CSE 106

Lecture 14 – ORM and SQLAlchemy

Acknowledgement:

https://www.altexsoft.com/blog/object-relational-mapping

https://www.sqlalchemy.org

https://flask-sqlalchemy.palletsprojects.com/en/2.x/

Object Relational Mapper (ORM)

- Provides an object-oriented layer between relational databases and object-oriented programming languages without writing SQL queries
- Creates highly-abstract data models
- Maps OOP code to/from SQL

Pros of an ORM

- Productivity
  - Don’t need to translate your object to/from SQL
- Application Design
  - The ORM implements design patterns to force you to use best practices for application design
- Code Reuse
  - Functions, classes and libraries using the same model
- Reduced Testing
  - The ORM is already tested and you can rely on it

Cons of an ORM

- Learning curve
  - Just one more thing to learn
- Performance
  - Not always the optimal SQL query for complex queries
- Still Need to Know SQL
  - Knowing SQL can help workaround mapping issues
- Potential Mapping Issues
  - Sometimes the mapping can be incorrect

SQLAlchemy

- The Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL
- Provides a full suite of enterprise-level persistence patterns
- Designed for efficient and high-performing database access
- Adapted into a simple and Pythonic domain language

Who uses SQLAlchemy

- Yelp!
- reddit
- DropBox
- The OpenStack Project
- Survey Monkey

Flask-SQLAlchemy - Overview

- An extension for Flask that adds support for SQLAlchemy to your application
- Aims to simplify using SQLAlchemy with Flask
  - Provides useful defaults and extra helpers to simplify common tasks
- Flask-SQLAlchemy is
  - Fun to use
  - Incredibly easy for basic applications
  - Readily extends for larger applications
- Install with pip: `pip install Flask-SQLAlchemy`

Getting Started – Data Model

```
# myapp.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///example.sqlite"
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
```

Getting Started – Create/Drop DB

```
# create_db.py
from myapp import db
db.create_all()
```

```
# drop_db.py
from myapp import db
db.drop_all()
```

Getting Started – Add Data

```
# add_users.py

from myapp import db, User

admin = User(username='admin', email='admin@example.com’)
guest = User(username='guest', email='guest@example.com’)

db.session.add(admin)
db.session.add(guest)

db.session.commit()
```

Getting Started – Query Data

```
# query.py

from myapp import db, User

# Queries all users
User.query.all()
# Queries first user
User.query.filter_by(username='admin').first()
```

Wake-up!

- https://youtu.be/tWpwwa8I7BY

Column Datatypes

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
```

Type	Description
Integer	an integer
String (size)	a string with a maximum length (optional in some databases, e.g. PostgreSQL)
Text	some longer unicode text
DateTime	date and time expressed as Python datetime object.
Float	stores floating point values
Boolean	stores a boolean value
PickleType	stores a pickled Python object
LargeBinary	stores large arbitrary binary data

Simple Relationships – Model Creation

```
from datetime import datetime
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    body = db.Column(db.Text, nullable=False)
    pub_date = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    category_id = db.Column(db.Integer, db.ForeignKey('category.id'), nullable=False)
    category = db.relationship('Category', backref=db.backref('posts', lazy=True))
    def __repr__(self):
        return '<Post %r>' % self.title

class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    def __repr__(self):
        return '<Category %r>' % self.name
```

Simple Relationships – Data Creation

```
py = Category(name='Python’)
# create it using category as an input argument
Post(title='Hello Python!', body='Python is pretty cool', category=py)

# Or make the post as a variable and add it
p = Post(title='Snakes', body='Sssssssss’)
py.posts.append(p)

db.session.add(py)
```

Simple Relationships – Data Query

```
# print out the posts in the category py
py.posts
# prints: [<Post 'Hello Python!'>, <Post 'Snakes'>]
# notice that both posts are added, even though they are added in different ways

# query with the parent relationship to category
Post.query.with_parent(py).filter(Post.title != 'Snakes').all()
# prints: [<Post 'Hello Python!'>]
```

Many-to-Many Relationships

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=True)
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags, lazy='subquery’,
           backref=db.backref('pages', lazy=True))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

Add/Delete Records

```
from myapp import User


# Add user
me = User('admin', 'admin@example.com’)
db.session.add(me)
db.session.commit()


# Delete user
db.session.delete(me)
db.session.commit()
```

Querying Records

```
# Retrieve a user by username
peter = User.query.filter_by(username='peter').first()
peter.id
# prints: 2
peter.email
# prints: 'peter@example.org’

sam = User.query.filter_by(username='sam').first()
sam is None
# prints: True
```

id	*username*	*email*
1	admin	admin@example.com
2	peter	peter@example.org
3	guest	guest@example.com

Querying Records

```
# Selecting a bunch of users by a more complex expression
User.query.filter(User.email.endswith('@example.com')).all()
# Prints: [<User 'admin'>, <User 'guest’>]

# Ordering users by something:
User.query.order_by(User.username).all()
# Prints: [<User 'admin'>, <User 'guest'>, <User 'peter'>]
```

id	*username*	*email*
1	admin	admin@example.com
2	peter	peter@example.org
3	guest	guest@example.com