

Relations: the mighty join

- left join
- inner join
- outer join

```
User.query.join(User.email).filter(Email.addr.endswith('@google.com'))
```

```
SELECT * FROM users, emails  
WHERE users.id = emails.user_id AND  
      emails.addr LIKE "%@google.com"
```

Relations: lazy lazy bums

- just joining == still lazy, but we can filter
- eager is the opposite of lazy

```
.join(User.email).options(contains_eager(User.email))
```


Organizing

Prepare for Rick's opinion

Instance vs Non Instance

```
User.XXX == a user instance
```

```
UserMgr.xxx = None, or a list of user objects
```

Relations: Organizing

```
class UserMgr(object):  
    """All non-instance helps for User class"""  
  
    def get_students(since=None):  
        ...  
  
    def find(email=None):  
        qry = User.query  
  
        if email:  
            qry = qry.join(User.email).options(contains_eager(User.email))  
            qry = qry.filter(email)  
  
    return qry.all()
```

Organizing

Building a model API. What do you want to write?

```
myuser = UserMgr.find(username="rick")  
gone = UserMgr.delete(id=15)  
  
user_list = UserMgr.get(age=21)  
  
for u in user_list:  
    print u.fullname
```

Relations: I can haz more?

```
class Phone(Base):  
    __tablename__ = 'emails'  
  
    id = Column(Integer, primary_key=True)  
    user_id = Column(Integer, ForeignKey('users.id'))  
    number = Column(String(10), unique=True, nullable=False)  
  
class User(Base):  
  
    email = relation(Email...  
    phone = relation(Phone...)
```


Relations: list by default, but dicts and sets rule

```
...
emails = relation(Email, column_mapped_collection('addr'))
phones = relation(Phone, collection_class=set)

rick = User.get(13)

# a dict so you can use dict items to check for existence
assert('rharding@mitechie.com' in rick.emails)

test_phones = {Phone('2485555555')}
rick.phones = rick.phones.union(test_phones)
```

Other tricks: autoload

- Great for existing dbs, quick scripts, ipython sessions

```
# does a query against the database at load time to load the columns
users_table = Table('users', meta, autoload=True)

class User(object):
    pass

mapper...
```

Other tricks: autoload declarative

- DON'T FOR THE LOVE OF !!!!!

```
class User(Base):  
    __tablename__ = 'users'  
    __table_args__ = (  
        UniqueConstraint('fullname'),  
        {'autoload': True}  
    )
```


Other tricks: Events!

- Who needs triggers
- Works cross db
- log items, update things
- I use for updating sqlite fulltext indexes on bookmarks

```
from sqlalchemy import event

def my_before_insert_listener(mapper, connection, target):
    # before we insert our record, let's say what server did this insert to
    # the db
    target.inserted_from = gethostname()

event.listen(User, 'before_insert', my_before_insert_listener)
```

Other tricks: Events Cont'd

- after (delete, update, insert)
- before (delete, update, insert)
- (create, populate) instance
- ...

Other tricks: Python properties

[illegible]

Let's show off something complicated

- Completion list for bookmarks
- Given selected tags "vagrant", "tips"
- Complete tag starting with "ub"

```
SELECT DISTINCT(tag_id), tags.name
FROM bmark_tags
JOIN tags ON bmark_tags.tag_id = tags.tid
WHERE bmark_id IN (
    SELECT bmark_id FROM bmark_tags WHERE tag_id IN (
        SELECT DISTINCT(t.tid) FROM tags t WHERE t.name IN ('vagrant', 'tips')
    )
)
AND tags.name LIKE ('ub%');
```

Show Off: cont'd

```
current_tags = Session.query(Tag.tid).\
                    filter(Tag.name.in_(current)).group_by(Tag.tid)

good_bmarks = select([bmarks_tags.c.bmark_id],
                    bmarks_tags.c.tag_id.in_(current_tags)).\
                    group_by(bmarks_tags.c.bmark_id).\
                    having('COUNT(bmark_id) >= ' + str(len(current)))

query = Session.query(Tag.name.distinct().label('name')).\
        join((bmarks_tags, bmarks_tags.c.tag_id == Tag.tid))
query = query.filter(bmarks_tags.c.bmark_id.in_(good_bmarks))
query = query.filter(Tag.name.startswith(prefix))

return Session.execute(query)
```