Tutorial Week 12

Embedded SQL in Flask

In today's lab, you will be writing SQL queries inside Flask to communicate with an external SQLite database. This type of SQL is called **embedded SQL**

For more reference for writing embedded SQL in Flask, consult the Flask documentation. In tutorial, we will be going over the "vanilla" way of using SQLite with Flask using only the sqlite3 standard Python library.

In lecture, Purva uses the 3rd party library SQLAlchemy but for your assignment, either strategy will be fine

The basic setup for a Flask app is similar to what we did in Week 6 with a couple extra functions to communicate with the database. The starter code for today's lab has everything pre-written but we will go over the process if you are starting from scratch. This tutorial assumes you have the starter app.py code from Week 6 lab

Setting up the database:

Note the required imports (there are more imports than in Week 6's lab)

```
import sqlite3
from flask import Flask, render_template, request, g


# the database file we are going to communicate with
DATABASE = './database.db'


# connects to the database
def get_db():
    # if there is a database, use it
    db = getattr(g, '_database', None)
    if db is None:
        # otherwise, create a database to use
        db = g._database = sqlite3.connect(DATABASE)
    return db
```

```
# converts the tuples from get_db() into dictionaries
# (don't worry if you don't understand this code)
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

# given a query, executes and returns the result
# (don't worry if you don't understand this code)
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

Whenever we want to query information from the database, we need to setup a connection with the database which is done using the function get_db(). This function returns a database object of which we can apply queries to

There are two helper functions make_dicts() and query_db(). You do not need to know exactly how to write these two functions but they are very useful for querying data:

1. make_dicts()
    - By default, Flask returns database data as tuples which are an inconvenient data structure to use. This function utilies the Factory design pattern to generate dictionaries for all the tuples normally returned by Flask
        - This function is technically not necessary but will make your life infinitely easier in the long run
2. query_db()
    - Given a query in the form of a string, executes and returns the result of the query
        - In conjunction with make_dicts(), this return will be in the form of a dictionary
        - query_db() uses a database construct called a **cursor** to select the data based upon the query and then packages the result to finally return. Cursors are outside of the scope of this course so if this is confusing you don't need to know this in great detail

Database teardown:

Whenever you close the Flask application, you need to close any open connections you have to the database. Otherwise, the next time you access the database, you may be blocked by a connection that was still left open. To do this, we create a function that will be called on application teardown that will destroy any currently open connections

```
# this function must come after the instantiation of the variable app
# (i.e. this comes after the line app = Flask(__name__))
@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        # close the database if we are connected to it
        db.close()
```

Querying data:

Querying data follows the same general structure:

1. Create a database instance using get_db()
    - Apply make_dicts to convert the tuples to dictionaries
2. Get any request parameters (if there are any)
3. Call and execute the database query while storing it somewhere
4. Close the database
5. Return the results

Generally, you want to do this wherever you use an @app.route as your routing functions will generally have some sort of functionality tied to them. As an example, this is what it would look like if you were to do something in the default route (i.e. @app.route('/'))

```
@app.route('/')
def root():
    # get the database instance
    db = get_db()
    # convert tuples to dictionaries
    db.row_factory = make_dicts

    # query for the items
    # (in this case, we do not have any request parameters)
    items = []
```

```
    for item in query_db('select * from items'):
        items.append(item)
    db.close()

    return render_template('index.html', items=items)
```

Querying with request parameters:

```
@app.route('/')
def root():
    # get the database instance
    db = get_db()
    # convert tuples to dictionaries
    db.row_factory = make_dicts

    # get request parameter
    # (in this case, the request parameter is called "name")
    name = request.args.get('name')

    # query for the item by name
    # because we are expecting this to return a single value, we set one=True
    # (read query_db() for more details)
    item = query_db('select * from items where name = ?', [name], one=True)
        items.append(item)
    db.close()

    return render_template('index.html', items=items)
```

Two important notes about the above code:

1. After every query, when we are completely done with the connection, you **must** close the connection
    - If you do not close the connection, the next time you access the database, you may get locked out
2. We replace the request parameter using a ? instead of something like `query_db('select * from items where name = {name}'.format(name=name))`
    - Using the `.format()` method to replace parameters leaves the code vulnerable to SQL injection

        - A malicious user could write SQL code instead of a name in the text box which could cause very bad things to happen
        - Using the ? sanitizes the input and makes it safe to execute

Exercise

NOTE: you must do this exercise on your own computer; you will not be able to finish this lab on the lab computers

Today we are making an employee database application for the company Dunder Mifflin from the completely real documentary *The Office*. The application behaves as follows:

Starter code is given here

- Default route (i.e. /):
    - Display a table of all the employees
        - This can be done with a `SELECT * FROM ...` query
- Employee route (i.e. /employee):
    - Display all the information of the selected employee
        - The value the user inputs will be equal to the column `firstname` in the database
        - This can be accomplished with a `SELECT * FROM employees WHERE firstname = ...` query

If you need help during the lab, please ask. If you would like to investigate the schema use the SQLite browser from earlier in the semester

References

- Using SQLite with Flask
- SQLite browser