COMS W4111-002, V02 (Spring 2022)
Introduction to Databases

Homework 2: Programming

Due Sunday, February 27, 2022 at 11:59 PM

Introduction

Overview

This homework has 1 section:

1. A section for programming track.

Submission

You will **submit 2 files** for this assignment.

1. Submit a zip file titled `<your_uni>_hw2_programming.zip` to **HW2 Programming - Zip** on Gradescope.
   - Replace `<your_uni>` with your uni. My submission would be `dff9_hw2_programming.zip` .
   - The zipped directory should include:
     - `classicmodels.sql`
     - `src`
       - `application.py`
       - `resources`
         - `__init__.py`
         - `base_resource.py`
         - `imdb_artists.py`
       - `rest_utils.py`
     - `<your_uni>_hw2_programming.ipynb` (substitute with your uni as above)
     - Any image files you embed in your notebook.

1. Submit a PDF title `<your_uni>_hw2_programming.pdf` to **HW2 Programming - PDF** on Gradescope.
   - This should be a PDF of your completed HW2 Programming Python notebook.
     - **Tag pages for each problem**. Per course policy, any untagged submission will receive an automatic 0.

- Double check your submission on Gradescope to ensure that the PDF conversion worked and that your pages are appropriately tagged.

Collaboration and Information

- Answering some of the questions may require independent research to find information. We encourage you to try troubleshooting problems independently before reaching out for help.
- You may use any information you get in TA or Prof. Ferguson's office hours, from lectures or from recitations. This includes slides related to the recommended textbook.
- You may use information that you find on the web.
- You are NOT allowed to collaborate with other students outside of office hours.

Programming

Setup

- Modify the cells below to setup your environment.
- The change should just be setting the DB user ID and password, replacing my user ID and password with yours for MySQL.

In [66]:
```
database_user_id = "root"
database_pwd = "dbuserdbuser"
```

In [67]:
```
database_url = "mysql+pymysql://" + \
    database_user_id + ":" + database_pwd + "@localhost"
database_url
```

Out[67]:
```
'mysql+pymysql://root:dbuserdbuser@localhost'
```

In [68]:
```
%reload_ext sql
```

In [69]:
```
%sql $database_url
```

Out[69]:
```
'Connected: root@None'
```

In [70]:
```
from sqlalchemy import create_engine
```

In [71]:
```
sqla_engine = create_engine(database_url)
```

In [72]:
```
#
```

```
# We are going to create a schema and some tables for the HW.
#
%sql create schema if not exists S22_W4111_HW2_B
%sql select 1;
```

```
 * mysql+pymysql://root:***@localhost
1 rows affected.
 * mysql+pymysql://root:***@localhost
1 rows affected.
```

Out[72]:

1
1

Install Dataset

Classic Models

- We will use the Classic Models Tutorial database for HW 2 Programming, other homework assignments, and exams.
- Lecture 5 briefly explained why this data model is interesting for educational purposes. The problems on homework assignments and exams will further explore why it's interesting.
- The zip file for HW 2 Programming contains an SQL script for creating a database `classicmodels` and loading the data. The script is `classicmodels.sql` .
- Use DataGrip to run the script. You performed this task for HW 0 with different SQL scripts. The basic approach is:
  - Right click on `@localhost`
  - Choose `Run SQL Script` .
  - Navigate to and select `classicmodels.sql` .
- The following cells test for correct installation.
- These cells are also examples of DDL statements and querying the "catalog."

In [73]:

```
%sql show tables from classicmodels
```

```
 * mysql+pymysql://root:***@localhost
8 rows affected.
```

Out[73]:

Tables_in_classicmodels
customers
employees
offices
orderdetails
orders
payments

Tables_in_classicmodels
productlines
products

In [74]:
```
%%sql

select
    table_schema, table_name, column_name, IS_NULLABLE, DATA_TYPE from informati
where
    table_schema='classicmodels'
order by
    table_schema, table_name, ORDINAL_POSITION
limit 20;
```

```
 * mysql+pymysql://root:***@localhost
20 rows affected.
```

Out[74]:

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	IS_NULLABLE	DATA_TYPE
classicmodels	customers	customerNumber	NO	int
classicmodels	customers	customerName	NO	varchar
classicmodels	customers	contactLastName	NO	varchar
classicmodels	customers	contactFirstName	NO	varchar
classicmodels	customers	phone	NO	varchar
classicmodels	customers	addressLine1	NO	varchar
classicmodels	customers	addressLine2	YES	varchar
classicmodels	customers	city	NO	varchar
classicmodels	customers	state	YES	varchar
classicmodels	customers	postalCode	YES	varchar
classicmodels	customers	country	NO	varchar
classicmodels	customers	salesRepEmployeeNumber	YES	int
classicmodels	customers	creditLimit	YES	decimal
classicmodels	employees	employeeNumber	NO	int
classicmodels	employees	lastName	NO	varchar
classicmodels	employees	firstName	NO	varchar
classicmodels	employees	extension	NO	varchar
classicmodels	employees	email	NO	varchar
classicmodels	employees	officeCode	NO	varchar
classicmodels	employees	reportsTo	YES	int

In [75]:
```
%%sql

use classicmodels;
with
```

```
    customer_orders_details as
        (
            select customerNumber, orderNumber, status, orderDate, shippedDate,
                   productCode, quantityOrdered, priceEach
            from orders natural join orderdetails
        ),
     customer_orders_totals as
         (
             select customerNumber, orderNumber,
                    concat(
                            '$',
                            format(sum(priceEach * quantityOrdered), 2)
                        ) as order_value
                from customer_orders_details
                group by customerNumber, orderNumber
         )
select * from customer_orders_totals
limit 20;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
20 rows affected.
```

Out[75]:

customerNumber	orderNumber	order_value
103	10123	$14,571.44
103	10298	$6,066.78
103	10345	$1,676.14
112	10124	$32,641.98
112	10278	$33,347.88
112	10346	$14,191.12
114	10120	$45,864.03
114	10125	$7,565.08
114	10223	$44,894.74
114	10342	$40,265.60
114	10347	$41,995.62
119	10275	$47,924.19
119	10315	$19,501.82
119	10375	$49,523.67
119	10425	$41,623.44
121	10103	$50,218.95
121	10158	$1,491.38
121	10309	$17,876.32
121	10325	$34,638.14
124	10113	$11,044.30

Tasks

- There is a sub-folder `src` of this directory that contains:
  - `application.py` which is a Flask application.
  - `rest_utils.py` is some helpful code for dealing with Flask and other objects.
  - `resources` is a package that contains:
    - `base_resource.py` defines the abstract class that all REST resources must implement.
    - `imdb_artists.py` contains a partially completed REST resource implementation.
- You must complete the implementation of `application.py` and implement a file `orders.py` that implements a class `Orders` . The class must implement the abstract methods defined in `base_resource` .
- In `application.py` you must implement support for the paths:
  - `/resource_collection`
    - GET on URLs of the forms `/orders?customerNumber=101&status=shipped&fields=customerNumber,orderNumber`
    - POST that has a JSON body defining the data for the new row.
  - `/resource_collection/id`
    - GET on URLs of the `/orders/101000`
    - DELETE
    - UPDATE, which takes a JSON body and updates the fields.
- You must test your paths below. The following is an example that tests GET.

In [13]:
```
import requests
```

In [ ]:
```
#
# Test get
#
url = "http://localhost:5003/api/imdb_artists/nm0000158"
res = requests.get(url)
res = res.json()

res
```

- Include at least one test for each remaining supported path below. You **must** display the output of each test.

In [53]:
```
#
# Test GET for /resource_collection
#
```

```
url = "http://localhost:5003/api/orders?customerNumber=114&status=shipped&fields
res = requests.get(url)
res = res.json()

res
```

Out[53]:
```
{'data': [{'customerNumber': 114, 'orderNumber': 10120},
  {'customerNumber': 114, 'orderNumber': 10125},
  {'customerNumber': 114, 'orderNumber': 10223},
  {'customerNumber': 114, 'orderNumber': 10342},
  {'customerNumber': 114, 'orderNumber': 10347}],
 'links': [{'rel': 'self',
   'href': 'http://localhost:5003/api/orders?customerNumber=114&status=shipped&f
ields=customerNumber,%20orderNumber'}]}
```

In [54]:
```
#
# Test GET for /resource_collection/id
#
url = "http://localhost:5003/api/orders/10101"
res = requests.get(url)
res = res.json()

res
```

Out[54]:
```
{'orderNumber': 10101,
 'orderDate': '2003-01-09',
 'requiredDate': '2003-01-18',
 'shippedDate': '2003-01-11',
 'status': 'Shipped',
 'comments': 'Check on availability.',
 'customerNumber': 128}
```

In [63]:
```
#
# Test POST
# calling create method (insert equiv)
#

url = "http://localhost:5003/api/orders"
post_data = {
    "orderNumber": 99995,
    "orderDate": '2001-07-22',
    "requiredDate": '2001-07-30',
    "status": "Shipped",
    "customerNumber": 103
}


res = requests.post(url, json=post_data)

res.text
```

Out[63]:
```
'CREATED'
```

In [62]:
```
#
# Test Delete
# calling
#
```

```
url = "http://localhost:5003/api/orders/99995"
res = requests.delete(url)

res.text
```

Out[62]:
```
'DELETED'
```

In [65]:
```
#
# Test UPDATE / PUT
# calling update_by_id (update equiv)
#

url = "http://localhost:5003/api/orders/10555"
update_data = {
    "customerNumber": 114,
    "status": "Not Shipped"
}


res = requests.put(url, json=update_data)
res.text
```

Out[65]:
```
'UPDATED'
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

- Include screenshots of all the code you wrote in `application.py` , `orders.py` , and any other Python files below.

In [ ]:
```
'''
#application.py

from flask import Flask, Response, request
from flask_cors import CORS
import json
from datetime import datetime
from resources.imdb_artists import IMDB_Artist
from resources.orders import Orders

import rest_utils

app = Flask(__name__)
CORS(app)

service_factory = dict()
```

```
##############################################################################

# DFF TODO A real service would have more robust health check methods.
# This path simply echoes to check that the app is working.
# The path is /health and the only method is GETs
@app.route("/health", methods=["GET"])
def health_check():
    rsp_data = {"status": "healthy", "time": str(datetime.now())}
    rsp_str = json.dumps(rsp_data)
    rsp = Response(rsp_str, status=200, content_type="application/json")
    return rsp


# TODO Remove later. Solely for explanatory purposes.
# The method take any REST request, and produces a response indicating what
# the parameters, headers, etc. are. This is simply for education purposes.
#
@app.route("/api/demo/<parameter1>", methods=["GET", "POST", "PUT", "DELETE"])
@app.route("/api/demo/", methods=["GET", "POST", "PUT", "DELETE"])
def demo(parameter1=None):
    """
    Returns a JSON object containing a description of the received request.

    :param parameter1: The first path parameter.
    :return: JSON document containing information about the request.
    """

    # DFF TODO -- We should wrap with an exception pattern.
    #

    # Mostly for isolation. The rest of the method is isolated from the specific
    inputs = rest_utils.RESTContext(request, {"parameter1": parameter1})

    # DFF TODO -- We should replace with logging.
    r_json = inputs.to_json()
    msg = {
        "/demo received the following inputs": inputs.to_json()
    }
    print("/api/demo/<parameter> received/returned:\n", msg)

    rsp = Response(json.dumps(msg), status=200, content_type="application/json")
    return rsp

##############################################################################


@app.route('/')
def hello_world():
    return '<u>Hello World!</u>'


@app.route('/api/<resource_collection>', methods=['GET', 'POST'])
def do_resource_collection(resource_collection):
    """
    1. HTTP GET return all resources.
    2. HTTP POST with body --> create a resource, i.e --> database. POST is inse
    :return:
    """
    request_inputs = rest_utils.RESTContext(request, resource_collection)
```

```
    svc = service_factory.get(resource_collection, None)

    if request_inputs.method == "GET":
        res = svc.get_by_template(path=None,
                                  template=request_inputs.args,
                                    field_list=request_inputs.fields,
                                    limit=request_inputs.limit,
                                    offset=request_inputs.offset)

        res = request_inputs.add_pagination(res)
        rsp = Response(json.dumps(res, default=str), status=200, content_type="a

    elif request_inputs.method == "POST":
        data = request_inputs.data
        res = svc.create(data)

        headers = [{"Location", "/users/" + str(res)}]
        rsp = Response("CREATED", status=201, headers=headers, content_type="tex
    else:
        rsp = Response("NOT IMPLEMENTED", status=501, content_type="text/plain")

    return rsp


@app.route('/api/<resource_collection>/<resource_id>', methods=['GET', 'PUT', 'D
def specific_resource(resource_collection, resource_id):
    """
    1. Get a specific one by ID.
    2. Update body and update. PUT is update
    3. Delete would ID and delete it.
    :param user_id:
    :return:
    """
    request_inputs = rest_utils.RESTContext(request, resource_collection)
    svc = service_factory.get(resource_collection)

    if request_inputs.method == "GET":
        res = svc.get_resource_by_id(resource_id)
        rsp = Response(json.dumps(res, default=str), status=200, content_type="a
    #I ADDED THIS
    elif request_inputs.method == "PUT":
        data = request_inputs.data
        res = svc.update_resource_by_id(resource_id, data) #data field here?
        if res == 1:
            rsp = Response("UPDATED", status=201, content_type="text/plain")
        else:
            rsp = Response("NOT UPDATED", status=500, content_type="text/plain")
    elif request_inputs.method == "DELETE":
        res = svc.delete_resource_by_id(resource_id)
        if res == 1:
            rsp = Response("DELETED", status=200, content_type="text/plain")
        else:
            rsp = Response("NOT DELETED", status=500, content_type="text/plain")
    #ENDS HERE
    else:
        rsp = Response("NOT IMPLEMENTED", status=501, content_type="text/plain")

    return rsp
```

```
if __name__ == '__main__':
    service_factory['imdb_artists'] = IMDB_Artist()
    service_factory['orders'] = Orders()
    app.run(host="0.0.0.0", port=5003)

'''
```

In [ ]:

```
'''
#orders.py

import pymysql
import json

from src.resources.base_resource import Base_Resource

class Orders(Base_Resource):

    def __init__(self):
        super().__init__()
        self.db_schema = 'classicmodels'
        self.db_table = 'orders'
        self.db_table_full_name = self.db_schema + "." + self.db_table
        self.primary_key_field = 'orderNumber'

    def _get_connection(self):
        """
        # DFF TODO There are so many anti-patterns here I do not know where to b
        :return:
        """

        # DFF TODO OMG. Did this idiot really put password information in source
        # Sure. Let's just commit this to GitHub and expose security vulnerabili
        #
        conn = pymysql.connect(
            host="localhost",
            port=3306,
            user="root",
            password="dbuserdbuser",
            cursorclass=pymysql.cursors.DictCursor,
            autocommit=True
        )
        return conn

    def get_resource_by_id(self, id):
        """
        # DFF TODO Will the anti-patterns never end?
        :return:
        """

        sql = "select * from " + self.db_table_full_name + " where orderNumber=%
        conn = self._get_connection()
        cursor = conn.cursor()

        the_sql = cursor.mogrify(sql, (id))
        print("The sql = ", the_sql)

        res = cursor.execute(sql, (id))

        if res == 1:
```

```
            result = cursor.fetchone()
        else:
            result = None

        return result


    def get_by_template(self,
                        path=None,
                        template=None,
                        field_list=None,
                        limit=None,
                        offset=None):
        """
        This is a logical abstraction of an SQL SELECT statement.

        Ignore path for now.

        Assume that
            - template is {'customerNumber': 101, 'status': 'Shipped'}
            - field_list is ['customerNumber', 'orderNumber', 'status', 'orderDa
            - self.get_full_table_name() returns 'classicmodels.orders'
            - Ignore limit for now
            - Ignore offset for now

        This method would logically execute

        select customerNumber, orderNumber, status, orderDate
            from classicmodels.orders
            where
                customerNumber=101 and status='Shipped'

        :param path: The relative path to the resource. Ignore for now.
        :param template: A dictionary of the form {key: value} to be converted t
        :param field_list: The subset of the fields to return.
        :param limit: Limit on number of rows to return.
        :param offset: Offset in the list of matching rows.
        :return: The rows matching the query.
        """
        field_str = ""
        for x in field_list:
            field_str += (x + ",")

        template_str = ""
        for (key,val) in template.items():
            template_str += (key + "='" + val + "' and ")

        sql = "select " + field_str[0:len(field_str)-1] + " from " + self.db_tab
        conn = self._get_connection()
        cursor = conn.cursor()

        the_sql = cursor.mogrify(sql)
        print("The sql = ", the_sql)

        res = cursor.execute(sql)

        if res > 0:
            result = cursor.fetchall()
        else:
            result = None
```

```
        return result

    def create(self, new_resource):
        """

        Assume that
            - new_resource is {'orderNumber': 101, 'status': 'Shipped'} #need a
            - self.get_full_table_name() returns 'classicmodels.orders'

        This function would logically perform

        insert into classicmodels.orders(customerNumber, status)
            values(101, 'Shipped')

        :param new_resource: A dictionary containing the data to insert.
        :return: Returns the values of the primary key columns in the order defi
            In this example, the result would be [101]
        """
        columns = ', '.join(new_resource.keys())

        values_str = ""
        for val in new_resource.values():
            values_str += ("'" + str(val) + "', ")

        sql = "insert into " + self.db_table_full_name + "(" + columns + ") valu
        conn = self._get_connection()
        cursor = conn.cursor()

        the_sql = cursor.mogrify(sql)
        print("The sql = ", the_sql)

        res = cursor.execute(sql)

        if res:
            result = new_resource["orderNumber"]
            print(result)
        else:
            result = None

        return result

    def update_resource_by_id(self, id, new_values):
        """
        This is a logical abstraction of an SQL UPDATE statement.

        Assume that
            - id is 30100
            - new_values is {'customerNumber': 101, 'status': 'Shipped'}
            - self.get_full_table_name() returns 'classicmodels.orders'

        This method would logically execute.

        update classicmodels.orders
            set customerNumber=101, status=shipped
            where
                orderNumber=30100


        :param id: The 'primary key' of the resource to update
```

```
        :new_values: A dictionary defining the columns to update and the new val
        :return: 1 if a resource was updated. 0 otherwise.
        """

        new_val_str = ""
        for (key,val) in new_values.items():
            new_val_str += (key + "='" + str(val) + "', ")

        sql = "update " + self.db_table_full_name + " set " + new_val_str[0:len(
        print(sql)
        conn = self._get_connection()
        cursor = conn.cursor()

        the_sql = cursor.mogrify(sql, (id))
        print("The sql = ", the_sql)

        res = cursor.execute(sql, (id))

        if res == 1:
            result = 1
        else:
            result = 0

        return result

    def delete_resource_by_id(self, id):
        """
        This is a logical abstraction of an SQL DELETE statement.

        Assume that
            - id is 30100
            - new_values is {'customerNumber': 101, 'status': 'Shipped'}

        This method would logically execute.

        delete from classicmodels.orders
            where
                orderNumber=30100


        :param id: The 'primary key' of the resource to delete
        :return: 1 if a resource was deleted. 0 otherwise.
        """

        sql = "delete from " + self.db_table_full_name + " where orderNumber=%s"
        conn = self._get_connection()
        cursor = conn.cursor()

        the_sql = cursor.mogrify(sql, (id))
        print("The sql = ", the_sql)

        res = cursor.execute(sql, (id))

        if res == 1:
            result = 1
        else:
            result = 0

        return result
```

```
if __name__ == "__main__":

    o = Orders() # made order
    res = o.get_resource_by_id('10101')
    print("Result = \n",
          json.dumps(res, indent=2, default=str))

'''
```