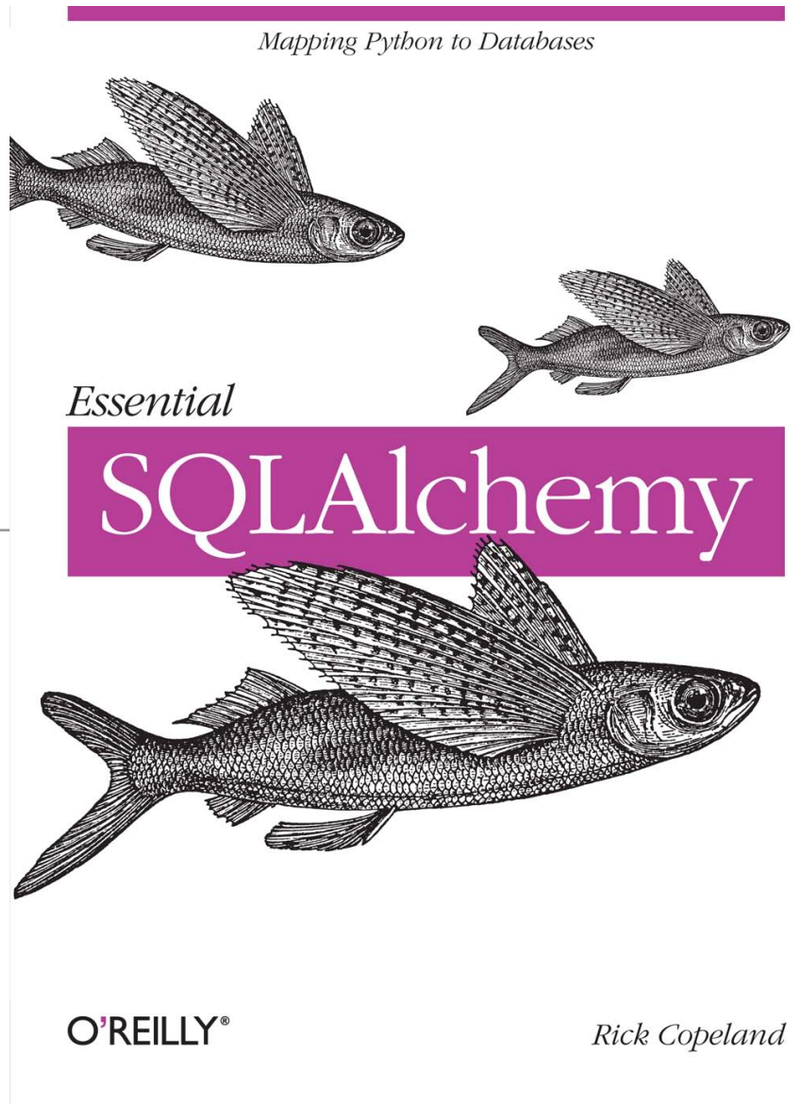
SQLAlchemy

Sutee Sudprasert

เนื้อหาจากหนังสือ Essential SQLAlchemy
และ SQLAlchemy 0.7 document



การใช้ SQL ตรงๆ ไม่ดีตรงไหน?

- SQL เป็นภาษาที่มีประสิทธิภาพในการค้นหาและจัดการข้อมูลในฐานข้อมูล แต่ว่าเขียนโปรแกรมติดต่อฐานข้อมูลผ่าน โดยการเรียกใช้คำสั่ง SQL ผ่านทาง DB-API ตรงๆ นั้นทำให้เกิดปัญหาตามมาได้

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

- ปัญหา SQL injection attacks
- ปัญหาการผูกติดกับ DB-API ที่ใช้

SQL injection attacks

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

- หากผู้ไม่ประสงค์ดีใส่ password เป็น

```
parrot'); DELETE FROM user; --
```

- SQL ที่ระบบจะต้องเรียกทำงานคือ

```
INSERT INTO user(user_name, password) VALUES ('rick', 'parrot');
DELETE FROM user; --'
```

  - ซึ่งคำสั่งนี้จะลบข้อมูลผู้ใช้ทั้งหมดในฐานข้อมูล

กรณีเปลี่ยน DB-API

Standard DB-API

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

Oracle DB-API

```
sql="INSERT INTO user(user_name, password) VALUES (:1, :2)"
cursor = conn.cursor()
cursor.execute(sql, 'rick', 'parrot')
```

What is SQLAlchemy?

- SQLAlchemy คือไพธอนไลบารี่ สร้างโดย Mike Bayer ซึ่งใช้สำหรับสร้างส่วนติดต่อระดับสูงสำหรับฐานข้อมูลเชิงสัมพันธ์ (relational database)
    - Oracle, DB2, MySQL, PostgreSQL, SQLite
- SQLAlchemy พยายามทำให้ส่วนการติดต่อฐานข้อมูล ไม่เข้าไม่ก่อกวนส่วนที่เป็นโค้ดโปรแกรม และ ทำให้การติดต่อฐานข้อมูลผ่าน SQL ไม่ขึ้นอยู่กับยี่ห้อของฐานข้อมูล
- การติดต่อฐานข้อมูลผ่าน SQLAlchemy ทำได้สองวิธีคือ
    - SQL expression language และ object-relational mapper (ORM)

What is SQLAlchemy?

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

- SQL expression language

```
statement = user_table.insert(user_name='rick', password='parrot')
statement.execute()
```

What is SQLAlchemy?

```
sql="INSERT INTO user(user_name, password) VALUES (%s, %s)"
cursor = conn.cursor()
cursor.execute(sql, ('rick', 'parrot'))
```

- object-relational mapper

```
Session = sessionmaker()
session = Session()

u = User()
u.user_name='rick'
u.password='parrot'
session.save(u)
session.flush()
```

SQLAlchemy Architecture

- Engine
- MetaData Management
- Types System
- SQL Expression Language
- Object Relational Mapper (ORM)

Engine

- เป็นจุดเริ่มต้นในการเริ่มใช้ SQLAlchemy (ต้องสร้างทุกครั้งก่อนเริ่มใช้)
- engine จะทำหน้าที่ในการจัดการ connection pool และ ทำให้การเรียกใช้ SQL ไม่ขึ้นกับฐานข้อมูล (ผ่านทาง SQL dialect layer)

วิธีที่ 1: สร้างจาก MetaData

```
metadata = MetaData('sqlite://')
engine = metadata.bind
```

วิธีที่ 2: สร้างขึ้นเองตรงๆ

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

Engine

- engine สามารถใช้ในการเรียกใช้คำสั่ง SQL โดยตรงได้ (ถ้าต้องการ)

```
for row in engine.execute("select user_name from tf_user"):
    print 'user name: %s' % row['user_name']
```

MetaData Management

- MetaData ใช้ในการเก็บและจัดการข้อมูลเกี่ยวกับโครงสร้างตาราง (database schema)

วิธีที่ 1: สร้างก่อน แล้วผูกกับ engine ทีหลัง

```
unbound_meta = MetaData()
db1 = create_engine('sqlite:///')
unbound_meta.bind = db1
```

วิธีที่ 2: สร้าง engine และผูกกับ MetaData

```
db2 = create_engine('sqlite:///test1.db')
bound_meta1 = MetaData(db2)
```

วิธีที่ 3: สร้างพร้อมกับ engine ในทีเดียว

```
bound_meta2 = MetaData('sqlite:///test2.db')
```

MetaData Management

- MetaData จะถูกใช้ในการสร้างตาราง

```
# Create a bound MetaData
meta = MetaData('sqlite://')

# Define a couple of tables
user_table = Table(
    'tf_user', meta,
    Column('id', Integer, primary_key=True),
    Column('user_name', Unicode(16), unique=True, nullable=False),
    Column('password', Unicode(40), nullable=False))

group_table = Table(
    'tf_group', meta,
    Column('id', Integer, primary_key=True),
    Column('group_name', Unicode(16), unique=True, nullable=False))

# Create all the tables in the (empty) database
meta.create_all()

# Select all the groups from the tf_group table
result_set = group_table.select().execute()
```

Types System

- เป็นส่วนในการเปลี่ยนชนิดข้อมูลในฐานข้อมูลให้เป็นชนิดข้อมูลใน Python
- SQLAlchemy ได้เตรียมคลาส (ซึ่งสืบทอดจาก TypeEngine) ไว้ที่โมดูล sqlalchemy.types ซึ่งในบางกรณีอาจไม่รองรับข้อมูลบางอย่าง ในกรณีเราสามารถสร้างคลาสใหม่ขึ้นมาเองได้

```
class ImageType(sqlalchemy.types.Binary):

    def convert_bind_param(self, value, engine):
        sfp = StringIO()
        value.save(sfp, 'JPEG')
        return sfp.getvalue()

    def convert_result_value(self, value, engine):
        sfp = StringIO(value)
        image = PIL.Image.open(sfp)
        return image
```

SQL Expression Language

- API สำหรับการค้นหาและปรับปรุงตารางที่ใช้งานในรูปแบบ Python และ ไม่ขึ้นอยู่กับฐานข้อมูล

```
select([user_table.c.user_name, user_table.c.password],
       where=user_table.c.user_name=='rick')
```

ถูกแปลงเป็น SQL **ดังนี้**

```
SELECT tf_user.user_name, tf_user.password
FROM tf_user
WHERE tf_user.user_name = ?
```

ค่าของ user_name **จะถูกส่งไปพร้อมกับ** SQL

SQL Expression Language

- ข้อดี
  - ปลอดภัย: เนื่องจากค่าที่ถูกใส่ใน bind parameters จะถูก escaped ก่อน ซึ่งทำให้ยากต่อการทำ SQL injection attacks
  - มีประสิทธิภาพ: เนื่องจากระบบมีโอกาสที่จะนำ query string เดิมกลับมาใช้ใหม่ได้ ซึ่งทำให้ระบบจัดการฐานข้อมูลไม่ต้องสร้าง execution plan ใหม่
  - ไม่ขึ้นกับฐานข้อมูล: SQLAlchemy จะจำกัดส่วนการแปลงให้เป็นภาษา SQL ที่สอดคล้องกับฐานข้อมูลที่เราต้องการจะใช้

Object Relational Mapper (ORM)

- การใช้แค่ Engine, Metadata, TypeEngine และ SQL expression language ก็สามารถใช้งานได้แล้ว แต่พลังที่แท้จริงของ SQLAlchemy อยู่ที่ ORM

- ORM ทำหน้าที่ในการเปลี่ยนตารางในฐานข้อมูลให้อยู่ในรูปแบบของ Python object โดยที่เราประกาศ object และ ตารางแยกจากกัน จากนั้นจึงใช้ mapper เชื่อมทั้งสองส่วนเข้าด้วยกัน

Object Relational Mapper (ORM)

- ประกาศตาราง

```
user_table = Table(
  'tf_user', metadata,
  Column('id', Integer, primary_key=True),
  Column('user_name', Unicode(16), unique=True, nullable=False),
  Column('email_address', Unicode(255), unique=True, nullable=False),
  Column('password', Unicode(40), nullable=False),
  Column('first_name', Unicode(255), default=''),
  Column('last_name', Unicode(255), default=''),
  Column('created', DateTime, default=datetime.now))

group_table = Table(
  'tf_group', metadata,
  Column('id', Integer, primary_key=True),
  Column('group_name', Unicode(16), unique=True, nullable=False))

user_group = Table(
  'user_group', metadata,
  Column('user_id', None, ForeignKey('tf_user.id'), primary_key=True),
  Column('group_id', None, ForeignKey('tf_group.id'), primary_key=True))
```

Object Relational Mapper (ORM)

- ประกาศ object

```
class User(object): pass
class Group(object): pass
```

- ประกาศ mapper

```
mapper(User, user_table,
       properties=dict( groups=relationship(Group,
                                            secondary=user_group,
                                            backref= ‘users’) )
mapper(Group, group_table)


# user1's "groups" property will automatically be updated
group1.users.append(user1)

# group2's "users" property will automatically be updated
user2.groups.append(group2)
```

Object Relational Mapper (ORM)

- การแก้ไข User คลาสเพื่อเก็บเฉพาะ hash ของ password

```
import sha
class User(object):
    def _get_password(self):
        return self._password

    def _set_password(self, value):
        self._password = sha.new(value).hexdigest()
    password=property(_get_password, _set_password)

    def password_matches(self, password):
        return sha.new(password).hexdigest() == self._password


mapper(User, user_table,
        properties=dict(_password=user_table.c.password))
```

Getting Started

วิธีการติดตั้ง SQLAlchemy

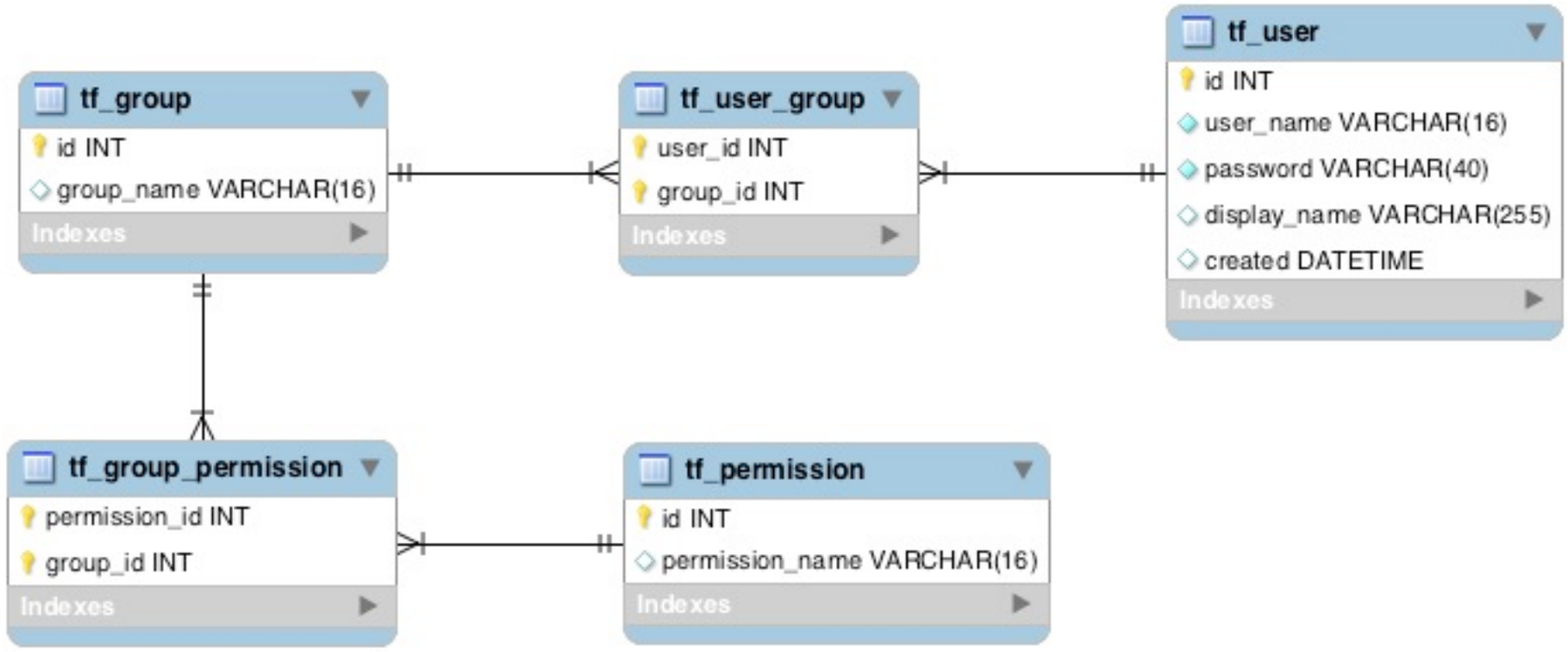
- ดาวน์โหลดไฟล์ http:// peak.telecommunity.com/dist/ez_setup.py
- พิมพ์คำสั่ง python ez_setup.py (ผู้ใช้ต้องมีสิทธิเป็น administrator)
  - เครื่องมือ easy_install จะถูกติดตั้งที่ c:\pythonxx\scripts

- พิมพ์คำสั่ง
  - easy_install -UZ SQLAlchemy

- ทดสอบการติดตั้งโดย

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
```

ตัวอย่างโครงสร้างฐานข้อมูล



ติดต่อฐานข้อมูลและสร้างตาราง

```
from sqlalchemy import *
from datetime import datetime

metadata = MetaData('sqlite:///tutorial.sqlite')

user_table = Table( 'tf_user', metadata,
  Column('id', Integer, primary_key=True),
  Column('user_name', Unicode(16), unique=True, nullable=False),
  Column('password', Unicode(40), nullable=False),
  Column('display_name', Unicode(255), default=u''),
  Column('created', DateTime, default=datetime.now()))
group_table = Table( 'tf_group', metadata,
  Column('id', Integer, primary_key=True),
  Column('group_name', Unicode(16), unique=True, nullable=False))
permission_table = Table( 'tf_permission', metadata,
  Column('id', Integer, primary_key=True),
  Column('permission_name', Unicode(16), unique=True, nullable=False))
user_group_table = Table( 'tf_user_group', metadata,
  Column('user_id', None, ForeignKey('tf_user.id'), primary_key=True),
  Column('group_id', None, ForeignKey('tf_group.id'), primary_key=True))
group_permission_table = Table( 'tf_group_permission', metadata,
  Column('permission_id', None, ForeignKey('tf_permission.id'), primary_key=True),
  Column('group_id', None, ForeignKey('tf_group.id'), primary_key=True))

metadata.create_all()
```

การค้นคืนและปรับปรุงฐานข้อมูล

```
>>> stmt = user_table.insert()
```

สร้าง INSERT statement

```
>>> stmt.execute(user_name='foo',
...               password='bar',
...               display_name='Foo Bar')
>>> stmt.execute(user_name='foo1',
...               password='bar',
...               display_name='Foo Bar Clone')
```

เพิ่มข้อมูล

```
>>> metadata.bind.echo = True
```

แสดง SQL ที่สร้างขึ้น

การค้นคืนและปรับปรุงฐานข้อมูล

```
>>> stmt = user_table.select()
>>> result = stmt.execute()
>>> for row in result:
...     print row
```

สร้าง SELECT statement

เลือกข้อมูลทั้งหมดใน user_table

```
>>> result = stmt.execute()
>>> row = result.fetchone()
```

ดูข้อมูลในแถวแรก

```
>>> row['user_name']
```

dict-like indexing

```
>>> row.password
>>> row.created
```

attribute lookup

```
>>> row.items()
```

สร้าง SELECT statement แบบมีเงื่อนไข

```
>>> stmt = user_table.select(user_table.c.user_name=='rick')
>>> print stmt.execute().fetchall()
```

การค้นคืนและปรับปรุงฐานข้อมูล

```
>>> stmt = user_table.update(user_table.c.user_name == 'foo')
>>> stmt.execute(password='secret123')
```

ปรับปรุงข้อมูลเดิมที่มีอยู่

```
>>> stmt = user_table.delete(user_table.c.user_name != 'foo')
>>> stmt.execute()
```

ลบข้อมูล

SQLAlchemy **ได้เตรียมฟังก์ชั่น** insert( ) select( ) update( ) **และ** delete( )
ไว้สำหรับการทำงานที่ซับซ้อน

การใช้งาน ORM

```
from sqlalchemy.orm import *

class User(object): pass
class Group(object): pass
class Permission(object): pass

mapper(User, user_table)
mapper(Group, group_table)
mapper(Permission, permission_table)
```

- unit of work (UOW) : การปรับปรุงฐานข้อมูลผ่านทาง object คำสั่งต่างๆ จะไม่ถูกส่งในที่ระบบจัดการฐานข้อมูลทันที จนกว่าผู้ใช้จะสั่ง flush( )
  - ประโยชน์ : ลดเวลาที่ใช้ในการเตรียมเพื่อติดต่อการข้อมูลในแต่ละครั้ง

การใช้งาน ORM

- คลาส Session : ใช้สำหรับการติดตามการเปลี่ยนแปลงของ object เพื่อที่จะปรับปรุงตารางให้สัมพันธ์กัน และ เป็นส่วนติดต่อเพื่อเรียกดูข้อมูลจากตาราง

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

- เรียกดูข้อมูลใน user_table ทั้งหมด

```
>>> query = session.query(User)
>>> list(query)
>>> for user in query:
...     print user.user_name
```

การใช้งาน ORM: query

- เรียกดูจาก primary key

```
>>> query.get(1)
```

- เรียกดูทั้งหมด

```
>>> query.all()
```

- เรียกดูแบบมีเงื่อนไข

```
>>> for user in query.filter_by(display_name='Rick Copeland'):
...     print user.id, user.user_name, user.password
...


>>> for user in query.filter(User.user_name.like('rick%')):
...     print user.id, user.user_name, user.password
...
```

การใช้งาน ORM: query

```
query.filter(User.name == 'ed') # equals

query.filter(User.name != 'ed') # not equals

query.filter(User.name.like('%ed%')) # LIKE

query.filter(User.name.in_(['ed', 'wendy', 'jack'])) # IN
query.filter(User.name.in_(
   session.query(User.name).filter(User.name.like('%ed%'))))

query.filter(~User.name.in_(['ed', 'wendy', 'jack'])) # NOT IN

query.filter(User.name == None) # IS NULL

query.filter(User.name != None) # IS NOT NULL

query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones')) # AND

query.filter(or_(User.name == 'ed', User.name == 'wendy')) # OR

query.filter(User.name.match('wendy')) # MATCH
```

การใช้งาน ORM: insert, update

- การเพิ่มข้อมูล

```
>>> newuser = User()
>>> newuser.user_name = 'mike'
>>> newuser.password = 'password'
>>> session.add(newuser)
```

ยังไม่เพิ่มข้อมูลจนกว่าจะสั่ง flush

```
>>> len(list(user_table.select().execute()))
```

ทดสอบนับในฐานข้อมูล

```
>>> query.count()
```

เมื่อเรียกดูผ่าน object ระบบจะ flush ข้อมูลให้เองโดยอัตโนมัติ
หากไม่ต้องการให้สร้าง Session = sessionmaker(atuoflush=False)

- ปรับปรุงข้อมูลเดิม

```
>>> newuser.password = 'new password'
>>> newuser.display_name = 'Michael'
>>> session.commit()
```

การใช้งาน ORM: delete

- ลบข้อมูล

```
>>> session.delete(newuser)
>>>
>>> session.commit()
```

การสร้างคลาสแบบ declarative

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
class User(Base):
  __tablename__ = 'tf_user'

  id = Column(Integer, primary_key=True),
  user_name = Column(Unicode(16), unique=True, nullable=False)
  password = Column(Unicode(40), nullable=False)
  display_name = Column(Unicode(255), default=u'')
  created = Column(DateTime, default=datetime.now())

  def __init__(self, user_name, password, display_name):
    self.user_name = user_name
    self.password = password
    self.display_name = display_name

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

user_table = User.__table__ # access the mapped Table
```

การกำหนด attributes

```
class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column("some_table_id", Integer, primary_key=True)
```

เราสามารกำหนดชื่อของ field ที่เป็น primary key เองได้ ซึ่งถ้าไม่กำหนดจะเป็น “id”

```
SomeClass.data = Column('data', Unicode)
SomeClass.related = relationship(RelatedInfo)
```

เราสามารถเพิ่ม field เข้าไปหลังจากการสร้างคลาสได้ ซึ่งค่าใน Table และ mapper จะถูกเปลี่ยนให้ถูกต้องโดยอัตโนมัติ

การสร้างความสัมพันธ์ (Relationship)

```
# one-to-many relationship

class Address(Base):
  __tablename__ = 'tf_address'

  id = Column(Integer, primary_key=True)
  email = Column(Unicode(50), nullable=False, unique= True)
  user_id = Column(Integer, ForeignKey('tf_user.id'))

  user = relationship(User,
                      backref=backref('addresses',order_by=id))

  def __init__(self, email, user=None):
    self.email = email
    self.user = user
```

การใช้งานวัตถุที่เกี่ยวข้องกัน (Related Objects)

```
>>> jack = User('jack', 'abcdef', 'Jack Bean')
>>> jack.addresses

>>> Address('jack1@hotmail.com', jack)
>>> Address('jack2@gmail.com', jack)

>>> jack.addresses[0]
>>> jack.addresses[0].user

>>> session.add(jack) # add jack and two addresses
>>> session.commit()

>>> jack = session.query(User).filter_by(name='jack').one()
>>> jack

>>> jack.addresses    # lazy loading relationship
```

การสร้างความสัมพันธ์ (Relationship)

```
# many-to-many relationship

author_keyword_table = Table(
    'author_keyword', Base.metadata,
    Column('author_id', Integer, ForeignKey('authors.id')),
    Column('keyword_id', Integer, ForeignKey('keywords.id'))
    )

class Author(Base):
    __tablename__ = 'authors'
    id = Column(Integer, primary_key=True)
    keywords = relationship("Keyword",
                            secondary=author_keyword_table,
                            backref='authors')

class Keyword(Base):
    __tablename__ = 'keywords'
    id = Column(Integer, primary_key=True)
```

การใช้งานวัตถุที่เกี่ยวข้องกัน (Related Objects)

```
>>> author1 = Author()
>>> author1.keywords

>>> keyword1 = Keyword()
>>> keyword2 = Keyword()
>>> author1.keywords.append(keyword1)
>>> author1.keywords.append(keyword2)

>>> session.add(author1) # add author1 and two keywords
>>> session.commit()

>>> author = session.query(User).one()
>>> author.keywords

>>> keyword = session.query(Keyword).one()
>>> keyword.authors
```

การสร้างความสัมพันธ์ (Relationship)

```python
# one-to-one relationship

class Foo(Base):
    __tablename__ = 'foos'
    id = Column(Integer, primary_key=True)
    bar_id = Column(Integer, ForeignKey(Bar.id))
    bar = relationship(Bar, uselist=False)

class Bar(Base):
    __tablename__ = 'bars'
    id = Column(Integer, primary_key=True)
```

Webshop database