CKCS 145 – Lab 5 – Middleware with a Relational Backend

In this lab you will be working with Python, Flask, PostgreSQL and the command prompt. You will need to use the Virtual Machine that was set up in Lab 1. Start your Virtual Machine, log into the machine using your account ("TorontoMet" is the username and "torontomet123" is the password). Open up a Terminal or command prompt and start with step 1.

Prerequisites:

- Flask version 3.0.3 or higher
- Werkzeug 3.0.3 or higher
- SQLAlchemy 2.0.31 or higher
- Flask-SQLAlchemy 3.1.1 or higher

1. Before we get started with this lab we need to use PostgreSQL and create a database.
2. PostgreSQL is a headless service that runs on your virtual machine. You can use PgAdmin 4, a GUI application to create and populate databases. You have to enter your credentials into PgAdmin 4 twice. The first time is to log onto PgAdmin4 and the second time is to log into PostgresSQL server running on your virtual machine. The user name is "postgres" and the password is "torontomet123". By default PgAdmin uses "postgres" as the default user. You will only have to enter your password.
3. In PgAdmin 4 you need to create a database named CKCS145. You can use figures 1 and 2 as a guide.

Figure 1: Create a database

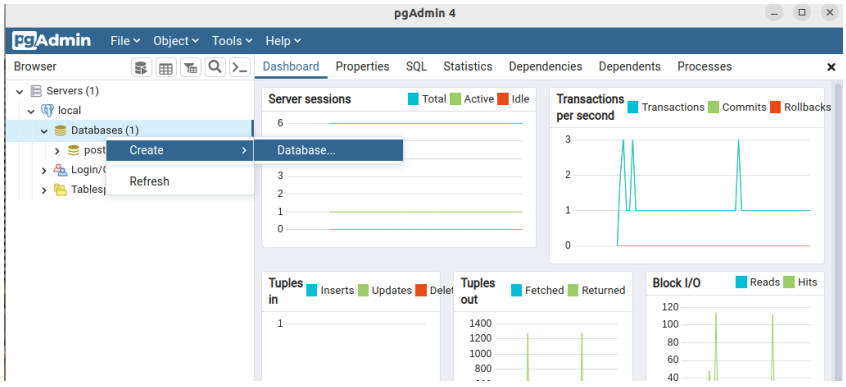
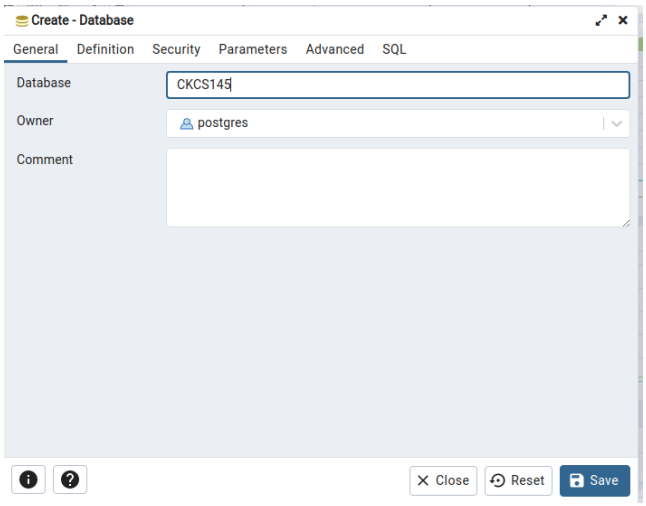


Figure 2: Enter name for database to be created



4. Inside the CKCS145 database you need to create a table named “User”. The “User” table has two columns “email” and “name” where their data types are “text”. Note, “email” column is a primary key. Once all of this has been entered you can click on the save button in the create table dialog. You can use figures 3, 4 and 5 as a guide. In Figure 5 you should notice the button with a + sign towards the top right of the window. This button is used for adding columns to your table.

Figure 3: Create table

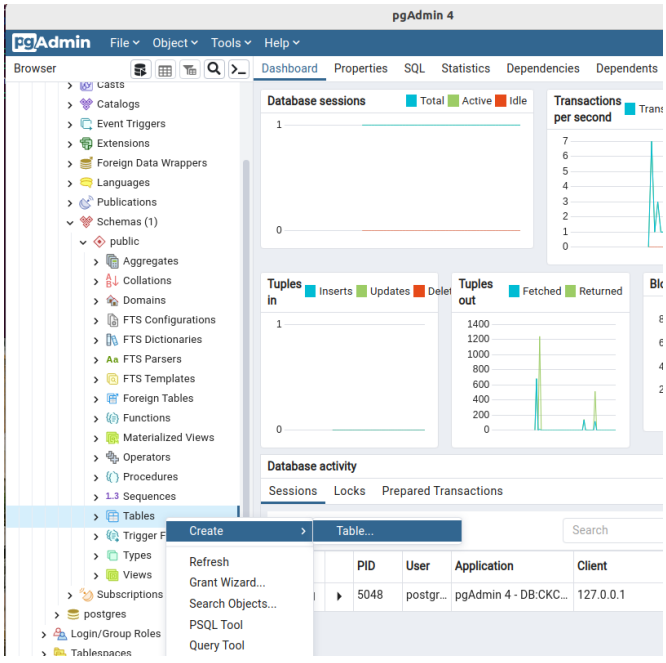


Figure 4:Enter name for table to be created

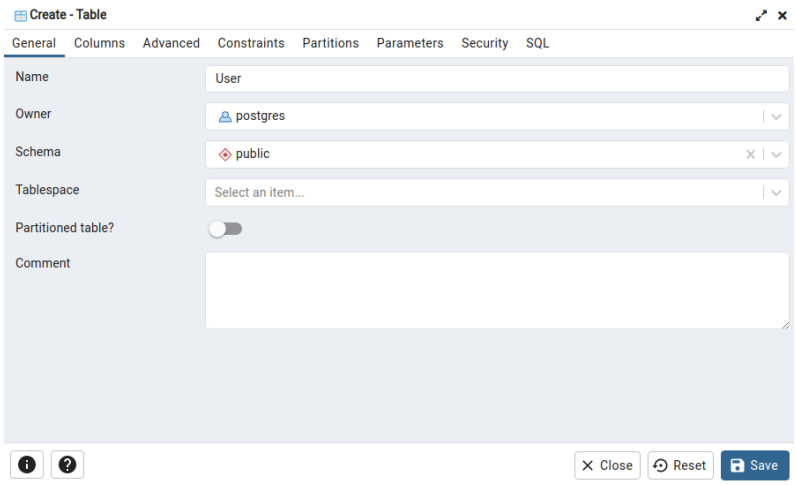
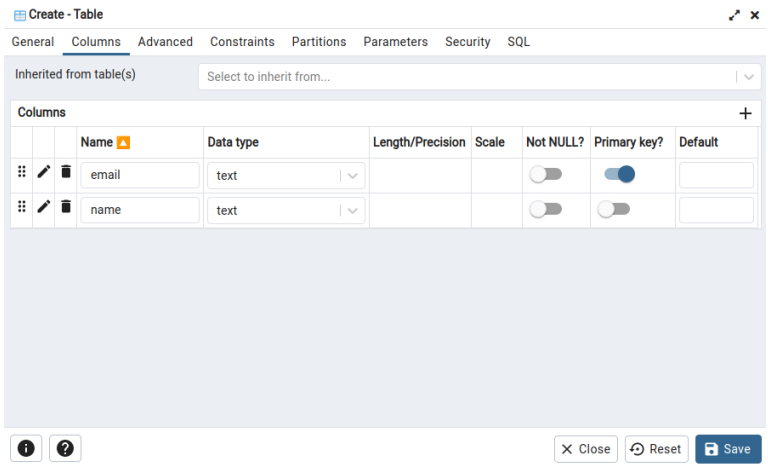


Figure 5: Enter columns and their data types for the table. Note the + sign towards the top right of the window. It is used for adding columns.



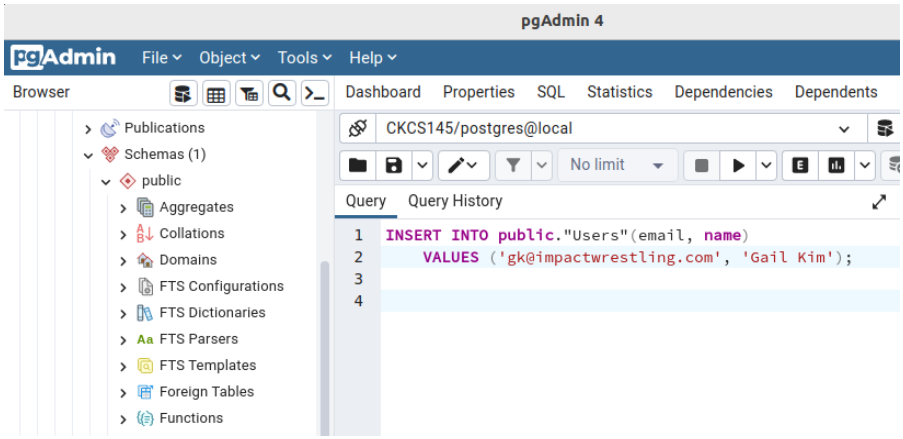
5. To the table named Users we can add records. Below you will find SQL commands to insert three records. The SQL insertion code is below. You can use figure 6 as a guide. To submit the insert statements below you need to click the execute button in PgAdmin 4. In figure 6, the execute button is located in the toolbar above the query. The execute button has a triangle pointing to the right and it is located to the right .

```
INSERT INTO public."User"(email, name)
   VALUES ('gk@impactwrestling.com', 'Gail Kim');


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


INSERT INTO public."User"(email, name)
   VALUES ('cc@photos.ca', 'Caitlin Cronenberg');
```

Figure 6: Insert a row using PgAdmin 4



6. Create a folder named “lab6”. This folder can be in your home folder or in your project folder.

```
mkdir lab6
```

7. Change the current directory to “lab6”. This can be achieved using the cd command

```
cd lab6
```

8. Install necessary tools to create a Flask application. From the command prompt you can run the following commands.

```
pip install sqlalchemy

pip install psycopg2-binary

pip install flask-sqlalchemy
```

9. Create a folder named “lab5”. This folder can be in your home folder or in your project folder.

```
mkdir lab5
```

10. Change the current directory to “lab5”. This can be achieved using the cd command.

```
cd lab5
```

11. Create a text file named “lab5-app.py”.

```
touch lab5-app.py
```

12. Open the above text file using a text editor and add the following inside it.

```
from flask import Flask, request, jsonify


from dataclasses import dataclass


from flask_sqlalchemy import SQLAlchemy
```

```
#from sqlalchemy import Column, Integer, String, ForeignKey



app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://postgres:torontomet123@localhost/CKCS145'

db = SQLAlchemy(app)



# A test route to determine if Flask application is initialized
properly.

# http://localhost:5000/test

@app.route('/test', methods=['GET'])

def test_route() :

    return 'test'


if __name__ == '__main__':

    app.run(debug=True)
```

13. Start the above application.

```
python3 lab5-app.py
```

14. The above application is Flask middleware with a single route. This means that it is a service that may be accessed on port 5000 using a web browser. The URL that may be used is http://localhost:5000/test/. What do you see in your web browser when you visit this URL?
15. Create a data model to access the document in MongoDB. The code below can be added above the route decorator used in test_route() method in lab4-app.py file.

```
@dataclass

class User(db.Model):

    __tablename__ = 'User'


    # declare fields - required for @dataclass

    email: str

    name: str


    # assignming values to fields
```

```
    email = db.Column( db.String(), primary_key=True )

    name = db.Column( db.String() )
```

16. Create a route for retrieving all documents from PostgreSQL. These routes can be added after test_route() method in lab5-app.py file.

```
# A route to list all users.

# http://localhost:5000/user/list

@app.route('/user/list', methods=['GET'])

def list_all_users():

    result_set = db.session.query( User ).all()

    return jsonify( result_set )@app.route('/user/list',
methods=['GET'])
```

    The above route can be tested by running the application. To run the application, use the command from step 13

17. Open a web browser and enter in the url: http://localhost:5000/user/list. You should see a JSON list of users. These users are retrieved by the middleware from the database.

18. Create a route for inserting a new user into PostgreSQL.

```
    # A route that allows the insertion for a single user.

    # http://localhost:5000/user/insert

    @app.route('/user/insert', methods=['POST'])

    def insert_user():

        name_val = request.form.get('user_name')

        email_val = request.form.get('user_email')


        print( 'data submitted:', name_val, email_val )


        new_user = User(email=email_val, name=name_val )

        db.session.add(new_user)

        db.session.commit()

        db.session.flush()


        status_message = 'working'


        return jsonify( {'status': status_message } )
```

19. The route in the above step requires a POST request from an html form to submit a name and email. RESTer can be used to submit such requests. Use figures 7 and 8 below to configure a POST request.

Figure 7: Configuration of content type for POST request.

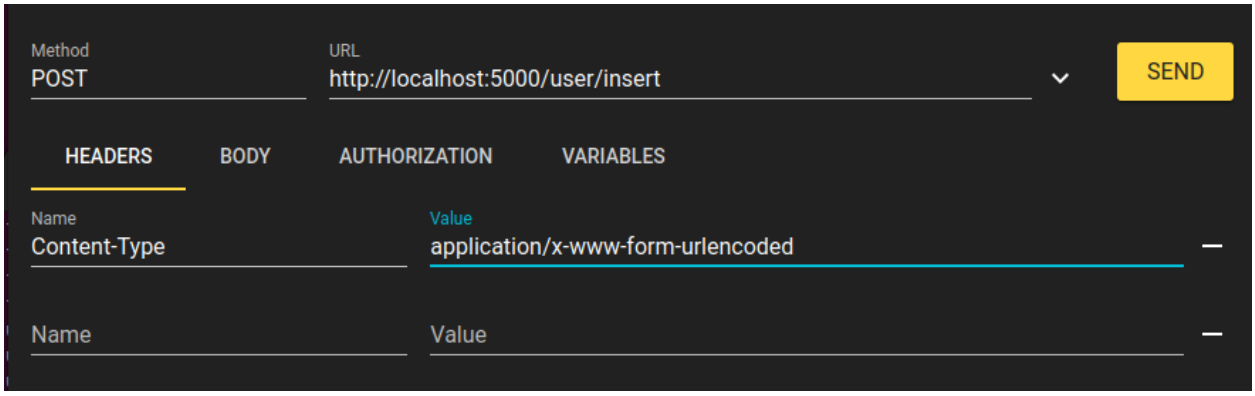
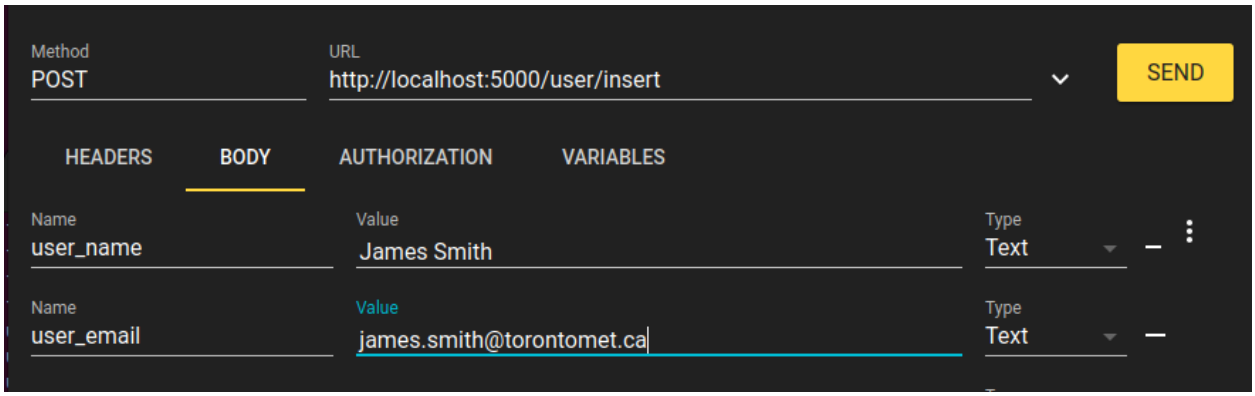


Figure 8: Configuration of “user_name” and “user_email” text fields and corresponding values.



20. Create a route for updating a User. To the route you must submit a user’s email and new name.

```
# A route that allows for changing the name of a single user.

# http://localhost:5000/user/update

@app.route('/user/update', methods=['POST'])

def update_student():


    name_val = request.form.get('user_name')

    email_val = request.form.get('user_email')


    query = db.session.query(User)

    query = query.filter(User.email==email_val)

    rows_changed = query.update({User.name: name_val, User.name:
name_val })


    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 } )
```

21. Create a route for deleting a User. To the route you must submit a user's email.

```
# A route that allows for deleting user.
# http://localhost:5000/user/delete
@app.route('/user/delete', methods=['POST'])
def delete_student():

    email_val = request.form.get('user_email')

    query = db.session.query(User)
    query = query.filter(User.email==email_val)

    rows_changed = query.delete( )

    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 } )
```

22. Use RESTer to create a URL POST request for the routes created in the above steps.
23. In this lab, only GET and POST requests are used. Why are PUT and DELETE requests not used?
24. If problems are encountered to complete this lab, it is recommended that debugging functionality be turned on. Stack traces can be viewed in the terminal where this app is running from. When errors are generated by web pages, stack traces can be used for identifying errors in your code. Alternatively, a stack trace can be viewed using your browser using a debug PIN. By default, debugging should be turned on.