Introduction

Welcome to Module 6! This week, we will learn about creating middleware that is powered by a datastore. The datastore used is a relational database named PostgreSQL. Relational databases are scalable and allow you to implement world systems. PostgreSQL databases are organized into tables that hold records.

Topics and Learning Objectives

Topics

- PostgreSQL, PGAdmin 4
- Domain modeling
- Working with an ORM – SQLAlchemy
- Datastore, database
- SQL: select, update, insert, delete

Learning Objectives

By the end of the module, you should be able to:

1. Explain the concept of ORM (Object-Relationship Mapping) using SQLAlchemy.
2. Store and retrieve data from a PostgreSQL database.
3. Create SQL queries indirectly by using SQLAlchemy.

Readings & Resources

Reading

Required

- Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media, Inc.
    - Chapter 5: Databases
    - Chapter 12: Followers
- PostgreSQL. (2022). About PostgreSQL.

SQLAlchemy

- SQL is the most common way to access relational databases. It can be difficult to work with due to the learning curve.
- SQLAlchemy is a Python ORM (Object Relationship Mapper) that interacts with relational databases. ORM is used as a cache for translating from a relational database model to an object-oriented model.
- Relational databases have two distinct advantages over all database engines or types. First, they treat data naturally or organize it according to normal forms. Second, they offer performance that is unmatched by any other database engine or type, hence they are used for high performance, high scalable applications.
- SQLAlchemy should be used with Python 2.7 and higher. This should not be an issue since Python 3 is used for new development, but can be an issue if you have to maintain older Python applications.
- SQLAlchemy is a Python module. Hence, it can be installed using the pip command.
- SQLAlchemy is designed to operate with relational databases and is a generic ORM. SQLAlchemy connects to individual database using a database driver.
- The following databases are supported by SQLAlchemy: Firebird, MS SQL Server, Oracle, PostgreSQL, SQLite and Sybase.

SQLAlchemy and ORM

- SQLAlchemy ORM capabilities allows classes to be mapped to database tables. More specific instances of classes represent rows within database tables.
- SQLAlchemy provides a second level cache for database engines. The second level cache is represented by class instances from memory.
- ORM is a programming technique for converting data between incompatible systems in object-oriented programming languages.
- The advantage that ORM provides is that you use Python to manipulate class instances that are synchronized with a relational database. This allows you to use Python instead of SQL. Furthermore, this provides a faster mechanism for execution since ORM exists in memory.
- The primary unit of storage for a relational database is file(s). Files are much slower than memory.

Installing SQLAlchemy

- To install SQLAlchemy use the following command:

- pip install sqlalchemy

- To verify that SQLAlchemy is install run the following two lines in interactive mode:
  - import sqlalchemy
  - sqlalchemy.__version__

Create_engine()

- The create_engine() method allows a Python program to connect to a relational database. This method may be executed directly by the application developer or indirectly which means that it is executed by the ORM framework. When the developer creates an instance of SQLAlchemy, then create_engine() method is executed by the framework, that is SQLAlchemy.
- The code below connects to an SQLite database. Create_engine() method requires a URI that points to a single database instance. Note: create_engine uses SQLite as the database engine.
  - from sqlalchemy import create_engine
  - engine = create_engine(‘sqlite:///test.db’, echo=True)
- SQLite is an in memory database. It is very small and easy to work with for development purposes. For production purposes a high-performance database is required. High-performance databases are represented by Oracle and PostgreSQL. These databases also allow the the creation of world systems.

Create_engine() Methods

- Each database engine has its own SQL dialect. These may be relational database engines but they each use their own unique SQL dialect. This is why each database engine requires a database driver to connect to.
- The URI for any database engine has the following formula:

  Dialect[+driver]://user:password@host:port/dbname
- The create_engine() function returns an Engine Object. There are several methods that are supported by the Engine Object:
  - **Connect() –** Returns the connection object.
  - **Execute() –** Executes a SQL statement construct.
  - **Begin() –** Returns a context manager that supports the execution of a transaction. If the Python code after begin() successfully executes then the transaction is committed. If an error is encountered the transaction is rolled back.
  - **Dispose() –** Disposes the connection pool used by the Engine object.
  - **Driver() –** Returns the driver name of the dialect in use by the Engine object.

  - **Table_names() –** Returns a list of all table names available in the database.
  - **Transaction() –** Executes a function as a transaction.

- Note: many of the above methods are called by the ORM framework. You need to work with the ORM framework and this abstracts the unique behaviour of each database. This also allows for your code to be portable. When you move from one database to another all you have to do is change your database connection URL.

Your First SQLAlchemy Program

- Here is a Python program that uses SQLAlchemy to connect to a PostgreSQL database to insert a row in a table.
- The database is similar in design to the MongoDB database with differences.
- Note that the database table is named “User”. The word “user” is a reserved word in relational databases. Hence, it is necessary to use “User” in order to avoid conflict with the “user” reserved word.
- In the example below, we use SQLAlchemy to manage the domain management between the User table in PostgreSQL and users object in the Python code.

```
from flask import Flask, request, jsonify
from dataclasses import dataclass
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:torontomet123@localhost/CKCS145'
db = SQLAlchemy(app)

@dataclass
class User(db.Model):
     __tablename__ = 'User'

     email: str
     name: str

     email = db.Column( db.String(), primary_key=True )
     name = db.Column( db.String(100) )

@app.route('/list')
def list_all() :
     all_users = User.query.all()
     print( all_users )
     return all_users

@app.route('/insert', methods=['POST'])
def insert_user():

     name = request.form.get('user_name')
     email = request.form.get('user_email')

     u1 = User(email=email, name=name)

     db.session.add(u1)

     db.session.commit()
     db.session.flush()

     status_message = 'row with primary key of ' + email + ' has
been inserted'

     return jsonify( {'status': status_message } )

# should always be at the end of your file
if __name__ == '__main__' :
  app.run()
```

SQL and Creating a Database

- SQL (Standard Query Language) is a standard language for inserting, updating and deleting data from a relational database.
- The previous example will only work if a database exists and it has a table name “User”. This can be created using SQL.
- To create a database we need to issue a create database statement. This is exhibited below.

```
CREATE DATABASE "CKCS145"
         WITH
         OWNER = postgres
         ENCODING = 'UTF8'
         LC_COLLATE = 'en_CA.UTF-8'
         LC_CTYPE = 'en_CA.UTF-8'
         TABLESPACE = pg_default
```

```
          CONNECTION LIMIT = -1
          IS_TEMPLATE = False;
```

- To create a table we need to issue a create table statement. This is exhibited below.

```
CREATE TABLE IF NOT EXISTS public."User"
(
     email text COLLATE pg_catalog."default" NOT NULL,
     name text COLLATE pg_catalog."default",
     CONSTRAINT "User_pkey" PRIMARY KEY (email)
)
```

- The above two SQL statements may be intimidating at first. For beginners, it is advisable to use PGAdmin 4 to create databases and their tables. PGAdmin 4 is a GUI application that creates the corresponding SQL code based on your actions.

SQL and Inserting Data into a Database

- Inserting data into a database requires the creation of a database and its corresponding table.

- Once the corresponding table has been created, then data may be inserted using an insert SQL statement. This is exhibited below. The insert command below inserts a single record into table “User”. The record has two fields: email and name.

```
INSERT INTO public."User"(email, name)
     VALUES ('is@fourseasons.ca', 'Isadore Sharp');
```

- The above SQL statement looks intimidating at first. For beginners, it is advisable to use PGAdmin 4 to insert data into tables. PGAdmin 4 is a GUI application that creates the corresponding SQL code based on your actions.

Video

Please watch the following video, Introduction to PostgreSQL and PgAdmin4. This video demonstrates how to work with a PostgreSQL database.

Introduction to PostgreSQL and PgAdmin4

TMU Video

Retrieving From a Database

- Below is a Python program that uses SQLAlchemy to connect to a PostgreSQL database and return all rows from the “User” table.
- In the example below, we use SQLAlchemy to manage the domain management between the User table in PostgreSQL and users object in the Python code.
- The method list_all_users() uses the “User” domain class to query all records from the “User” table. This is done using the query.all() method. This method operates on the SQLAlchemy ORM, which in turn retrieves the data from the PostgreSQL database.

```
from flask import Flask, request, jsonify
from dataclasses import dataclass
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:torontomet123@localhost/CKCS145'
db = SQLAlchemy(app)

@dataclass
class User(db.Model):
     __tablename__ = 'User'
     email: str
     name: str

     email = db.Column( db.String(), primary_key=True )
     name = db.Column( db.String(100) )

@app.route('/list')
def list_all_users() :
     all_users = User.query.all()
     print( all_users )
     return all_users

# should always be at the end of your file
if __name__ == '__main__' :
  app.run()
```

Inserting Table Rows

- Below is a Python program that uses SQLAlchemy to connect to a PostgreSQL database and inserts a record into the “User” table.
- In the example below, we use SQLAlchemy to manage the domain management between the User table in PostgreSQL and users object in the Python code.
- The method insert_user() uses the “User” domain class to insert a record in in the “User” table. This is done using the db.session.add() method. This method operates on the SQLAlchemy ORM, which in turn inserts the data into the PostgreSQL database.

```
from flask import Flask, request, jsonify
from dataclasses import dataclass
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:torontomet123@localhost/CKCS145'
db = SQLAlchemy(app)

@dataclass
class User(db.Model):
     __tablename__ = 'User'

     email: str
     name: str

     email = db.Column( db.String(), primary_key=True )
     name = db.Column( db.String(100) )

@app.route('/insert', methods=['POST'])
def insert_user():

     name = request.form.get('user_name')
     email = request.form.get('user_email')

     u1 = User(email=email, name=name)

     db.session.add(u1)

     db.session.commit()
     db.session.flush()

     status_message = 'row with primary key of ' + email + ' has
been inserted'

     return jsonify( {'status': status_message } )

# should always be at the end of your file
if __name__ == '__main__' :
  app.run()
```

Updating Table Rows

- Below is a Python program that uses SQLAlchemy to connect to a PostgreSQL database and updates a record into the “User” table.
- In the example below, we use SQLAlchemy to manage the domain management between the User table in PostgreSQL and users object in the Python code.
- The method update_user() uses the “User” domain class to query for the record in question in the “User” table. This is done using the query.filter() method. This method operates on the SQLAlchemy ORM, which in turn retrieves the data from the PostgreSQL database.

```
from flask import Flask, request, jsonify
from dataclasses import dataclass
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:torontomet123@localhost/CKCS145'
db = SQLAlchemy(app)

@dataclass
class User(db.Model):
     __tablename__ = 'User'

     email: str
     name: str

     email = db.Column( db.String(), primary_key=True )
     name = db.Column( db.String(100) )

@app.route('/update', methods=['POST'])
def update_user():

     name = request.form.get('user_name')
     email = request.form.get('user_email')

     query = db.session.query(models.User)
     query = query.filter(models.User.email==email)
     rows_changed = query.update({models.User.name: name })

     print ('rows_changed : ', rows_changed)
     print('query :', query)

     db.session.commit()
     db.session.flush()

     status_message = str(rows_changed) + ' rows have been
affected/changed'

     return jsonify( {'status': status_message } )

# should always be at the end of your file
if __name__ == '__main__' :
  app.run()
```

SQLAlchemy ORM Flush and Commit

- In the previous examples, we see two methods commit() and flush() belonging to the ORM or SQLAlchemy.
- The commit method is responsible for executing appropriate create_engine() methods such as execute() and transaction().
- The flush method is responsible for sending the generated SQL code to the PostgreSQL database. Remember when you are operating on ORM or SQLAlchemy, you are manipulating objects in memory. The data needs to be synchronized with the database. This is done using the flush method.

Deleting in Relational Databases

It is good practice not to physically delete information in databases. Deletes should only be logical.

Physical deletes are not allowed, since historical information has been lost. Using logical deletes allows us to keep historical information.

Logical deletes can be implemented using an active flag. If the flag is set to false, then it can be treated as logically deleted.

Any inactive records can be moved to a historical database in order to keep current database with only active records. This is done in order to enable efficient searches or selects on tables in a database

Using PostgreSQL with SQLAlchemy

- The coding examples exhibited in this lecture are database independent, since they manipulate SQLAlchemy ORM. This means that the examples can work with any relational database engine.
- What would it take to make our examples work with PostgreSQL? The answer is simple; that is, the connection string needs to be changed. The connection string is the parameter for create_engine() method.
- The connection string below is for a PostgreSQL server that houses a ccps530 database:
  - “postgres://admin:password@localhost:5432/ccps530”

Why use PostgreSQL Over MongoDB

- PostgreSQL is a highly optimized database.
- PostgreSQL has a higher level of performance than MongoDB. This is a bit misleading, but you have to look at it in the following way.
  - Suppose you have a database with 5,000 records. MongoDB will outperform PostgreSQL. However, suppose that you have a database with 100,000 records. In such a situation, PostgreSQL will outperform MongoDB.

  - From the point of view of a database,100,000 may be viewed as a small database these days.
  - It is not uncommon to have databases with millions of records.

Summary

Middleware enables an application to provide business logic that is scalable and accessible over the internet. Middleware responds to GET and POST requests submitted by web pages running inside a web browser. Middleware that is backed by a database in our case PostgreSQL allows for the creation of business logic that can provide session contents for users' web pages. That means that each user can have information that is relevant to their session. Furthermore sessions can be isolated based on users' requests.

PostgreSQL has a distinct advantage over MongoDB. This advantage is that it is highly scalable and it allows for the creation of world systems. This comes at the cost of a higher learning curve than MongoDB. Note: with PosgreSQL we need to learn SQL in order to manipulate data in the database.

Assessments

Each week, you'll find here reminders of assignments or labs that you should be working on. For Week 6:

- Lab 6: Creating a middleware application that uses a PostgreSQL database (5% of the final grade) is due in 7 days, i.e., the end of day, Friday, of Week 6.

Reminder

You have a weekly Zoom session. The date/timing for your Zoom session can be found in the Course Schedule in the Course Outline. The link to the Zoom session will be provided separately to you by your instructor. The Zoom session will consist of 20–30 minutes of lecture, followed by questions and answers (Q&A), followed by a lab period. You may ask questions about the previous or current week's content or labs.

Please bring any questions to the weekly Zoom session. However, if you have any questions during the week outside of the Zoom session, you may post them in the Discussion Board. Your instructor may respond in the Discussion board or during the weekly Zoom session.

Click-n-reveal: **Where is the Discussion board?**

Click on the “Communication” area at the top main course menu, and select Discussions from the dropdown menu.

References

None for this week.