


Working with SQLite databases in Flask with Flask-SQLAlchemy

COMP0034 2023-24 Week 2 coding activities

1. Preparation and introduction

This assumes you have already forked the coursework repository and cloned the resulting repository to your IDE.

1. Create and activate a virtual environment
2. Install the requirements  `pip install -r requirements.txt`
3. Run the app `flask --app paralympics run --debug`
4. Open a browser and go to <http://127.0.0.1:5000>
5. Try it again with <http://127.0.0.1:5000/name> (replace name with your name)
6. You should see the variable route for the homepage (the final activity from last week)
7. Stop the app using `CTRL+C`

Consider installing the VS Code extension SQLite Viewer to allow you to view the content of a database through the VS Code interface.

If you are using PyCharm Professional then you can already view database files. You cannot do this in PyCharm Community edition which is why Professional is recommended.

When you install from requirements.txt this included [Flask-SQLAlchemy](#). The SQLAlchemy package will also be installed as it is a dependency for Flask-SQLAlchemy. Together they provide functionality that lets you more easily create Python classes that map to database tables; and handles the database interaction, i.e. SQL queries, using Python functions. This follows a design pattern called ORM, Object Relational Mapper. An ORM encapsulates, or wraps, data stored in a database into an object that can be used in Python.

Flask-SQLAlchemy works with many database formats but will not work directly with .csv/.xlsx file. You will use a SQLite database which stores the tables and data in a single file which is convenient for the coursework.

Step 1: Change the Flask app to be created using the Flask application factory pattern

Create a function that allows you to create a Flask app and then enable that app to use extensions such as Flask-SQLAlchemy and to add configuration parameters.

This is an [application factory](#) pattern. Like a factory production line, you create the app, then you pass it along a production line adding extra 'features' to it as needed.

1. Open `paralympics/__init__.py`
2. The following is based on the `create_app()` function from the [Flask tutorial](#):

```
import os

from flask import Flask

def create_app(test_config=None):
    # create the Flask app
    app = Flask(__name__, instance_relative_config=True)
    # configure the Flask app (see later notes on how to generate your config)
    app.config.from_mapping(
        SECRET_KEY='dev',
        # Set the location of the database file called paralympics.sqlite
        SQLALCHEMY_DATABASE_URI= "sqlite:/// " + os.path.join(app.instance_path, 'paralympics.sqlite')
    )

    if test_config is None:
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile('config.py', silent=True)
    else:
        # load the test config if passed in
        app.config.from_mapping(test_config)

    # ensure the instance folder exists
    try:
        os.makedirs(app.instance_path)
```

```
except OSError:
    pass

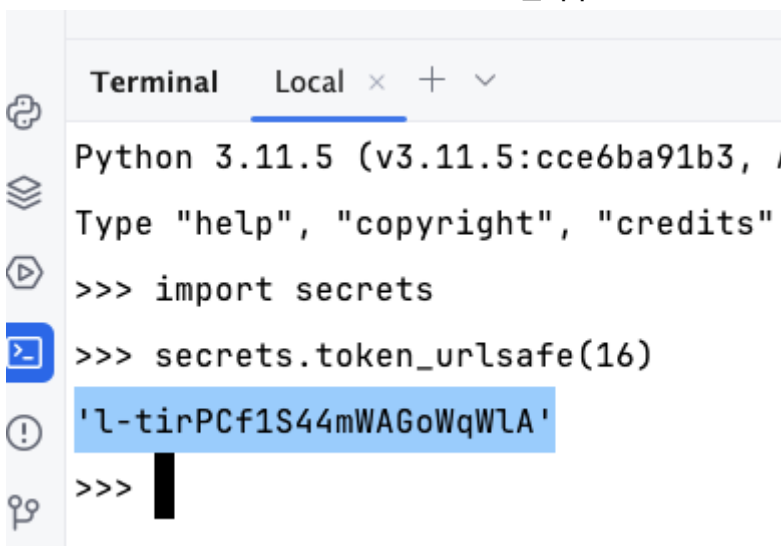
return app
```

3. Create your own unique SECRET_KEY .

SECRET_KEY is used by Flask and extensions to keep data safe. It's set to 'dev' to provide a convenient value during development, but it should be overridden with a random value when deploying.

SQLALCHEMY_DATABASE_URI is the path where the SQLite database file will be saved. It's under app.instance_path, which is the path that Flask has chosen for the instance folder.

You can generate a secret key from the Terminal command line. Type `python3` or `python` and press enter. At the `>>>` prompt type `import secrets` and press enter. Then type `secrets.token_urlsafe(16)` and press enter. You should see a string of 16 characters. Copy this and use it to replace the word 'dev' in the SECRET_KEY line in the `create_app()` function.

A screenshot of a terminal window with a light gray background. The title bar shows 'Terminal' and 'Local' with a close button. The terminal text shows a Python prompt where 'import secrets' and 'secrets.token_urlsafe(16)' are entered. The output is a 16-character string: 'l-tirPCf1S44mWAGoWqWLA'. The string is highlighted in blue. The prompt '>>>' is visible at the bottom.

```
Terminal Local x + v
Python 3.11.5 (v3.11.5:cce6ba91b3, )
Type "help", "copyright", "credits"
>>> import secrets
>>> secrets.token_urlsafe(16)
'l-tirPCf1S44mWAGoWqWLA'
>>>
```

4. Now that the app is created in the `create_ap()` function, you need to modify `paralympics.py` app to use this.

Use the Flask `current_app` object to access the configured app.

Replace the contents on `paralympics.py` with the following:

```
from flask import current_app as app

@app.route('/')
def hello():
```

```
return f"Hello!"
```

5. Return to the `create_app()` function and now let the app know about the routes that are defined in `paralympics.py`.

```
# Put the following code inside the create_app function after the code t
# This is likely to be circa line 40.
with app.app_context():
    # Register the routes with the app in the context
    from paralympics import paralympics
```

NB: Consider renaming `paralympics.py` to `routes.py` or `controllers.py` to avoid confusion between the `paralympics` package and the `paralympics` module within that package.

6. Check that you can run the app `flask --app paralympics run --debug`. Flask recognises the `create_app()` function.

Initialise the SQLAlchemy extension

Return to `__init__.py` and add the following code to *before* the `create_app()` function to initialise the SQLAlchemy object.

```
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import DeclarativeBase
```

```
class Base(DeclarativeBase):
    pass
```

```
db = SQLAlchemy(model_class=Base)
```

Define a model

Create a python file called `models.py`. This will contain classes that map to your database tables. Create a python file. This is often named `models.py` but doesn't have

to be.

The syntax for a class that maps to a database table is given in the [Flask-SQLAlchemy documentation](#) . The table is defined as follows:

- Define the class with an appropriate name.
- The tablename should match the tablename in the database.
- The column names should match the column names used in the database.
- The column datatypes should match the data types used in the database.
- The classes inherit the Flask-SQLAlchemy Model class. This automatically gives the class access to functions that will handle the constructor so you don't need to define it. You can access the instance of SQLAlchemy, called `db` , that you just created in `__init__.py` .

At some point the paralympics app will have authentication and so needs a table to hold user details. Add the following class to `models.py` :

```
from sqlalchemy import Integer, String
from sqlalchemy.orm import Mapped, mapped_column
from paralympics import db

class User(db.Model):
    id: Mapped[int] = mapped_column(db.Integer, primary_key=True)
    email: Mapped[str] = mapped_column(db.Text, unique=True, nullable=False)
    password: Mapped[str] = mapped_column(db.Text, unique=True, nullable=False)
```

Add the following code to create two classes that represents the tables in the database, Region and Event.

```
# Adapted from https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/quickstart
from typing import List
from sqlalchemy import Integer, String, ForeignKey
from sqlalchemy.orm import Mapped, mapped_column, relationship
from paralympics import db

# This uses the latest syntax for SQLAlchemy, older tutorials will show different
# SQLAlchemy provide an __init__ method for each model, so you do not need to
class Region(db.Model):
```

```

__tablename__ = "region"
NOC: Mapped[str] = mapped_column(db.Text, primary_key=True)
region: Mapped[str] = mapped_column(db.Text, nullable=False)
notes: Mapped[str] = mapped_column(db.Text, nullable=True)
# one-to-many relationship with Event, the relationship in Event is call
# https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html#one-to-
events: Mapped[List["Event"]] = relationship(back_populates="region")

```

```

class Event(db.Model):
    __tablename__ = "event"
    id: Mapped[int] = mapped_column(db.Integer, primary_key=True)
    type: Mapped[str] = mapped_column(db.Text, nullable=False)
    year: Mapped[int] = mapped_column(db.Integer, nullable=False)
    country: Mapped[str] = mapped_column(db.Text, nullable=False)
    host: Mapped[str] = mapped_column(db.Text, nullable=False)
    NOC: Mapped[str] = mapped_column(ForeignKey("region.NOC"))
    # add relationship to the parent table, Region, which has a relationship
    region: Mapped["Region"] = relationship("Region", back_populates="events")
    start: Mapped[str] = mapped_column(db.Text, nullable=True)
    end: Mapped[str] = mapped_column(db.Text, nullable=True)
    duration: Mapped[int] = mapped_column(db.Integer, nullable=True)
    disabilities_included: Mapped[str] = mapped_column(db.Text, nullable=True)
    countries: Mapped[str] = mapped_column(db.Text, nullable=True)
    events: Mapped[int] = mapped_column(db.Integer, nullable=True)
    athletes: Mapped[int] = mapped_column(db.Integer, nullable=True)
    sports: Mapped[int] = mapped_column(db.Integer, nullable=True)
    participants_m: Mapped[int] = mapped_column(db.Integer, nullable=True)
    participants_f: Mapped[int] = mapped_column(db.Integer, nullable=True)
    participants: Mapped[int] = mapped_column(db.Integer, nullable=True)
    highlights: Mapped[str] = mapped_column(db.Text, nullable=True)

```

```

class User(db.Model):
    id: Mapped[int] = mapped_column(db.Integer, primary_key=True)
    email: Mapped[str] = mapped_column(db.Text, unique=True, nullable=False)
    password: Mapped[str] = mapped_column(db.Text, unique=True, nullable=False)

    def __init__(self, email: str, password: str):
        """
        Create a new User object using hashing the plain text password.
        :type password_string: str
        :type email: str
        :returns None
        """
        self.email = email

```

```
self.password = password
```

The relationship between the two tables is defined using the primary and foreign keys with the `relationship` function as follows:

```
from typing import List
from sqlalchemy import ForeignKey
from sqlalchemy.orm import Mapped, mapped_column, relationship
from paralympics import db

# non-Key/relationship column details have been omitted from the classes below
# one-to-many relationship from Region to Event
# https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html#one-to-many

class Region(db.Model):
    __tablename__ = "region"
    # Primary key attribute
    NOC: Mapped[str] = mapped_column(db.Text, primary_key=True)
    # Add a relationship to Event. The Region then has a record of the Event
    # This references the relationship 'region' in the Event table.
    events: Mapped[List["Event"]] = relationship(back_populates="region")

class Event(db.Model):
    __tablename__ = "event"
    # add ForeignKey that maps to the primary key of the Region table
    NOC: Mapped[str] = mapped_column(ForeignKey("region.NOC"))
    # add relationship to Region, this references the relationship 'events'
    region: Mapped["Region"] = relationship(back_populates="events")
```

Update the `create_app()` function to generate the database tables

Add a line of code to the `init.py` in the `paralympic_app` package to import the models. To avoid circular imports, put this after the app is created; so NOT at the top of the file where you would usually place imports.

If you are using a linter you will need to ignore the warnings about placing the import at the top of the file.

To create the tables for User, Region and Event in the database use a Flask-SQLAlchemy function `db.create_all()`. This will create the tables if they do not already exist. Add this line *after* importing the models.

```
def create_app(test_config=None):
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_mapping(
        SECRET_KEY='l-tirPCf1S44mWAGoWqWlA',
        SQLALCHEMY_DATABASE_URI="sqlite://" + os.path.join(app.instance_path, 'db.sqlite')
    )

    if test_config is None:
        app.config.from_pyfile('config.py', silent=True)
    else:
        app.config.from_mapping(test_config)

    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    # Initialise Flask with the SQLAlchemy database extension
    db.init_app(app)

    # Models are defined in the models module, so you must import them before
    # will not know about them.
    from paralympics.models import User, Region, Event
    # Create the tables in the database
    # create_all does not update tables if they are already in the database.
    with app.app_context():
        db.create_all()

    # Register the routes with the app in the context
    from paralympics import paralympics

    return app
```


Run the app to generate the database

Run the app `flask --app paralympics run --debug .`

As the database does not exist it will be created. You can check this by looking in the instance folder. You should see a file called `paralympics.sqlite`.

Add data to the database

There are many ways to add data to a database using Python.

This method assumes you created database as above and are then going to add the data from the .csv files to the existing tables using SQLAlchemy. The code will be called every time the app runs.

1. Add the following code to the end of `__init__.py` :

```
import csv
from pathlib import Path

def add_data_from_csv():
    """Adds data to the database if it does not already exist."""

    # Add import here and not at the top of the file to avoid circular import
    from paralympics.models import Region, Event

    # If there are no regions in the database, then add them
    first_region = db.session.execute(db.select(Region)).first()
    if not first_region:
        print("Start adding region data to the database")
        noc_file = Path(__file__).parent.parent.joinpath("data", "noc_regions.csv")
        with open(noc_file, 'r') as file:
            csv_reader = csv.reader(file)
            next(csv_reader) # Skip header row
            for row in csv_reader:
                # row[0] is the first column, row[1] is the second column
                r = Region(NOC=row[0], region=row[1], notes=row[2])
                db.session.add(r)
            db.session.commit()
```

```

# If there are no Events, then add them
first_event = db.session.execute(db.select(Event)).first()
if not first_event:
    print("Start adding event data to the database")
    event_file = Path(__file__).parent.parent.joinpath("data", "paralymp
with open(event_file, 'r') as file:
    csv_reader = csv.reader(file)
    next(csv_reader) # Skip header row
    for row in csv_reader:
        # row[0] is the first column, row[1] is the second column et
        e = Event(type=row[0],
                    year=row[1],
                    country=row[2],
                    host=row[3],
                    NOC=row[4],
                    start=row[5],
                    end=row[6],
                    duration=row[7],
                    disabilities_included=row[8],
                    countries=row[9],
                    events=row[10],
                    sports=row[11],
                    participants_m=row[12],
                    participants_f=row[13],
                    participants=row[14],
                    highlights=row[15])
        db.session.add(e)
    db.session.commit()

```

2. Update the `create_app()` function to call the `add_data_from_csv()` function after the tables are created

```

def create_app(test_config=None):
    # ... CODE OMITTED FOR BREVITY HERE ...

    with app.app_context():
        # Create the database and tables if they don't already exist
        db.create_all()
        # Add the data to the database if not already added
        add_data_from_csv()

    # ... CODE OMITTED FOR BREVITY HERE ...

```

```
return app
```

Another approach would be to create the database using Python code as a one-off action. This may be preferable if there is a lot of data to load and the code takes a while to execute.

The file `data\create_db_add_data` contains an example of this approach. You do not need this for this activity, it is included in case you want to take this approach for your coursework.

These are not the only options; you will find blog posts and tutorials that offer other approaches that you could use instead.

Reading

There are many aspects not covered in this tutorial that you could investigate.

- Track database changes: If you change a model's columns, use a migration library like Alembic with [Flask-Alembic](#) or [Flask-Migrate](#) to generate migrations that update the database schema.
- Alternative Python class definitions using [Python Dataclasses](#) with [SQLAlchemy](#) `MappedAsDataclass`)
- [Reflecting tables](#) can be used if you have a database with the data already in.
- [Python sqlite3 tutorial](#) may be useful if you create the database and add data separately from the Flask application code.