COMS W4111: Introduction to Databases
Spring 2024, Sections 002/V02

Homework 2: Programming

Introduction

This notebook contains HW2 Programming. **Only students on the programming track should complete this part.** To ensure everything runs as expected, work on this notebook in Jupyter.

Submission instructions:

- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
  - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. **Switch the orientation to landscape mode**, and hit save.
  - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
- For the ZIP:
  - Zip the folder that contains this notebook, any screenshots, and the code you write.
  - To avoid freezing Gradescope with too many files, when you finish this assignment, delete any unnecessary directories. Such directories include `venv`, `.idea`, and `.git`.

Setup

SQL Magic

The `sql` extension was installed in HW0. Double check that if this cell doesn't work.

In [1]:
```
%load_ext sql
```

You may need to change the password below.

In [2]:
```
%sql mysql+pymysql://root:tzy123456@localhost
```

In [3]:
```
%sql SELECT * FROM db_book.student WHERE ID = 12345
```

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

Out[3]:

ID	name	dept_name	tot_cred
12345	Shankar	Comp. Sci.	32

Python Libraries

In [4]:
```
# %pip install pandas
# %pip install sqlalchemy
```

```
# %pip install requests
```

In [5]:
```
import json

import pandas as pd
from sqlalchemy import create_engine
import requests
```

You may need to change the password below.

In [6]:
```
engine = create_engine("mysql+pymysql://root:tzy123456@localhost")
```

Data Definition and Insertion

Create Tables

- The directory contains a file `people_info.csv`. The columns are
  - `first_name`
  - `middle_name`
  - `last_name`
  - `email`
  - `employee_type,` which can be one of `Professor`, `Lecturer`, `Staff`. The value is empty if the person is a student.
  - `enrollment_year` which must be in the range `2016–2023`. The value is empty if the person is an employee.
- In the cell below, create two tables, `student` and `employee`
  - You should choose appropriate data types for the attributes
  - You should add an attribute `student_id` to `student` and `employee_id` to `employee`. **These attributes should be auto-incrementing numbers.** They are the PKs of their tables.
  - `email` should be unique and non-null in their tables. You don't need to worry about checking whether `email` is unique across both tables.
  - `student` should have all the columns listed above except `employee_type`. You should have some way to ensure that `enrollment_year` is always in range.
  - `employee` should have all the columns listed above except `enrollment_year`. You should have some way to ensure that `employee_type` is one of the valid values.

In [7]:
```
%%sql

DROP SCHEMA IF EXISTS s24_hw2;
CREATE SCHEMA s24_hw2;
USE s24_hw2;

## Add CREATE TABLEs below
create table student
(
    student_id      INT auto_increment,
    first_name      VARCHAR(32)  null,
    middle_name     varchar(32)  null,
    last_name       varchar(32)  null,
    email           varchar(128) not null,
    enrollment_year INT          not null,
    constraint student_pk
        primary key (student_id),
    constraint student_uk
        unique (email),
    constraint enroll_year
        check (student.enrollment_year between 2016 and 2023)
);
```

```
create table employee
(
    employee_id   int auto_increment,
    first_name    varchar(32)                              null,
    middle_name   varchar(32)                              null,
    last_name     varchar(32)                              null,
    email         varchar(128)                             not null,
    employee_type enum ('Professor', 'Lecturer', 'Staff') not null,
    constraint employee_pk
        primary key (employee_id),
    constraint employee_uk
        unique (email)
);
```

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

Out[7]: []

Inserting Data

- Below we read `people_info.csv` into a Pandas Dataframe
- You should implement `get_students` and `get_employees`, which extract the student/employee rows from a dataframe of people
- If you implement the functions correctly, the next cell should run with no errors and insert data into the tables you created above

In [8]:
```
df = pd.read_csv("./people_info.csv")
df.head()
```

Out[8]:

	first_name	middle_name	last_name	email	employee_type	enrollment_year
0	Sanders	Arline	Breckell	abreckell1x@fotki.com	Professor	NaN
1	Zared	NaN	Fenelon	afenelona@themeforest.net	NaN	2021.0
2	Ethelin	NaN	Fidele	afidele12@google.ru	Lecturer	NaN
3	Bibbye	Annabal	Guesford	aguesfordb@tumblr.com	NaN	2018.0
4	Xenia	Ardella	Kief	akieft@free.fr	Staff	NaN

In [9]:
```
def get_students(df):
    """Given a dataframe of people df, returns a new dataframe that only contains students.
    The returned dataframe should have all the attributes of the people df except `employee_type`.
    """
    df = df[df['employee_type'].isna()]
    df = df.drop(columns=['employee_type'])
    return df

def get_employees(df):
    """Given a dataframe of people df, returns a new dataframe that only contains employees.
    The returned dataframe should have all the attributes of the people df except `enrollment_year`.
    """
    df = df[df['enrollment_year'].isna()]
    df = df.drop(columns=['enrollment_year'])
    return df
```

In [10]:
```
student_df = get_students(df)
employee_df = get_employees(df)
```

```
student_df.head()
```

Out[10]:

	first_name	middle_name	last_name	email	enrollment_year
1	Zared	NaN	Fenelon	afenelona@themeforest.net	2021.0
3	Bibbye	Annabal	Guesford	aguesfordb@tumblr.com	2018.0
6	Duffy	Auberon	Pounder	apounder2h@reuters.com	2017.0
7	Base	NaN	Baybutt	bbaybutty@tmall.com	2021.0
9	Jenine	Berry	Habberjam	bhabberjam2k@examiner.com	2021.0

In [11]:
```
student_df.to_sql("student", schema="s24_hw2", index=False, if_exists="append", con=engine)
employee_df.to_sql("employee", schema="s24_hw2", index=False, if_exists="append", con=engine)
```

Out[11]:
```
50
```

API Implementation

- You will create an API that allows users to read, create, update, and delete students and employees
- The `src/` directory has the following structure:

```
src
|
|- db.py
|
|- db_test.py
|
|- main.py
```

Python Environment

1. Open the `src/` folder in PyCharm
2. Follow these instructions to set up a virtual environment. This'll give us an blank, isolated environment for packages that we install. It's fine to use the `Virtualenv Environment` tab.
3. Open the Terminal in PyCharm. Make sure your virtual environment is active (you'll probably see `(venv)` somewhere).
   A. If you don't, the docs may be helpful
4. Run `pip install -r requirements.txt`
   A. `requirements.txt` contains all the packages that the project needs, such as `pymysql`

db.py

- Implement the eight methods in `db.py`: `build_select_query`, `select`, `build_insert_query`, `insert`, `build_update_query`, `update`, `build_delete_query`, and `delete`
  - To see examples of the inputs and expected outputs for the `build_*` functions, see `db_test.py`

db_test.py

- To test your `build_*` methods, run the `db_test.py` file. This file defines some unit tests.
- **Post a screenshot of your successful tests below**

Successful Unit Tests

main.py

- `main.py` declares our API and defines paths for it
  - The `@app` decorator above each method describes the HTTP method and the path associated with that method
- Implement the ten endpoints in `main.py`: `get_students`, `get_student`, `post_student`, `put_student`, `delete_student`, `get_employees`, `get_employee`, `post_employee`, `put_employee`, and `delete_employee`

Testing Your API

Student Testing

- With your API running, execute the following cells
  - Successful cells may have no output. However, failing cells will generate an error.

In [12]:
```
import json

import pandas as pd
from sqlalchemy import create_engine
import requests
```

In [13]:
```
BASE_URL = "http://localhost:8002/"

def print_json(j):
    print(json.dumps(j, indent=2))
```

In [14]:
```
# Healthcheck

r = requests.get(BASE_URL)
print(r.text)
```

```
<h1>Heartbeat</h1>
```

In [15]:
```
# Get all students

r = requests.get(BASE_URL + "students")

# {}

j = r.json()

assert len(j) == 50, "There should be 50 students after inserting data"
```

In [16]:
```
# Get specific attributes

r = requests.get(BASE_URL + "students?enrollment_year=2018&fields=first_name,last_name")
```

```
# {'enrollment_year': '2018', 'fields': 'first_name,last_name'}

j = r.json()

print_json(j)
assert len(j) == 7, "There should be 7 students that graudated in 2018"
assert all(map(lambda o: len(o) == 2 and "first_name" in o and "last_name" in o, j)), \
"All student JSONs should have two attributes, first_name and last_name"
```


```
[
  {
    "first_name": "Bibbye",
    "last_name": "Guesford"
  },
  {
    "first_name": "Barry",
    "last_name": "Elias"
  },
  {
    "first_name": "Avie",
    "last_name": "Blissitt"
  },
  {
    "first_name": "Shea",
    "last_name": "Bates"
  },
  {
    "first_name": "Mal",
    "last_name": "Issett"
  },
  {
    "first_name": "Rozelle",
    "last_name": "Vigar"
  },
  {
    "first_name": "Drona",
    "last_name": "McKinie"
  }
]
```


In [17]:
```
# Test bad gets

# Invalid ID
r = requests.get(BASE_URL + "students/100")
assert r.status_code == 404, f"Invalid ID: Expected 404 Not Found but got {r.status_code}"
```

In [18]:
```
# Create a new student

or_student = {
    "first_name": "Michael",
    "last_name": "Jan",
    "email": "ap@columbia.edu",
    "enrollment_year": 2019,
}

r = requests.post(BASE_URL + "students", json=or_student)
assert r.status_code == 201, f"Expected 201 Created but got {r.status_code}"
```

In [19]:
```
# Get that student

r = requests.get(BASE_URL + "students/51")
j = r.json()[0]
```

```
print_json(j)

assert j == {
    'student_id': 51,
    'first_name': 'Michael',
    'middle_name': None,
    'last_name': 'Jan',
    'email': 'ap@columbia.edu',
    'enrollment_year': 2019,
}, "Newly inserted student does not match what we specified"
```


```
{
  "student_id": 51,
  "first_name": "Michael",
  "middle_name": null,
  "last_name": "Jan",
  "email": "ap@columbia.edu",
  "enrollment_year": 2019
}
```


In [20]:
```
# Test bad posts

# Duplicate email
bad_student = {
    "first_name": "Foo",
    "last_name": "Bar",
    "email": "ap@columbia.edu",
    "enrollment_year": 2018,
}
r = requests.post(BASE_URL + "students", json=bad_student)
assert r.status_code == 400, f"Duplicate email: Expected 400 Bad Request but got {r.status_code}"
```

In [21]:
```
# Email not specified
bad_student = {
    "first_name": "Foo",
    "last_name": "Bar",
    "enrollment_year": 2018,
}
r = requests.post(BASE_URL + "students", json=bad_student)
assert r.status_code == 400, f"Email not specified: Expected 400 Bad Request but got {r.status_code}"
```

In [22]:
```
# Invalid year
bad_student = {
    "first_name": "Foo",
    "last_name": "Bar",
    "email": "fb@columbia.edu",
    "enrollment_year": 2011,
}
r = requests.post(BASE_URL + "students", json=bad_student)
assert r.status_code == 400, f"Invalid year: Expected 400 Bad Request but got {r.status_code}"
```

In [23]:
```
# Update the student

r = requests.put(BASE_URL + "students/51", json={"enrollment_year": "2020"})
assert r.status_code == 200, f"Expected 200 OK but got {r.status_code}"
```

In [24]:
```
# Test bad puts

# Duplicate email
bad_student = {
    "email": "csimeons2@microsoft.com",
}
r = requests.put(BASE_URL + "students/51", json=bad_student)
```

```
assert r.status_code == 400, f"Duplicate email: Expected 400 Bad Request but got {r.status_code}"
```

In [25]:
```
# Email set to null
bad_student = {
    "email": None
}
r = requests.put(BASE_URL + "students/51", json=bad_student)
assert r.status_code == 400, f"Null email: Expected 400 Bad Request but got {r.status_code}"
```

In [26]:
```
# Invalid year
bad_student = {
    "enrollment_year": 2011
}
r = requests.put(BASE_URL + "students/51", json=bad_student)
assert r.status_code == 400, f"Invalid year: Expected 400 Bad Request but got {r.status_code}"
```

In [27]:
```
# Invalid ID
bad_student = {
    "enrollment_year": 2018
}
r = requests.put(BASE_URL + "students/100", json=bad_student)
assert r.status_code == 404, f"Invalid ID: Expected 404 Not Found but got {r.status_code}"
```

In [28]:
```
# Delete the student

r = requests.delete(BASE_URL + "students/51")
assert r.status_code == 200, f"Expected 200 OK but got {r.status_code}"
```

In [29]:
```
# Try to get deleted student

r = requests.get(BASE_URL + "students/51")
assert r.status_code == 404, f"Expected 404 Not Found but got {r.status_code}"
```

In [30]:
```
# Test bad deletes

r = requests.delete(BASE_URL + "students/100")
assert r.status_code == 404, f"Invalid ID: Expected 404 Not Found but got {r.status_code}"
```

Employee Testing

- Write similar tests below to test your employee endpoints

In [31]:
```
# Get all employees

r = requests.get(BASE_URL + "employees")

# {}

j = r.json()

assert len(j) == 50, "There should be 50 employeess before inserting data"
```

In [32]:
```
# Test bad gets

# Invalid ID
r = requests.get(BASE_URL + "employees/100")
assert r.status_code == 404, f"Invalid ID: Expected 404 Not Found but got {r.status_code}"
```

In [33]:
```
# Create a new employee
```

```
new_employee = {
    "first_name": "Zhiyuan",
    "last_name": "Tan",
    "email": "zt2337@columbia.edu",
    "employee_type": 'Professor',
}

r = requests.post(BASE_URL + "employees", json=new_employee)
assert r.status_code == 201, f"Expected 201 Created but got {r.status_code}"
```

In [34]:
```
# Get that employee

r = requests.get(BASE_URL + "employees/51")
j = r.json()[0]

print_json(j)

assert j == {
    'employee_id': 51,
    'first_name': 'Zhiyuan',
    'middle_name': None,
    'last_name': 'Tan',
    'email': 'zt2337@columbia.edu',
    'employee_type': 'Professor',
}, "Newly inserted employee does not match what we specified"
```


```
{
  "employee_id": 51,
  "first_name": "Zhiyuan",
  "middle_name": null,
  "last_name": "Tan",
  "email": "zt2337@columbia.edu",
  "employee_type": "Professor"
}
```


In [35]:
```
# Test bad posts

# Duplicate email
bad_employee = {
    "first_name": "Foo",
    "last_name": "Bar",
    "email": "abreckell1x@fotki.com",
    "employee_type": 'Professor',
}
r = requests.post(BASE_URL + "employees", json=bad_employee)
assert r.status_code == 400, f"Duplicate email: Expected 400 Bad Request but got {r.status_code}"
```

In [36]:
```
# Email not specified
bad_employee = {
    "first_name": "Foo",
    "last_name": "Bar",
    "employee_type": 'Professor',
}
r = requests.post(BASE_URL + "employees", json=bad_employee)
assert r.status_code == 400, f"Email not specified: Expected 400 Bad Request but got {r.status_code}"
```

In [37]:
```
# Invalid enrollment type
bad_employee = {
    "first_name": "Foo",
    "last_name": "Bar",
    "email": "fb@columbia.edu",
    "employee_type": 'Student',
}
r = requests.post(BASE_URL + "employees", json=bad_employee)
```

```
assert r.status_code == 400, f"Invalid year: Expected 400 Bad Request but got {r.status_code}"
```

In [38]:
```
# Update the employee

r = requests.put(BASE_URL + "employees/51", json={"employee_type": "Lecturer"})
assert r.status_code == 200, f"Expected 200 OK but got {r.status_code}"
```

In [39]:
```
# Delete the employee

r = requests.delete(BASE_URL + "employees/51")
assert r.status_code == 200, f"Expected 200 OK but got {r.status_code}"
```

In [40]:
```
# Test bad deletes

r = requests.delete(BASE_URL + "employees/100")
assert r.status_code == 404, f"Invalid ID: Expected 404 Not Found but got {r.status_code}"
```

Appendix

db.py - myversion

In [41]:
```
from typing import Any, Dict, List, Tuple, Union

import pymysql

# Type definitions
# Key-value pairs
KV = Dict[str, Any]
# A Query consists of a string (possibly with placeholders) and a list of values to be put in the placeholders
Query = Tuple[str, List]

class DB:
        def __init__(self, host: str, port: int, user: str, password: str, database: str):
                conn = pymysql.connect(
                        host=host,
                        port=port,
                        user=user,
                        password=password,
                        database=database,
                        cursorclass=pymysql.cursors.DictCursor,
                        autocommit=True,
                )
                self.conn = conn

        def get_cursor(self):
                return self.conn.cursor()

        def execute_query(self, query: str, args: List, ret_result: bool) -> Union[List[KV], int]:
                """Executes a query.

                :param query: A query string, possibly containing %s placeholders
                :param args: A list containing the values for the %s placeholders
                :param ret_result: If True, execute_query returns a list of dicts, each representing a returned
                                                        row from the table. If False, the number of rows affected is returned. Note
                                                        that the length of the list of dicts is not necessarily equal to the number
                                                        of rows affected.
                :returns: a list of dicts or a number, depending on ret_result
                """
                cur = self.get_cursor()
                count = cur.execute(query, args=args)
```

```
                if ret_result:
                        return cur.fetchall()
                else:
                        return count


        # TODO: all methods below


        @staticmethod
        def build_select_query(table: str, rows: List[str], filters: KV) -> Query:
                """Builds a query that selects rows. See db_test for examples.

                :param table: The table to be selected from
                :param rows: The attributes to select. If empty, then selects all rows.
                :param filters: Key-value pairs that the rows from table must satisfy
                :returns: A query string and any placeholder arguments
                """
                # start with the SLECT
                query_str = 'SELECT '

                # add the 'rows' part
                if len(rows) == 0:
                        query_str += '* '
                else:
                        for i in range(len(rows) - 1):
                                query_str += rows[i]
                                query_str += ', '

                        query_str += rows[-1]
                        query_str += ' '

                # add the FROM and the table
                query_str += 'FROM '
                query_str += table
                query_str += ' '

                # add the WHERE and the filter
                replace_val = []

                if len(filters) == 0:
                        query_str = query_str[:-1]
                else:
                        query_str += 'WHERE '
                        for i in range(len(filters) - 1):
                                query_str += list(filters.keys())[i]
                                query_str += ' = %s AND '
                                replace_val.append(list(filters.values())[i])

                        query_str += list(filters.keys())[-1]
                        query_str += ' = %s'
                        replace_val.append(list(filters.values())[-1])

                return query_str, replace_val

        def select(self, table: str, rows: List[str], filters: KV) -> List[KV]:
                """Runs a select statement. You should use build_select_query and execute_query.

                :param table: The table to be selected from
                :param rows: The attributes to select. If empty, then selects all rows.
                :param filters: Key-value pairs that the rows to be selected must satisfy
                :returns: The selected rows
                """
                query_str, replace_val = self.build_select_query(table, rows, filters)
```

```
        return self.execute_query(query_str, replace_val, True)

    @staticmethod
    def build_insert_query(table: str, values: KV) -> Query:
        """Builds a query that inserts a row. See db_test for examples.

        :param table: The table to be inserted into
        :param values: Key-value pairs that represent the values to be inserted
        :returns: A query string and any placeholder arguments
        """
        # begin with INSERT INTO
        query_str = 'INSERT INTO '

        # add table
        query_str += table
        query_str += ' ('

        # add values key
        keys = list(values.keys())
        vls  = list(values.values())

        for i in range(len(keys) - 1):
            query_str += keys[i]
            query_str += ', '

        query_str += keys[-1]
        query_str += ') VALUES ('

        # add values values
        replace_val = []
        for i in range(len(vls) - 1):
            query_str += '%s, '
            replace_val.append(vls[i])

        query_str += '%s)'
        replace_val.append(vls[-1])

        return query_str, replace_val

    def insert(self, table: str, values: KV) -> int:
        """Runs an insert statement. You should use build_insert_query and execute_query.

        :param table: The table to be inserted into
        :param values: Key-value pairs that represent the values to be inserted
        :returns: The number of rows affected
        """
        query_str, replace_val = self.build_insert_query(table, values)

        return self.execute_query(query_str, replace_val, False)

    @staticmethod
    def build_update_query(table: str, values: KV, filters: KV) -> Query:
        """Builds a query that updates rows. See db_test for examples.

        :param table: The table to be updated
        :param values: Key-value pairs that represent the new values
        :param filters: Key-value pairs that the rows from table must satisfy
        :returns: A query string and any placeholder arguments
        """
        # start with the SLECT
        query_str = 'UPDATE '

        # add the table and SET
```

```
                query_str += table
                query_str += ' SET '

                # add the value
                replace_val = []

                values_key = list(values.keys())
                values_val = list(values.values())
                filter_key = list(filters.keys())
                filter_val = list(filters.values())

                for i in range(len(values_key) - 1):
                        query_str += values_key[i]
                        query_str += ' = %s, '
                        replace_val.append(values_val[i])

                query_str += values_key[-1]
                query_str += ' = %s'
                replace_val.append(values_val[-1])

                # add the filter
                if len(filters) == 0:
                        pass
                else:
                        query_str += ' WHERE '
                        for i in range(len(filter_key) - 1):
                                query_str += filter_key[i]
                                query_str += ' = %s AND '
                                replace_val.append(filter_val[i])

                        query_str += filter_key[-1]
                        query_str += ' = %s'
                        replace_val.append(filter_val[-1])

                return query_str, replace_val

        def update(self, table: str, values: KV, filters: KV) -> int:
                """Runs an update statement. You should use build_update_query and execute_query.

                :param table: The table to be updated
                :param values: Key-value pairs that represent the new values
                :param filters: Key-value pairs that the rows to be updated must satisfy
                :returns: The number of rows affected
                """
                query_str, replace_val = self.build_update_query(table, values, filters)

                return self.execute_query(query_str, replace_val, False)

        @staticmethod
        def build_delete_query(table: str, filters: KV) -> Query:
                """Builds a query that deletes rows. See db_test for examples.

                :param table: The table to be deleted from
                :param filters: Key-value pairs that the rows to be deleted must satisfy
                :returns: A query string and any placeholder arguments
                """
                # start with the DELETE FROM
                query_str = 'DELETE FROM '

                # add the table
                query_str += table
                query_str += ' '

                # add the WHERE and the filter
```

```
                replace_val = []

                if len(filters) == 0:
                        query_str = query_str[:-1]
                else:
                        query_str += 'WHERE '
                        for i in range(len(filters) - 1):
                                query_str += list(filters.keys())[i]
                                query_str += ' = %s AND '
                                replace_val.append(list(filters.values())[i])

                        query_str += list(filters.keys())[-1]
                        query_str += ' = %s'
                        replace_val.append(list(filters.values())[-1])

                return query_str, replace_val

        def delete(self, table: str, filters: KV) -> int:
                """Runs a delete statement. You should use build_delete_query and execute_query.

                :param table: The table to be deleted from
                :param filters: Key-value pairs that the rows to be deleted must satisfy
                :returns: The number of rows affected
                """
                query_str, replace_val = self.build_delete_query(table, filters)

                return self.execute_query(query_str, replace_val, False)
```

main.py - myversion

In [ ]:

```
from typing import Any, Dict

# Simple starter project to test installation and environment.
# Based on https://fastapi.tiangolo.com/tutorial/first-steps/
from fastapi import FastAPI, Response, Request, status
from fastapi.responses import HTMLResponse, JSONResponse
# Explicitly included uvicorn to enable starting within main program.
# Starting within main program is a simple way to enable running
# the code within the PyCharm debugger
import uvicorn

from db import DB

# Type definitions
KV = Dict[str, Any]  # Key-value pairs

app = FastAPI()

# NOTE: In a prod environment, never put this information in code!
# There are design patterns for passing confidential information to
# application.
# TODO: You may need to change the password
db = DB(
        host="localhost",
        port=3306,
        user="root",
        password="tzy123456",
        database="s24_hw2",
)

@app.get("/")
async def healthcheck():
        return HTMLResponse(content="<h1>Heartbeat</h1>", status_code=status.HTTP_200_OK)
```

```python
# TODO: all methods below

# --- STUDENTS ---

@app.get("/students")
async def get_students(req: Request):
        """Gets all students that satisfy the specified query parameters.

        For instance,
                GET http://0.0.0.0:8002/students
        should return all attributes for all students.

        For instance,
                GET http://0.0.0.0:8002/students?first_name=John&last_name=Doe
        should return all attributes for students whose first name is John and last name is Doe.

        You must implement a special query parameter, `fields`. You can assume
        `fields` is a comma-separated list of attribute names. For instance,
                GET http://0.0.0.0:8002/students?first_name=John&fields=first_name,email
        should return the first name and email for students whose first name is John.
        Not every request will have a `fields` parameter.

        You can assume the query parameters are valid attribute names in the student table
        (except `fields`).

        :param req: The request that optionally contains query parameters
        :returns: A list of dicts representing students. The HTTP status should be set to 200 OK.
        """

        # Use `dict(req.query_params)` to access query parameters
        query_param = dict(req.query_params)

        # prep table, rows, and filters for select
        table = 'student'
        rows  = []
        if 'fields' in query_param.keys():
                rows = list(query_param['fields'].split(','))

        filters = query_param.copy()
        if 'fields' in query_param.keys():
                filters.pop('fields')

        res = db.select(table, rows, filters)
        return JSONResponse(content=res, status_code=status.HTTP_200_OK)

@app.get("/students/{student_id}")
async def get_student(student_id: int):
        """Gets a student by ID.

        For instance,
                GET http://0.0.0.0:8002/students/1
        should return the student with student ID 1. The returned value should
        be a dict-like object, not a list.

        If the student ID doesn't exist, the HTTP status should be set to 404 Not Found.

        :param student_id: The ID to be matched
        :returns: If the student ID exists, a dict representing the student with HTTP status set to 200 OK.
                                If the student ID doesn't exist, the HTTP status should be set to 404 Not Found.
        """
        # prep table, rows, and filters for select
        table = 'student'
```

```
        rows  = []
        filters = {'student_id': student_id}

        res = db.select(table, rows, filters)
        if not res:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)
        else:
                return JSONResponse(content=res, status_code=status.HTTP_200_OK)

@app.post("/students")
async def post_student(req: Request):
        """Creates a student.

        You can assume the body of the POST request is a JSON object that represents a student.
        You can assume the request body contains only attributes found in the student table and does not
        attempt to set `student_id`.

        For instance,
                POST http://0.0.0.0:8002/students
                {
                        "first_name": "John",
                        "last_name": "Doe",
                        ...
                }
        should create a student with the attributes specified in the request body.

        If the email is not specified in the request body, the HTTP status should be set to 400 Bad Request.
        If the email already exists in the student table, the HTTP status should be set to 400 Bad Request.
        If the enrollment year is not valid, the HTTP status should be set to 400 Bad Request.

        :param req: The request, which contains a student JSON in its body
        :returns: If the request is valid, the HTTP status should be set to 201 Created.
                                If the request is not valid, the HTTP status should be set to 400 Bad Request.
        """

        # Use `await req.json()` to access the request body
        query_param = await req.json()

        # prep table, rows, and filters for select
        table = 'student'

        try:
                res = db.insert(table, query_param)
                return HTMLResponse(status_code=status.HTTP_201_CREATED)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)

@app.put("/students/{student_id}")
async def put_student(student_id: int, req: Request):
        """Updates a student.

        You can assume the body of the PUT request is a JSON object that represents a student.
        You can assume the request body contains only attributes found in the student table and does not
        attempt to update `student_id`.

        For instance,
                PUT http://0.0.0.0:8002/students/1
                {
                        "first_name": "Joe"
                }
        should update the student with student ID 1's first name to Joe.

        If the student does not exist, the HTTP status should be set to 404 Not Found.
```

```
        If the email is set to null in the request body, the HTTP status should be set to 400 Bad Request.
        If the email already exists in the student table, the HTTP status should be set to 400 Bad Request.
        If the enrollment year is not valid, the HTTP status should be set to 400 Bad Request.

        :param student_id: The ID of the student to be updated
        :param req: The request, which contains a student JSON in its body
        :returns: If the request is valid, the HTTP status should be set to 200 OK.
                                If the request is not valid, the HTTP status should be set to the appropriate error code.
        """

        # Use `await req.json()` to access the request body
        query_param = await req.json()

        # prep table, rows, and filters for select
        table = 'student'

        values = query_param

        filters = {'student_id': student_id}

        # examine if the student exists
        stu_existence = db.select(table, [], filters)
        if not stu_existence:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)

        # try to update
        try:
                res = db.update(table, values, filters)
                return HTMLResponse(status_code=status.HTTP_200_OK)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)
```

```
@app.delete("/students/{student_id}")
async def delete_student(student_id: int):
        """Deletes a student.

        For instance,
                DELETE http://0.0.0.0:8002/students/1
        should delete the student with student ID 1.

        If the student does not exist, the HTTP status should be set to 404 Not Found.

        :param student_id: The ID of the student to be deleted
        :returns: If the request is valid, the HTTP status should be set to 200 OK.
                                If the request is not valid, the HTTP status should be set to 404 Not Found.
        """
        # examine if the student exists
        table = 'student'
        filters = {'student_id': student_id}

        stu_existence = db.select(table, [], filters)
        if not stu_existence:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)

        # try to delete
        try:
                res = db.delete(table, filters)
                return HTMLResponse(status_code=status.HTTP_200_OK)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)
```

```python
# --- EMPLOYEES ---

@app.get("/employees")
async def get_employees(req: Request):
        """Gets all employees that satisfy the specified query parameters.

        For instance,
                GET http://0.0.0.0:8002/employees
        should return all attributes for all employees.

        For instance,
                GET http://0.0.0.0:8002/employees?first_name=Don&last_name=Ferguson
        should return all attributes for employees whose first name is Don and last name is Ferguson.

        You must implement a special query parameter, `fields`. You can assume
        `fields` is a comma-separated list of attribute names. For instance,
                GET http://0.0.0.0:8002/employees?first_name=Don&fields=first_name,email
        should return the first name and email for employees whose first name is Don.
        Not every request will have a `fields` parameter.

        You can assume the query parameters are valid attribute names in the employee table
        (except `fields`).

        :param req: The request that optionally contains query parameters
        :returns: A list of dicts representing employees. The HTTP status should be set to 200 OK.
        """

        # Use `dict(req.query_params)` to access query parameters
        query_param = dict(req.query_params)

        # prep table, rows, and filters for select
        table = 'employee'
        rows  = []
        if 'fields' in query_param.keys():
                rows = list(query_param['fields'].split(','))

        filters = query_param.copy()
        if 'fields' in query_param.keys():
                filters.pop('fields')

        res = db.select(table, rows, filters)
        return JSONResponse(content=res, status_code=status.HTTP_200_OK)

@app.get("/employees/{employee_id}")
async def get_employee(employee_id: int):
        """Gets an employee by ID.

        For instance,
                GET http://0.0.0.0:8002/employees/1
        should return the employee with employee ID 1. The returned value should
        be a dict-like object, not a list.

        If the employee ID doesn't exist, the HTTP status should be set to 404 Not Found.

        :param employee_id: The ID to be matched
        :returns: If the employee ID exists, a dict representing the employee with HTTP status set to 200 OK.
                                If the employee ID doesn't exist, the HTTP status should be set to 404 Not Found.
        """
        # prep table, rows, and filters for select
        table = 'employee'
        rows  = []
        filters = {'employee_id': employee_id}

        res = db.select(table, rows, filters)
```

```
        if not res:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)
        else:
                return JSONResponse(content=res, status_code=status.HTTP_200_OK)

@app.post("/employees")
async def post_employee(req: Request):
        """Creates an employee.

        You can assume the body of the POST request is a JSON object that represents an employee.
        You can assume the request body contains only attributes found in the employee table and does not
        attempt to set `employee_id`.

        For instance,
                POST http://0.0.0.0:8002/employees
                {
                        "first_name": "Don",
                        "last_name": "Ferguson",
                        ...
                }
        should create an employee with the attributes specified in the request body.

        If the email is not specified in the request body, the HTTP status should be set to 400 Bad Request.
        If the email already exists in the employee table, the HTTP status should be set to 400 Bad Request.
        If the employee type is not valid, the HTTP status should be set to 400 Bad Request.

        :param req: The request, which contains an employee JSON in its body
        :returns: If the request is valid, the HTTP status should be set to 201 Created.
                                If the request is not valid, the HTTP status should be set to 400 Bad Request.
        """

        # Use `await req.json()` to access the request body
        query_param = await req.json()

        # prep table, rows, and filters for select
        table = 'employee'

        try:
                res = db.insert(table, query_param)
                return HTMLResponse(status_code=status.HTTP_201_CREATED)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)

@app.put("/employees/{employee_id}")
async def put_employee(employee_id: int, req: Request):
        """Updates an employee.

        You can assume the body of the PUT request is a JSON object that represents an employee.
        You can assume the request body contains only attributes found in the employee table and does not
        attempt to update `employee_id`.

        For instance,
                PUT http://0.0.0.0:8002/employees/1
                {
                        "first_name": "Donald"
                }
        should update the employee with employee ID 1's first name to Donald.

        If the employee does not exist, the HTTP status should be set to 404 Not Found.
        If the email is set to null in the request body, the HTTP status should be set to 400 Bad Request.
        If the email already exists in the employee table, the HTTP status should be set to 400 Bad Request.
        If the employee type is not valid, the HTTP status should be set to 400 Bad Request.
```

```
        :param employee_id: The ID of the employee to be updated
        :param req: The request, which contains an employee JSON in its body
        :returns: If the request is valid, the HTTP status should be set to 200 OK.
                                If the request is not valid, the HTTP status should be set to the appropriate error code.
        """

        # Use `await req.json()` to access the request body
        query_param = await req.json()

        # prep table, rows, and filters for select
        table = 'employee'

        values = query_param

        filters = {'employee_id': employee_id}

        # examine if the student exists
        stu_existence = db.select(table, [], filters)
        if not stu_existence:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)

        # try to update
        try:
                res = db.update(table, values, filters)
                return HTMLResponse(status_code=status.HTTP_200_OK)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)

@app.delete("/employees/{employee_id}")
async def delete_employee(employee_id: int):
        """Deletes an employee.

        For instance,
                DELETE http://0.0.0.0:8002/employees/1
        should delete the employee with employee ID 1.

        If the employee does not exist, the HTTP status should be set to 404 Not Found.

        :param employee_id: The ID of the employee to be deleted
        :returns: If the request is valid, the HTTP status should be set to 200 OK.
                                If the request is not valid, the HTTP status should be set to 404 Not Found.
        """
        # examine if the employee exists
        table = 'employee'
        filters = {'employee_id': employee_id}

        stu_existence = db.select(table, [], filters)
        if not stu_existence:
                return HTMLResponse(status_code=status.HTTP_404_NOT_FOUND)

        # try to delete
        try:
                res = db.delete(table, filters)
                return HTMLResponse(status_code=status.HTTP_200_OK)
        except Exception as e:
                print(e)
                return HTMLResponse(status_code=status.HTTP_400_BAD_REQUEST)

if __name__ == "__main__":
        uvicorn.run(app, host="0.0.0.0", port=8002)
```