SQLAlchemy
by Example

DBAPI Is Horrible

```
cur.execute ("SELECT * FROM versions");
rows = cur.fetchall()
for i, row in enumerate(rows):
    print "Row", i, "value = ", row

Row 0 value =  (datetime.date(2007, 10, 18), '2.4.4', 'stable')
Row 1 value =  (datetime.date(2007, 4, 18), '2.5.1', 'stable')
Row 2 value =  (None, '2.6.0', 'devel')
Row 3 value =  (None, '3.0.0', 'alpha'
```

```
try:
    cur.execute ("""UPDATE versions SET status='stable' where version='2.6.0' """)
    cur.execute ("""UPDATE versions SET status='old' where version='2.4.4' """)
    db.commit()
except Exception, e:
    db.rollback()
```

Connecting

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

Session = sessionmaker()
engine = create_engine(DSN, **{ 'echo': False } )
Session.configure(bind=Backend._engine)

db = Session( )

try:
    # DO STUFF
except:
    db.rollback()
else:
    db.commit()

db.close()
```

Tables To Objects

```
from sqlalchemy.ext.declarative import declarative_base
from sqlahcmey import Column, ForeignKey, Integer

Base = declarative_base()

class Task(Base):
    __tablename__       = 'job'

    object_id           = Column('job_id',
                            ForeignKey('log.object_id'),
                            ForeignKey('job.job_id'),
                            ForeignKey('object_acl.object_id'),
                            primary_key=True)
    version             = Column('object_version', Integer)
    parent_id           = Column('parent_job_id', Integer,
                             ForeignKey('job.job_id'),
                             nullable=True)
...
```

```
new_task = Task()
db.add(new_task)
db.commit()
```

Make Your Own Types

Date values from strings

```
class JSONDateTime(TypeDecorator):
    """
    Allow storing a string into a datetime value, this allows for
    automatically conversion of the JSON date strings into date values
    """
    impl = DateTime

    def __init__(self, *arg, **kw):
        TypeDecorator.__init__(self, *arg, **kw)

    def process_bind_param(self, value, dialect):
        if value:
            if isinstance(value, basestring):
                if (len(value) == 19):
                    return datetime.strptime(value, '%Y-%m-%dT%H:%M:%S')
                elif (len(value) == 16):
                    return datetime.strptime(value, '%Y-%m-%dT%H:%M')
                elif (len(value) == 0):
                    return None
            elif isinstance(value, datetime):
                return value
            raise Exception('Cannot store value "{0}" as DateTime'.format(value))
        return None
```

Using Your Own Type

```
class LogEntry(Base):
    __tablename__ = 'logentry'
    LUID            = Column(String(32), primary_key=True)
    PARENT_LUID     = Column(String(32))
    ACTOR_LUID      = Column(String(32))
    objectid        = Column(Integer, index=True)
    actiondate      = Column(JSONDateTime)
    actorobjectid   = Column(Integer)
    message         = Column(String(255))
    action          = Column(String(20))
```

- This will create a DateTime column in the database that can be assigned to using a string.
- The ORM knows that this is a date-time value.

Avoiding DateTime TZ Hell

```
class UTCDateTime(sqla.types.TypeDecorator):
    impl = sqla.types.DateTime

    def convert_bind_param(self, value, engine):
        return value

    def convert_result_value(self, value, engine):
        if value:
               return value.replace(tzinfo=UniversalTimeZone))
       return None
```

Let's Do A Query

```
from sqlalchemy import and_, or_

query = db.query(Task).\
    filter(
        and_(
            Task.executor_id.in_([10100, 11530, ]),
            Task.title.ilike('%fred%'),
            Task.keywords.like('%fred%'),
            Task.state != '30_archived'
        )
    ).limit(150)
results = query.all()
```

Is it there? (presence query)

```
query = db.query(Task).\
    filter(
        and_(
            Task.executor_id.in_([10100, 11530, ]),
             Task.state != '30_archived'
        )
    )
result = query.first()
if result:
    print('SHAZAM!')
else:
    print('WHA WHA.')
```

One, and only one.

```
from sqlalchemy.exc import MultipleResultsFound, NoResultFound
query = db.query(Task).\
    filter(
        and_(
            Task.executor_id.in_([10100, 11530, ]),
             Task.state != '30_archived'
        )
    )
try:
    task = query.one()
except MultipleResultsFound:
    ...
except NoResultFound:
    …
else:
    print('SHAZAM!')
```

How many?

```
query = db.query(Task).\
    filter(
        and_(
            Task.executor_id.in_([10100, 11530, ]),
                  Task.state != '30_archived'
        )
    ).count()

count_is = query.first()
```

Distinct & order

```
 db.query(Contact.first_name, ).distinct()
```

```
db.query(Contact).\
    order_by(Contact.last_name.desc(), ).all()

db.query(Contact).\
    order_by(Contact.last_name.asc(), ).all()
```

Cost :(

Everything has a price. Everything, no exceptions, not ever.

Turning records into objects consumes cycles, memory; ultimately the convenience is paid for with a decrease in theoretical efficiency.

- A query that returns a thousand rows will create a thousand objects.

Upsides:

- Code is easier to read/write.
- No SQL injection issues.
- Common tasks can be automated away.
  - Less thinking about the plumbing.
    - In turn – fewer bugs.
  - Safer – SQLAlchemy won't even start if your defined relationships are pathological.

Cheaper!

```
query = db.query(Task.object_id, ).\
    filter(
        and_(
            Task.executor_id.in_([10100, 11530, ]),
             Task.state != '30_archived'
        )
    )
result = query.first()
if result:
    print('Object#{0}'.format(result[0]))
else:
    print('WHA WHA.')
```

We are not forced to use full objects!

Polymorphism

```
class _Doc(Base):
    __tablename__       = 'doc'

    ....
    _entity_name = \
      column_property(case([(_is_folder==1, "folder", ) ,
                            (_is_link==1, "link", ), ],
                           else_="document" ) )
    ...

    __mapper_args__     = { 'polymorphic_on': _entity_name }

class Folder(_Doc, KVC):
    __mapper_args__ = {'polymorphic_identity': 'folder'}
    ...

class Document(_Doc, KVC):
    __mapper_args__ = {'polymorphic_identity': 'document'}
    ...
```

The records in a table correspond to different classes based upon values in the record!

result = db.query(_Doc).all() will result in a list containing both Document and Folder objects.

Relations

```
from sqlalchemy.orm import relation

class Attachment(Base, KVC):
    __tablename__ = 'attachment'
    uuid = Column('attachment_id', String(255), primary_key=True)
    related_id = Column('related_id', Integer,
                 ForeignKey( 'person.company_id' ),
                 ForeignKey( 'enterprise.company_id' ),
                 ForeignKey( 'date_x.date_id' ),
                 ForeignKey( 'job.job_id' ),
....

Task.attachments = \
    relation(Attachment,
             lazy=False,
             uselist=True,
             primaryjoin=Attachment.related_id==Task.object_id)
```

Using The Relationship

```
if __name__ == '__main__':

    engine = create_engine('postgresql://OGo@127.0.0.1:5432/OGo', echo=False)

    session = sessionmaker(engine)()

    query = session.query(Task).filter(Task.owner_id == 10100)
    tasks = query.all()
    for task in tasks:
        print(task)
        for attachment in task.attachments:
            print(attachment)
```

Eager vs. Lazy

- Eager loading loads at the time the query is executed.
  - Data is ready to be used.
  - Reduced number of queries [back-and-forth]
- Lazy loading loads the related entities if and when the property is accessed.
  - Data is not loaded if it is not needed.
  - Less data is marshaled for the initial response.

Loader Strategies

- **select** – Every iteration of a property results in a SELECT statement.
  - This is the *default* loader strategy.
- **subquery** – Retrieve the data for the relation using a second SELECT statement, for all the entities returned in the first SELECT statement.
  - Efficient for one-to-many relationships, especially if they return large results.
- **joined** – Retrieve the data for a relation using a LEFT OUTER JOIN.
  - Efficient for one-to-one relationships and small results.
  - Can often materialize multiple objects of multiple classes with a single SELECT statement.
- noload – Do not materialize a relationship.
  - Override an eager load to a lazy load.

Using a loader strategy

```
query = db.query(Task).\
        filter(
                and_(
                        Task.executor_id.in_([10100, 11530, ]),
                         Task.state != '30_archived'
                )).\
        noload('attachments').\
        noload('info').\
        subqueryload('properties').\
        joinedload('notes').\
        lazyload('projects').\
        eagerload('creator')
```

A call to subqueryload or joinedload implies eagerload. A call to eagerload will eagerly load the relation using its default loader strategy.

This is so much easier than constructing the appropriate JOIN clauses for a literal SQL statement!!!

Using Relations Manually

```
op1 = aliased(ObjectProperty)
op2 = aliased(ObjectProperty)
op3 = aliased(ObjectProperty)

q = db.query( Process, op1, op2, op3 ).\
        join( Route, Route.object_id == Process.route_id ).\
        outerjoin( op1,
                and_(op1.parent_id == Route.object_id,
                          op1.namespace=='http://www.opengroupware.us/oie',
                          op1.name=='expireDays' ), ).\
        outerjoin( op2,
                and_(op2.parent_id == Route.object_id,
                          op2.namespace=='http://www.opengroupware.us/oie',
                          op2.name=='preserveAfterCompletion' ), ).\
        outerjoin(op3,
                and_(op3.parent_id == Route.object_id,
                          op3.namespace=='http://www.opengroupware.us/oie',
                          op3.name=='archiveAfterExpiration' ), ).\
        filter(and_(Process.state.in_( [ 'C', 'F', 'Z' ] ),
                           Process.status != 'archived' ) )
```

Relationships As Dictionaries

```
Contact.company_values = \
    relationship(
        'CompanyValue',
        primaryjoin = 'CompanyValue.parent_id==Contact.object_id',
        collection_class = attribute_mapped_collection( 'name' ),
        lazy = False,
        cascade = 'all, delete-orphan')
```

```
contact = db.query(Contact).get(10100)
print(contact.company_values['territory'])
```

Association Proxies

When an attribute of an object is the value for another related table.

```
Project._info = \
        relation(
                "ProjectInfo",
                 uselist=False,
                 backref=backref('project_info'),
                 primaryjoin=('ProjectInfo.project_id==Project.object_id'))

Project.comment = association_proxy('_info', 'comment')
```

Update

```
session.query(TmpXrefrRecord).\
        filter(
           and_(TmpXrefrRecord.batch_id == record.updated_by,
                     TmpXrefrRecord.record_id == record.record_id, )).\
update({TmpXrefrRecord.sku: record.sku,
        TmpXrefrRecord.phase_a1_status: record.phase_a1_status,
        TmpXrefrRecord.phase_a2_status: record.phase_a2_status,
        TmpXrefrRecord.phase_a3_status: record.phase_a3_status,
        TmpXrefrRecord.phase_s1_status: record.phase_s1_status,
        TmpXrefrRecord.phase_b1_status: 'NA',
        TmpXrefrRecord.phase_b2_status: 'NA',
        TmpXrefrRecord.vendor_stock_code:
record.vendor_stock_code,
        TmpXrefrRecord.hidden_form: record.hidden_form,
        TmpXrefrRecord._supersede: supersede_flag, },
       synchronize_session=False)
```

'fetch'
'evaluate'
False

But Operation XYZ Is Not Supported

EXPLAIN an SQLalchemy Query

- What is the query path of a query generated by SQLalchemy.
  - Manual queries can be easily testing using “EXPLAIN” / “EXPLAIN ANALYZE”.
  - SQLalchemy queries only appear in the logs, and criteria and statement must then be combined for testing.
    - Awkward!

Let's Extend SQLAlchemy!

EXPLAIN an SQLalchemy Query

```
import pprint
from sqlalchemy import *
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.sql.expression import Executable, ClauseElement, _literal_as_text
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, aliased

class Explain(Executable, ClauseElement):
    def __init__(self, stmt, analyze=False):
        self.statement = _literal_as_text(stmt)
        self.analyze = analyze

@compiles(Explain, 'postgresql')
def pg_explain(element, compiler, **kw):
    text = "EXPLAIN "
    if element.analyze:
        text += "ANALYZE "
    text += compiler.process(element.statement)
    return text
```

Using Out Extension

EXPLAIN an SQLalchemy Query

```
if __name__ == '__main__':

    Base = declarative_base()

    class Person(Base):
        __tablename__ = 'person'
        objectid   = Column('company_id', Integer, primary_key=True)
        first_name = Column('firstname', String)
        last_name  = Column('name', String)

    engine = create_engine('postgresql://OGo@127.0.0.1:5432/OGo', echo=False)

    sess = sessionmaker(engine)()

    query = sess.query(Person).filter(and_(Person.objectid > 10000,
                                           Person.last_name.ilike('W%')))

    print 'STATEMENT:\n {0}'.format(query.statement)
    x = sess.execute(Explain(query, analyze=True)).fetchall()
    pprint.pprint(x)
```

Results Of Our Extension

EXPLAIN an SQLalchemy Query

```
awilliam@linux-yu4c:~> python explain.py
STATEMENT:
 SELECT person.company_id, person.firstname, person.name
FROM person
WHERE person.company_id > :company_id_1 AND lower(person.name) LIKE
lower(:name_1)
[(u'Seq Scan on person  (cost=0.00..798.07 rows=1276 width=16)
                        (actual time=11.532..63.180 rows=1331 loops=1)',),
 (u"  Filter: ((company_id > 10000) AND ((name)::text ~~* 'W%'::text))",),
 (u'Total runtime: 63.307 ms',)]
```