Relational Databases and RESTful APIs

Goals: The goal of this lab assignment is twofold. In the first part, you'll learn how to code a Python application that talks to a relational database. In the second part, you'll discover how open data is shared over the Web through RESTful APIs.

The following Python packages will be used in this lab assignment:

- SQLAlchemy
- requests
- csv
- time
- json

1 SQL and Python: Object Relational Mapper

We want to create *LinkCS*, a new social network that allows students in CentraleSupélec to connect and communicate. The implementation of *LinkCS* requires a relational database, which stores the actual social network, and a software application, which lets the students access the social network. The application will be the interface between the students, which are the *end users* of the social network, and the underlying relational database.

To know : What is an Object Relational Mapper?

The communication between an application and a database is hindered by the fact that the two don't speak the same language. More specifically, an application written in an object-oriented language understands the notions of *classes* and *objects*, while a relational database only uses, as its name implies, *relations*, or *tables*. This gap between the application and the database is often referred to as *impedance mismatch*.
One way to address the impedance mismatch is to leave the programmer with the tedious task of writing the code necessary to convert classes and objects into tables, and the other way round. Another option is to use a *Object Relational Mapper* (ORM), a software library that bridges the gap between an object-oriented application and a relational database with minimal intervention from the programmer. By using an ORM, the programmer can write an application that communicates with a relational database without using SQL.

Most programming languages have at least an ORM; Python has a few: *DjangoORM*, *Storm*, *PoneyORM*, and *SQLAlchemy*, just to name a few. We will use *SQLAchemy*.

1.1 Database design

We intend to create a simple database with two tables:

- A table named `Student` with the following columns:
  - `studentNumber`: a numeric identifier of a student.
  - `firstName`: the first name of a student.
  - `lastName`: the last name of a student.
  - `nickname`: the nickname of a student.
  - `country`: the country of origin of a student.
- A table named `Association` with the following columns:
  - `asso_id`: a numeric identifier of an association.
  - `name`: the name of an association.
  - `description`: a textual description of the activities of an association.

Exercice 1 : Logical Model

Draw the logical model for this database.

1.2 Database creation

We'll now walk you through the creation of this database by using *SQLAlchemy* and a relational database management system (DBMS) called *SQLite*. Unlike other DBMSs, such as *MySQL* and *PostgreSQL*, SQLite is a database engine embedded in the application rather than a client-server engine.

Open a new Python file in Pycharm named `orm-introduction.py` to write the code of the application. We first import from SQLAlchemy all the functions that we'll need in the remainder of the assignment.

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
from sqlalchemy import Table, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.orm import sessionmaker
```

Next, we create the SQLite database engine with the following code.

```
engine = create_engine('sqlite:///:memory:', echo=False)
```

The directive `sqlite:///:memory:` means that the database will be stored in memory (RAM), rather than in the hard disk. This is an acceptable choice when the database is expected to contain little data. The directive `echo=False` indicates that SQLAlchemy will not print the SQL queries that it generates. `create_engine()` returns an instance of the class `Engine` that represents the interface to the underlying relational database. When using SQLAlchemy, the programmer doesn't need to manipulate the object `engine` directly; it will be used by SQLAlchemy behind the scenes.

Declare a Mapping. The objective of using an ORM, such as SQLAlchemy, is to manipulate (i.e., query and update) the tables in the relational database through classes defined in the Python application. Thus, the first brick in the application must be the definition of the classes with the code necessary to declare the mapping to the corresponding relational tables. The mapping is stored in a so-called **declarative base class** that is instantiated as follows.

```
Base = declarative_base()
```

We can now define the classes that describe the tables in our database. The following code declares a class named `Student` that corresponds to the homonymous table (directive `__tablename__='Student'`) and five columns: `studentNumber` (integer, the primary key), `firstName` (String, not null), `lastName` (String, not null), `nickname` (String) and `country` (String).

```
class Student(Base):
  __tablename__ = 'Student'
  studentNumber = Column(Integer, primary_key=True)
  firstName = Column(String, nullable=False)
  lastName = Column(String, nullable=False)
  nickname = Column(String)
  country = Column(String)
```

Exercice 2 : Class `Association`

Based on the code that declares the class `Student`, write the code to create a class named `Association` corresponding to the homonymous table.

A student might belong to some associations. In other words, there is a many-to-many relationship between any student and any association. According to the rules that dictate the translation of the logical model into the relational model, this relationship is mapped to a relational table (referred to as an *association table*), where the columns are `studentNumber` and `asso_id` (the primary keys of the table `Student` and `Association`, respectively)

Before the definition of the classes `Student` and `Association`, we create the association table that we name `Affiliation`, as follows.

```
affiliation = Table('Affiliation', Base.metadata,
  Column('asso_id', Integer,ForeignKey('Association.asso_id'), nullable=False),
  Column('student_id',Integer,ForeignKey('Student.studentNumber'),nullable=False))
```

The previous code declares a table that is added to the base class with the directive `Base.metadata`; the column `asso_id` (respectively, `student_id`) is a foreign key referencing the column `asso_id` (resp., `studentNumber`) in table `Association` (resp., `Student`).

Unlike the tables `Student` and `Association`, the table `affiliation` is not mapped to any Python class. The reason is that this table is not manipulated directly in the Python code, but is populated automatically by SQLAlchemy whenever a student is added to an association. In order to activate this behavior, we need to modify the definition of the class `Association` to add a new attribute `students`, and to modify the definition of the class `Student` to add a new attribute `associations`, as follows:

```
class Student(Base):
  __tablename__ = 'Student'
  studentNumber = Column(Integer, primary_key=True)
  firstName = Column(String, nullable=False)
  lastName = Column(String, nullable=False)
  nickname = Column(String)
  country = Column(String)
  associations = relationship('Association', secondary=affiliation,
                              back_populates='students')


class Association(Base):
  __tablename__ = 'Association'
  asso_id = Column(Integer, primary_key=True)
  name = Column(String, nullable=False)
  description = Column(String)
  students = relationship('Student', secondary=affiliation,
                          back_populates='associations')
```

Both new attributes `associations` and `students` are defined by using the SQLAlchemy function `relationship`. We can think of `associations` and `students` as two collections (e.g., lists), where each item is an object of the class `Association` and `Student` respectively. The function `relationship` in the above code takes in three arguments. The first (e.g., "Student") informs the ORM that each item of the collection `students` in the table `Association` is an object of the class `Student`. The second (introduced with the keyword `secondary`) informs the ORM that the relationship is many-to-many and that the association table is `affiliation` (that we defined above). The third argument (introduced with the keyword `back_populates`) is used to make the relationship bidirectional; in other words, the attribute `associations` in class `Student` is symmetric to the attribute `students` in class `Association`. As a result, when we add an association to the collection `associations` of a student, that student will be automatically added to the collection `students` of that association.

In order to actually create the tables in the underlying database, the following instruction must be added:

```
Base.metadata.create_all(engine)
print("Tables created!")
```

1.3 Populating the database

Once the tables are created, we need to populate them (i.e., add some data). To this extent, SQLAlchemy provides the notion of *session*. A session is an object that serves as the interface to the database; session objects are created through a `Session` class that is defined as follows:

```
Session = sessionmaker(bind=engine)
```

In other words, the class `Session` is the factory for new session objects. Note that `Session` is binded to the database `engine` that we defined previously.

In order to create a session object, the constructor of the class `Session` is invoked:

```
session = Session()
```

If we want to add a student to the database, we simply create a new `Student` object and then we add it to the session, as follows:

```
nox = Student(studentNumber=1, firstName='Thomas', lastName='Adier',
              nickname='nox', country='France')
session.add(nox)
print("nox added!")
```

Exercice 3 : Create an association

Create a new association called "ViaRezo" with the description "Bringing Internet to the campus".
We can add more students at once using the function `add_all()`:

```
enizor = Student(studentNumber=2, firstName='Remi', lastName='Garde',
                 nickname='enizor', country='France')
gossex = Student(studentNumber=3, firstName='David', lastName='Gosset',
                 nickname='gossex', country='France')
xix = Student(studentNumber=4, firstName='Julen', lastName='Dixneuf',
              nickname='xix', country='France')
session.add_all([enizor, gossex, xix])
print("enizor, gossex, xix added!")
```

We can add all the students as members of ViaRezo, as follows:

```
viarezo.students.append(nox)
viarezo.students.append(enizor)
viarezo.students.append(gossex)
viarezo.students.append(xix)
```

We can verify that the students have been correctly added with the following code:

```
print("STUDENTS IN VIAREZO")
for student in viarezo.students:
    print(student.nickname)
```

Given the bi-directionality of the relationship between a student and an association, ViaRezo is automatically added to the list of the associations of each student.

Exercice 4 :

Write the code to verify that ViaRezo appears in the associations of the student `xix`.

Note that the data added to the `session` is not persisted into the database until either it is queried or the application explicitly calls the function `commit()` on the `session`. As a rule of thumb, after you're done with adding or modifying data, you should call the function `commit()` explicitly:

```
session.commit()
```

1.4 Queries

Queries are executed against the database by using the method `query()` on the `session` object. The following query returns all the students in the database

```
session.query(Student)
```

In order to loop over the result of this query and, say, print the first and family name of each student, the following code is used:

```
print("GETTING ALL STUDENTS")
for student in session.query(Student):
    print(student.firstName, student.lastName)
```

The call `session.query(Student)` is tantamount to the SQL query `SELECT * FROM Student`; the following code translates the query `SELECT firstName, lastName FROM Student`:

```
print("GETTING ALL STUDENTS")
for firstName, lastName in session.query(Student.firstName, Student.lastName):
    print (firstName, lastName)
```

The function `filter()` is used to specify a WHERE clause:

```
print("GETTING ADIER")
for student in session.query(Student).filter(Student.lastName=='Adier'):
    print(student.nickname)
```

We can add as many WHERE clauses as we wish, by concatenating the calls to the function `filter()`, as follows:

```
print("GETTING THOMAS ADIER")
for student in session.query(Student).\
        filter(Student.lastName=='Adier')\
        .filter(Student.firstName=='Thomas'):
    print(student.nickname)
```

Here you'll find a list of operators that can be used as an argument of the function `filter()`.

If you're certain that your query only returns one row (e.g., there is only one association with name ViaRezo), you can use the function `one()` as follows:

```
viarezo = session.query(Association).filter(Association.name=='ViaRezo').one()
print(viarezo.description)
```

However, if you apply `one` to a query that returns multiple results, SQLAlchemy will raise a `MultipleResultsFound` exception.

Finally, you can obtain a row by specifying the value of the primary key with the function `get()`. The following code returns the student with `studentNumber`=1:

```
first = session.query(Student).get(1)
print(first.lastName)
```

1.5 Exercises

Exercice 5 : Modifying the database

Modify the logical model in order to:

- Add the role of a student in an association.
- Add the email addresses of each student (a student can have several).

Exercice 6 : Create the tables

Open a new Python file named `orm-exo.py` in Pycharm and write the code to generate the tables of the new database.
Hint: since we added the role of a student in an association, the affiliation will be implemented with an *association object*, whose description can be found here. Moreover, in order to add the email addresses of the students, you'll need to learn how to implement a one-to-many relationship.

Exercice 7 : Populate the tables

Write a Python function named `populate()` that reads the CSV files `students.csv`, `associations.csv`, `affiliation.csv` and `addresses.csv` and populates the database. You can then use the two functions `print_students()` and `print_associations()` defined below to verify that the database is actually populated.
Hint:

- You'll need to learn how to add data when a many-to-many relationship is implemented with an association object.
- The delimiter character in each CSV file is the semicolon; please note that the first row of each file is the header and therefore must be skipped by using the function next().

```
def print_students(session):
    for stud in session.query(Student):
        info = ",".join([str(stud.studentNumber), stud.firstName,
                         stud.lastName, stud.nickname, stud.country])
        affiliations = "["
        for affi in stud.associations:
            affiliations += affi.asso.name + " --> " + affi.role \
                if len(affiliations) == 1 \
                else ", " + affi.asso.name + " --> " + affi.role
        affiliations += "]"
        info += ", associations=" + affiliations
        addresses = "["
        for addr in stud.addresses:
            addresses += addr.email if len(addresses) == 1 else ", " + addr.email
        addresses += "]"
```

```
        info +=  ", addresses=" + addresses
        print(info)

def print_associations(session):
    for asso in session.query(Association):
        info = asso.name + " \"" + asso.description
        students = "["
        for affi in asso.students:
            students += affi.student.firstName + " " + affi.student.lastName \
                        + " --> " + affi.role \
                if len(students) == 1 \
                else  ", " + affi.student.firstName + " " \
                      + affi.student.lastName + " --> " + affi.role
        students += "]"
        info += "students=" + students
        print(info)
```

Exercice 8 : Counting

Write a query in SQLAlchemy to count the number of students from Brazil (Expected result: 3).
Hint: Check out how to count in SQLAlchemy.

Exercice 9 : Counting

Write a query in SQLAlchemy to count the students in the association ViaRezo (Expected result: 8).
Hint: use `any()` for many-to-many relationships and `has()` for many-to-one relationships. You can learn more here.

Exercice 10 : Counting and grouping by

Write a query in SQLAlchemy to count the students in the association ViaRezo by country (Expected result: Brazil 3, France 2, Germany 2, Morocco 1).
Hint: Look here for an analogous example.

Exercice 11 : Query with filter

Write a query in SQLAlchemy to print the name of all students that are presidents of at least one association (Expected result: Thomas Adier, Remi Garde, Julie Rose, Elizabeth Red, Nour Ayoub).

Exercice 12 : Delete

Write the code necessary to remove the association named "BDI". Remember to call `commit()` to make the changes persistent in the database.
Hint: When we remove the association BDI (that has `asso_id` 2), by default SQLAlchemy replaces the value 2 with NULL in the column `asso_id` of the table `Affiliation`. However, this would not be allowed by the underlying relational database, because the column `asso_id` is part of the primary key. Therefore, we have to instruct SQLAlchemy to delete the rows in `Affiliation` where the value of `asso_id` is 2. This is achieved by adding the directive `cascade="all, delete, delete-orphan"` to the relationship `students` in the table `Association`, as explained here.

Exercice 13 : Update

Write the code necessary to add the email address `julier@gmail.com` to the student Julie Rose (`student_id` = 8).

2 Open Data and APIs

Have you ever heard of the buzzword *Open Data*? It's the idea that data should be freely available to anyone. In the the past few years, the Open Data movement gained traction so much so that even governments started to disclose some of their data to the public. Of course, data needs to be in a format that allows fast retrieval and automatic processing.

One common way to share data that software applications can grab and use is through an API, short for *Application Programming Interface*, more specifically a `RESTful` API, that allows an application to get some data through a simple HTTP request, the same technology used by a Web browser to request a Web page.

An example of RESTful API is the one provided by OpenWeatherMap, a website that provides weather forecast across the world.

2.1 Setup

OpenWeatherMap provides different APIs depending on the the type of application you intend to develop. The free API allows your application to get the current weather in any location, as well as 5-days forecasts, with no more than 60 requests per minute.

In order to use the API, you'll need to create a free account; go to the home page, click on `Sign Up` and follow the instructions. Once the account is created, you'll be redirected to your home page; if you click on `API keys`, you'll land on a page where you can find the API key that your application needs to use to request weather data.

2.2 First Example

Suppose that we want to get the current weather in Paris through a Python application. We can query OpenWeatherMap by using a URL in a specific format:

`http://api.openweathermap.org/data/2.5/weather?q=Paris,France&APPID=your-app-id`

The prefix `http://api.openweathermap.org/data/2.5/weather?` is the way the application connects to the weather service of the OpenWeatherMap website. What comes after the question mark is a list of parameters separated by the character &. The first parameter `q` is used to specify the name of the city, possibly followed by a comma (,) and the name of the country; the value of the parameter `APPID` is the API key needed to submit the request to the Web service.

Open a new Python file named `weather.py` and write the following code (remember to specify your API key in the URL request).

```
import requests

request = requests.get("http://api.openweathermap.org/data/2.5/weather?"
                       "q=Paris,France&APPID=your-app-id")
rawData = request.content.decode('utf-8')
print(rawData)
```

As you can see, the data returned by *OpenWeatherMap* is in a special key-value format, where the key (e.g., `base`, `coord`) is the name of a parameter and the value can be a simple literal (e.g., `stations`) or a set of key-value pairs (e.g., `{"lon":2.35,"lat":48.86}`). This format is called JSON (JavaScript Object Notation) and is commonly used as a data representation in RESTful APIs.

The syntax of JSON is quite simple:

- Curly braces ({ }) hold JSON documents and each key is followed by a colon (:), the key/value pairs are separated by a comma (,).
- Square brackets ([ ]) hold arrays and values are separated by a comma (,).

```
{
   "coord":{
      "lon":2.35,
      "lat":48.86
   },
   "weather":[
      {
         "id":800,
         "main":"Clear",
         "description":"clear sky",
         "icon":"01d"
      }
   ],
   "base":"stations",
   "main":{
      "temp":297.52,
      "pressure":1024,
      "humidity":36,
      "temp_min":296.15,
      "temp_max":298.15
   },
   "visibility":10000,
   "wind":{
      "speed":3.1,
      "deg":280
   },
   "clouds":{
      "all":0
   },
   "dt":1536588000,
   "sys":{
      "type":1,
      "id":5610,
      "message":0.3603,
      "country":"FR",
      "sunrise":1536556840,
      "sunset":1536603192
   },
   "id":2988507,
   "name":"Paris",
   "cod":200
}
```


The variable `rawData` is a string that contains the text of the JSON document; as a result, it is not suitable to efficiently extract the weather information that we need, such as, for instance, the temperature. For this reason, we first load the textual JSON content into a Python dictionary `jsonData` that is the ideal data structure, given that it is itself based on key-value pairs.

```
import json
jsonData = json.loads(rawData)
```

Finally, if we need to access the value of any key (e.g., `coord`), we just type:

```
print(jsonData["coord"])
```

If we want to only get the latitude, we would type:

```
print(jsonData["coord"]["lat"])
```

Finally, arrays are handled in the usual way. For instance, the value of the key `weather` is an array that only contains one JSON document; to get it, we write as follows:

```
print(jsonData["weather"][0])
```

2.3 Exercises

Exercice 14 : Get current weather

Write the code to print the textual description of the current weather in Paris.

Exercice 15 : Get temperature in Celsius

As you can see from the responses, the temperature is in Kelvin. Looking at the documentation, write the code to obtain the temperature in Celsius.

Exercice 16 : Cities in cycle

Looking at the documentation, write the code to obtain up to 50 locations around Paris and their temperature. **Hint:** Print the JSON response of OpenWeatherMap in order to see which keys you'll need to use to extract the requested information.

The free version of *OpenWeatherMap* allows an application to get a 5-day weather forecast for any location or city. For each day, the data is provided every three hours. The documentation that you need to access the weather forecast service is available here. In section "Weather parameters in API respond" of the documentation, you'll find a description of the keys in the response. The key `list` has as its value an array where each element is itself a JSON document containing the information of the forecast at a specific time and date.

The temporal information can be extracted with the key `list.dt`, whose value is the Unix time (also known as *Epoch*), i.e., the number of seconds elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. In order to translate the Epoch into a readable date, you can use the following code:

```
import time
epoch = 1536656400
t = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(epoch))
print(t)
```

Exercice 17 : Weather forecast

Write the Python code to print the 5-day forecast for a location of your choice. Each line of the output should have the following information: date and time, description of the weather conditions (e.g., "clear sky") and the temperature.

Current average temperature in a country. Open a new Python file named `temperature.py`. We're now going to code a function `avg_temperature()` that calculates the current average temperature in a given country. More specifically, the function takes in the bounding box containing the country that is identified by the following four parameters:

- `lon_left`: the longitude of the westmost point on the box.
- `lat_bottom`: the latitude of the southmost point on the box.

- `lon_right`: the longitude of the eastmost point on the box.
- `lat_top`: the latitude of the northmost point on the box.

All the above coordinates are given as decimal values. Additionally, the function takes in a fifth parameter `zoom` that is used to control the number of locations in the bounding box.

You can go to this website to obtain the bounding box of a country; in the menu at the bottom left corner ("Copy & Paste"), select CSV in order to get the coordinates of the bounding box as (`lon_left`, `lat_bottom`, `lon_right` and `lat_top`).

The OpenWeatherMap documentation describes how to pass a bounding box to the Web service. As for the value of `zoom`, set it so as to have between 100 and 300 cities (for France, a good value is 9).

Exercice 18 : Function `avg_temperature`

Code the function `avg_temperature` and test it on a country of your choice.
You'd probably noticed that a bounding box may contain some locations that are not in the country enclosed in the box. We need to filter out those locations. Unfortunately, if you look at the raw JSON document returned by *OpenWeatherMap* in response to the query, you'll see that only the identifier (key `id`) and the name (key `name`) of the location is mentioned (not the country). In order to get the country of a location, we need to query *OpenWeatherMap* by using the identifier of the location.

Exercice 19 : Function `get_country()`

Define a function `get_country()` in `temperature.py` that takes in the identifier of a location and returns the name of its country. Try the function with the identifier 3107775 that corresponds to a Spanish location called Torrelavega.
Hint: Look at the documentation to find out how to query by location ID.
You can now modify the function `avg_temperature` that you defined above in order to filter out the locations that are not in the country of your choice.

IMPORTANT: please consider that you have up to 60 requests per minute; if you exceed this number, your API key will be disabled. To avoid the problem, you might want to use the function `time.sleep()` that delays your application for a selected number of seconds.

```
import time
time.sleep(5) # delays for 5 seconds.
```

Exercice 20 : Function `avg_temperature`

Modify the function `avg_temperature` so as to calculate the average of the temperature based only on the locations in the targeted country.