Flask-SQLAlchemy + Flask-Migrate Demo

Joshua Weiner <jweiner@stuy.edu> May 20, 2019 12:47 PM
Posted in group: **SoftDev**

Hey thinkers,

In case you were thinking of using one or both of these libraries for the final project, this should serve as a comprehensive post about how to use both and the basic functionality that you should know.

If you're wondering what these libraries are / what they do, please reference this QAF post.

So, let's get started with this quick demo!

Using pip, install **flask-sqlalchemy** and **flask-migrate**.

Second, let us create a schema file for our database. Instead of using database-builder scripts like we have in the past, we can create a **Class File** that SQLAlchemy will interpret into a database structure.

models.py

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

As you can see, this file creates two classes, a User and a Post, with their own attributes and methods. Examining what this class file will do: in the database SQLAlchemy will create a tables for each class, with each attribute entered into the table as a column. As you can see, you can specify the type and size each column should hold. The methods will enable us to interact with queried database entries as objects, and we can access each attribute of database entries as if they are objects. This will make interaction with database objects very simple.

Now, we need to create a configuration file for our database to be housed in the main directory of our project. This will allow the app to create and access the database with the given file location (the folder of config.py).
config.py

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Before we proceed, let us make the following change to our __init__.py file, in order for us to use these libraries effectively.
__init__.py

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

These changes will allow us to interact with the libraries in our virtual environment from python scripts, and set up the necessary reference to access the database. Furthermore, this references the configuration file we just created to access the database.

After we have done this, we can run the following commands within our virtual environment:

```
(venv)$flask db init
(venv)$flask db migrate -m "creating first database migration"
(venv)$flask db upgrade
```

There's a lot going on here, so let's break it down.
$flask db init will create a migrations folder in your repository. **What is a migration folder?** Flask-migrate essentially introduces a workflow into your database transformations via your class file. Essentially, every time you make a change to the database schema, the library generates a python script detailing what those changes are (tables created, columns added or dropped, etc.) Every change is stored as a script, which can be upgraded or downgraded, should you want to incorporate the new changes or get rid of them if they don't work, all without corrupting the data stored within them.
$flask db migrate will create that migration script, and in our case it will look like the following:
app/migrations/versions/<random sequence>.py

```
"""first

Revision ID: 47cb944bdab7
Revises:
Create Date: 2019-05-20 12:20:13.691316

"""
from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = '47cb944bdab7'
down_revision = None
branch_labels = None
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('user',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('username', sa.String(length=64), nullable=True),
    sa.Column('email', sa.String(length=120), nullable=True),
    sa.Column('password_hash', sa.String(length=128), nullable=True),
    sa.PrimaryKeyConstraint('id')
    )
    op.create_index(op.f('ix_user_email'), 'user', ['email'], unique=True)
    op.create_index(op.f('ix_user_username'), 'user', ['username'], unique=True)
    op.create_table('post',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('body', sa.String(length=140), nullable=True),
    sa.Column('user_id', sa.Integer(), nullable=True),
    sa.ForeignKeyConstraint(['user_id'], ['user.id'], ),
    sa.PrimaryKeyConstraint('id')
    )
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_table('post')
    op.drop_index(op.f('ix_user_username'), table_name='user')
    op.drop_index(op.f('ix_user_email'), table_name='user')
    op.drop_table('user')
    # ### end Alembic commands ###
```

If you compare these to what I told you before about database migrations, this should make sense.

$flask db upgrade is the final piece of the puzzle, this runs the upgrade() function from the generated script and will incorporate these changes into our database – if it doesn't exist already it will create one, or if it does it will add the changes without editing any user data. We could just as easily run **$flask db downgrade** afterwards, getting rid of our changes, but again, not getting rid of existing data in the database.

So, how can we interact with database data?

First, let's modify whatever file we're using to have app routes in our flask app with the following lines:

```
from app.models import User, Post
from functools import wraps
from sqlalchemy import func
import uuid

route_code = str(uuid.uuid4())
# print(route_code)

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User}
```

This will allow us to interact with the database from our python scripts.

Now, let's learn how to add database entries from this script. If we're to add a user object, for example, we write the following line:

```
user = User(username=<username>, email=<email>)
db.session.add(user)
db.session.commit()
```

This will create a new User object (think constructors), then add it to the session, then commit it to the database. Just like that, we've created a user!

If we want to query database objects, we can run the following lines:

```

Users = User.query.all()

or


u = User.query.filter_by(username="MrBrown").first()
```

As you can see, from the first line we query all of the User objects. From the second, we query the User objects with the given condition, that their username is "MrBrown". Note also, that we will return a Query Object and not the User Object unless we specify **first()** or **all()**: the former giving us the first object (or none) that satisfies the conditions, the latter giving us a list of all of the queried objects.

We can access and modify user attributes in the following way:

```
u = User.query.filter_by(username=“MrBrown”).first()
print(u.email)
u.username = "new_username"
db.session.add(u)
db.session.commit()
```

While you don't need to add/commit anything simply after accessing user information, you do need to do this after modifying attributes. But, it should be recognized how much simpler this is over our usual SQL operations. Noice!

Finally, you may have noticed earlier some interesting lines in our classes file. I am, of course, referring to:

```
In User:
    posts = db.relationship('Post', backref='author', lazy='dynamic')

In Post:
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
```

These lines establish **chained tables**, essentially allowing us to reference table entries from other objects. It's super useful, and as you can guess, we can add many posts to each user without having to store them as comma-separated strings.
Let's look at an example

```
u = User.query.filter_by(username=“MrBrown”).first()
p = Post(body='my first post!', author=u)

db.session.add(p)
db.session.commit()
```

Notice here how the back-reference we specified earlier is used as a reference to the user object.

We can do this many times over, and it allows us to do the following:
Query posts on their own:

```
posts = Post.query.all()
for p in posts:
     print(p.id, p.author.username, p.body)
```

Or access a user's posts:

```
u = User.query.filter_by(username="MrBrown").first()
print(u.posts.all())
```

Ok folks, this looks to be about all for a Flask-SQLAlchemy and Flask-Migrate. Please follow up with any more insights you have or oversights that I might have made.

Hope this helps someone, or at least increases your learnination!

KJU