Flask IV: Database

Harry J. Wang, Ph.D.

University of Delaware

Relational Databases

- What you learned in MISY330
- Relational Database Systems: SQLite, MySQL, PostgreSQL, Oracle, SQL Server
- We use SQLite in this course, which is a popular open source SQL database that stores an entire database in a single file.
- Self-study Learn SQL at https://www.codecademy.com/learn/learn-sql if necessary
- Basic concepts: tables, primary key, foreign key, SQL query, joins, etc.
- Basic CRUD operations: Create, Read, Update, and Delete

ORM (Object Relational Mapper)

- ORM provides a database abstraction layer that allows you to work at a higher level with regular objects instead of database entities such as tables and SQL query languages.
- Compared with database engines, ORM packages are:
  - better in ease of use, e.g., just python code, no need to write SQL queries
  - better in portability, e.g., you can change from SQLite to MySQL with one line without change the SQL statements
  - slower in performance
- SQLAlchemy is a popular Python ORM package we use in this class with flask extension: Flask-SQLAlchemy (http://flask-sqlalchemy.pocoo.org/2.3/)

Setup Flask-SQLAlchemy

```
Flask==0.12.2
Flask-Script==2.0.6
Flask-SQLAlchemy==2.3.0
```

1. Install Flask-SQLAlchemy package by adding it to requirements.txt file and run `pip install –r requirements.txt`
2. Import SQLAlchemy

```
from flask_sqlalchemy import SQLAlchemy
```

3. Setup database (local SQLite file in the same folder, remember to import os library)

```
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'data.sqlite')
db = SQLAlchemy(app)
```

4. Database model definition (more details later)
5. Initialize the database (using script command in manage.py)

```
# reset the database and create two artists
@manager.command
def deploy():
    db.drop_all()
    db.create_all()
```

Table in ORM

- Each table is a Class

```
class Artist(db.Model):
    __tablename__ = 'artists'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64))
    about = db.Column(db.Text)
```

```
class Song(db.Model):
    __tablename__ = 'songs'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(256))
    year = db.Column(db.Integer)
    lyrics = db.Column(db.Text)
```

Insert Data in ORM

- One record/row is an Object (you can use dot to access the value, such as artist.name)

```
coldplay = Artist(name='Coldplay', about='Coldplay is a British rock band.')
maroon5 = Artist(name='Maroon 5', about='Maroon 5 is an American pop rock band.')
```

- Insert records (always commit for changes to take effect):

```
db.session.add(coldplay)
db.session.add(maroon5)
db.session.commit()
```

http://flask-sqlalchemy.pocoo.org/2.3/queries

Queries in ORM

- Select all using .all():

```
artists = Artist.query.all()
```

- Select based on conditions using .filter_by() and Select one using .first():

```
one_artist = Artist.query.filter_by(id=2).first()
artists = Artist.query.filter_by(name='Maroon 5').all()
```

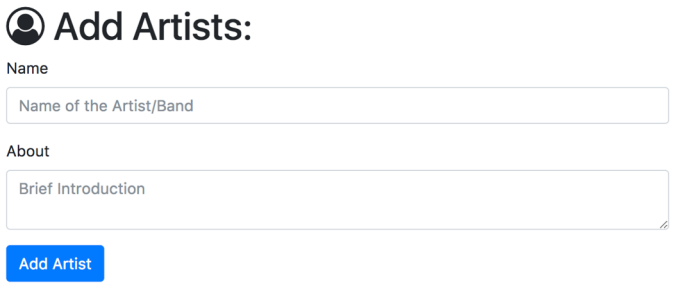
Update records in ORM

1. Get the record you want to update
2. Update the values
3. Commit the changes to the database

```
# get the record
one_artist = Artist.query.filter_by(name='Maroon 5').first()
# update the record
artist.about = 'Adam Levine is the lead singer.'
# commit the changes to the database
db.session.commit()
```

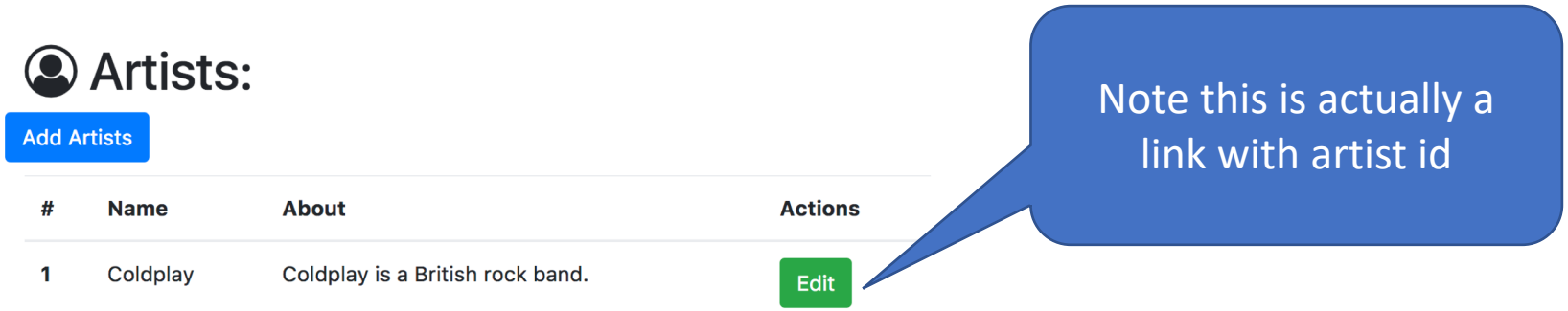
Exercise

- Build a database with one table for artists
- Create pages to:
  - show all artists
  - Add artists

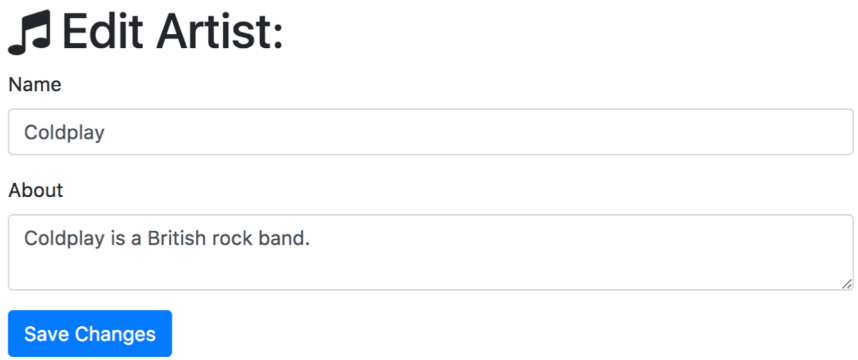


After-class Exercise

- Look at the sample code and see how Edit page is created




```
<a href="/artist/edit/{{artist.id}}" class="btn btn-success" role="button">Edit</a>
```



Relationship in ORM

- One to Many relationship (one artist can have many songs)

```python
class Artist(db.Model):
    __tablename__ = 'artists'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64))
    about = db.Column(db.Text)
```

```python
class Song(db.Model):
    __tablename__ = 'songs'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(256))
    year = db.Column(db.Integer)
    lyrics = db.Column(db.Text)
```

```python
songs = db.relationship('Song', backref='artist')
```

```python
artist_id = db.Column(db.Integer, db.ForeignKey('artists.id'))
```

- Using a Foreign key

```python
coldplay = Artist(name='Coldplay', about='Coldplay is a British rock band.')
maroon5 = Artist(name='Maroon 5', about='Maroon 5 is an American pop rock band.')
song1 = Song(name='Yellow', year=2000, lyrics="Look at the stars", artist=coldplay)
```

Exercise

- Create the Add Song page
- Note: the artist has to be in the database before a song can be added

Delete Records in ORM

1. Get the record you want to delete
2. Delete the record using the session
3. Commit the changes to the database

```
@app.route('/song/delete/<int:id>', methods=['GET', 'POST'])
def delete_song(id):
    song = Song.query.filter_by(id=id).first()
    artists = Artist.query.all()
    if request.method == 'GET':
        return render_template('song-delete.html', song=song, artists=artists)
    if request.method == 'POST':
        db.session.delete(song)
        db.session.commit()
        return redirect(url_for('show_all_songs'))
```

http://docs.sqlalchemy.org/en/latest/orm/cascades.html

Delete Cascade

- Delete cascade specifies how to handle “child” objects when the corresponding “parent” object is deleted, e.g.,
  - For one-to-many relationship between Artist and Song: when an artist is deleted from the database, how to handle his/her songs in the database.
- The default behavior is de-associating “child” objects from its “parent” object by setting their foreign key references to NULL.
- We can change the default behavior to “strongly enforce” the referential integrity by setting cascade on the “parent” side to delete all “child” objects:

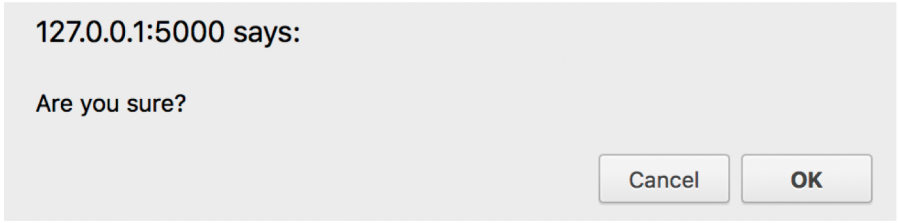
```
class Artist(db.Model):
    __tablename__ = 'artists'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64))
    about = db.Column(db.Text)
    songs = db.relationship('Song', backref='artist', cascade="delete")
```

Confirmation Window

- We can use jQuery to pop up a confirmation window for the deletion button

```
<button type="submit" class="btn btn-danger" id="delete_btn">Delete</button>
```

```
{% block scripts%}
  {{super()}}
  <script>
  $(document).ready(function () {
    $('#delete_btn').click(function () {
      if (! confirm('Are you sure?')) {
        return false;
      }
    })
  });
  </script>
{% endblock%}
```

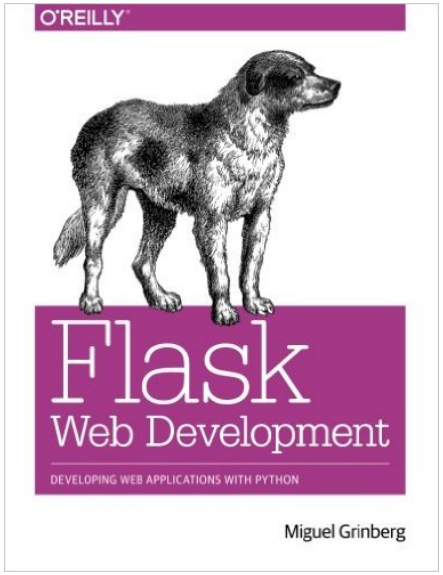


In-class Exercise

- Develop a delete page for artist
- Use `git checkout v4` to see the delete code without ajax

Resources

- Flask Web Development: Developing Web Applications with Python by Miguel Grinberg (1st Edition)
- Chapter 5



Use `git checkout v3.3`
to see the code

- Sample Course Application: SongBase: https://github.com/udmis/songbase

Interact with DB using Shell (optional)

- Shell can be used to manipulate database (useful for testing and debugging)

```
$ python manage.py shell
>>> from songbase import db
>>> db.drop_all()
>>> db.create_all()
>>> from songbase import Artist, Song
>>> cp = Artist.query.filter_by(id=1).first()
>>> coldplay = Artist(name='Coldplay', about='Coldplay is a British rock band.')
>>> song1 = Song(name='Yellow', year=2000, lyrics="Look at the stars", artist=coldplay)
>>> db.session.add(coldplay)
>>> db.session.add(song1)
>>> db.session.commit()
>>> cp = Artist.query.filter_by(id=1).first()
>>> cp.name
u'Coldplay'
>>> cp.about
u'Coldplay is a British rock band.'
>>> cp.songs
[<Song 1>]
>>> for song in cp.songs:
...     print song.name
...
Yellow
```