

Création d'une base de données pour une application de gestion d'étudiants

Objectif :

Développer la partie “*Back-end*” d'une application en ligne permettant de gérer une liste d'étudiants.

Travail demandé :

1. Tester les 4 cas d'utilisation du **CRUD** avec les commandes **SQL** décrites dans le cours dans un “bac à sable” (<https://mysql-sandbox.nullx.me/>)
2. Reprendre ces tests avec **MySQL** en local. On utilisera l'interface graphique de **phpMyAdmin** (<http://127.0.0.1/phpmyadmin>)
3. Après avoir défini un utilisateur spécifique à la base de données pour un accès externe (port **3306**), coder une application en langage **Python** pour accéder à ces données.
4. Faire évoluer cette application pour qu'elle soit capable de répondre à des requêtes **HTTP** et renvoyer les résultats demandés en **JSON**
(cf doc du TP front-end : Utilisation d'une API REST)
Cela nécessite l'installation du serveur Flask pour Python

Librairies utilisées :

1. *mysql.connector* : Connexion à la base de données
(https://www.w3schools.com/python/python_mysql_getstarted.asp)

```
import mysql.connector

mydb = mysql.connector.connect(
    host = "127.0.0.1",
    user = "root",
    password = "",
    database = "ciel2025"
)

cursor = mydb.cursor()
request = "SELECT * FROM etudiant"
cursor.execute(request)
result = cursor.fetchall()

for record in result:
    print(record)
```

flask : Serveur web

Il faut tout d'abord créer un environnement virtuel pour votre application :

<https://code.visualstudio.com/docs/python/environments>

Une fois *flask* installé dans cet environnement virtuel, on peut coder un programme simple avec 2 routes (URL) :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Page d\'accueil'

@app.route('/etudiants/')
def about():
    return 'Page etudiants'

if __name__ == "__main__":
    app.run(debug=True)
```

Après avoir lancé **Flask** et l'exécution du programme en Python, on peut se connecter à ce serveur en **local** sur le port **5000** :



Page d'accueil



Page etudiants

Intégration finale :

Pour réaliser notre API, l'application backend va utiliser flask pour router les requêtes http vers des fonctions spécifiques, et mysql pour exécuter dans chacune de ces fonctions la requête SQL appropriée.

Ainsi, pour récupérer la totalité de la table etudiant dans un flux JSON, on routera la requête HTTP <http://127.0.0.1:5000/etudiants> vers la fonction **getEtudiants()** qui présente maintenant les données sous forme de dictionnaire (clé:valeur) :

```
@app.route('/etudiants/', methods=['GET'])
def getEtudiants():
    etudiants = []
    request = "SELECT * FROM etudiant"
    cursor.execute(request)
    result = cursor.fetchall()
    for row in result:
        etudiant = {
            "idetudiant": row[0],
            "nom": row[1],
            "prenom": row[2],
            "email": row[3],
            "telephone": row[4]
        }
        etudiants.append(etudiant)
    return jsonify(etudiants), 201
```

Pour ne récupérer les données que d'un seul étudiant, nous écrirons une nouvelle fonction **getEtudiants(id)** qui reçoit en paramètre l'id de cet étudiant, qui sera appelée par la requête HTTP <http://127.0.0.1:5000/etudiant/{id}>

```
@app.route('/etudiant/<int:id>', methods=['GET'])
def getEtudiant(id):
    request = "SELECT * FROM etudiant WHERE idetudiant = " + str(id)
    print (request)
    cursor.execute(request)
    row = cursor.fetchone()
    etudiant = {
        "idetudiant": row[0],
        "nom": row[1],
        "prenom": row[2],
        "email": row[3],
        "telephone": row[4]
    }
    return jsonify(etudiant)
```

Travail demandé : compléter l'API avec les routes et les fonctions permettant de réaliser tous les cas d'utilisation du CRUD (*annexe 2*) :

- Ajouter un étudiant
- Voir les détails d'un étudiant
- Modifier les détails d'un étudiant
- Supprimer un étudiant

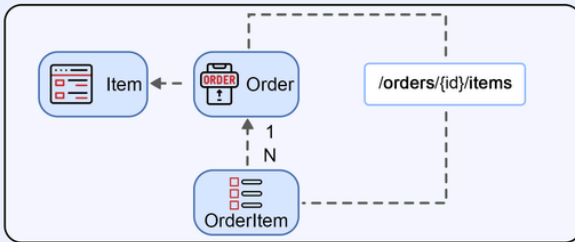
*** Améliorations finales ***

- Gestion des privilèges :
 - Créer un utilisateur **user** avec le mot de passe **bt5@c13l972** pour tout hôte (%) ayant les privilèges nécessaires pour les requêtes **SELECT**, **INSERT**, **UPDATE**, et **DELETE**
 - Tester cet utilisateur en exécutant l'API depuis un autre poste à partir de son adresse IP dans le réseau local
- Versionning (annexe 5) :
 - Chaque amélioration majeure de l'API fera l'objet d'une nouvelle version (v1, v2, etc.)
 - Le fichier python devra être enregistré avec ce numéro de version dans le nom du fichier (api_v1.py, api_v2.py, etc.)
 - Les routes mentionneront la version de l'API utilisée dans l'URL (<http://127.0.0.1/v1/etudiants/>)
- Codes de retour HTTP (annexe 4) :
 - Écrire une nouvelle version de l'API qui utilise correctement les codes de retour HTTP
- Routage des requêtes :
 - Après avoir demandé au prof de créer une **entrée NAT** dans le routeur (box SFR), tester la connexion à votre API depuis votre téléphone portable, après avoir désactivé le wifi, afin de l'obliger à passer par votre connexion de données cellulaires (4G) et entrer dans le réseau local de l'extérieur
- Gestion des erreurs
 - Écrire une nouvelle version de l'API qui gère correctement les erreurs possibles (ex : id inexistant)
 - <https://www.youtube.com/watch?v=sNJYSPNJIAc>

Top 8 Tips For Restful API Design



1 Domain Model Driven



2 Choose HTTP Methods

GET	✓	For querying list to reading detail
POST	✓	For creating
PUT	✓	For updating
DELETE	✓	For deleting
PATCH and Others	✗	Can be confused with some actions

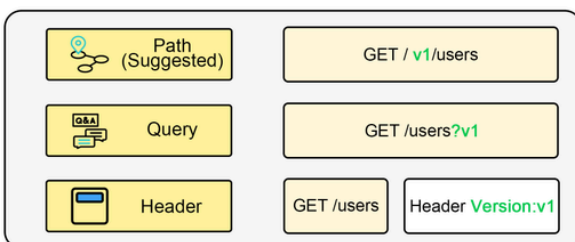
3 Implement Idempotence Properly



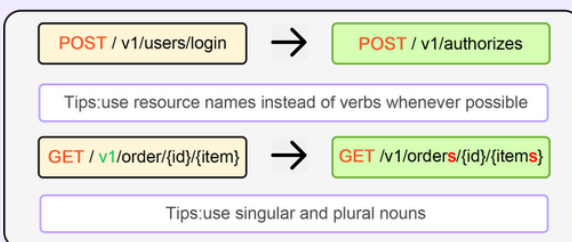
4 Choose HTTP Status Codes

201	Created Successfully	405	Method Not Supported
400	Data Validation Failed	409	Business Rule Conflict
401	User Not Authenticated	415	Unsupported Data Request Format
403	Permission Check Failed	500	Internal Server Error
200	Request Successful	404	Resource Not Found

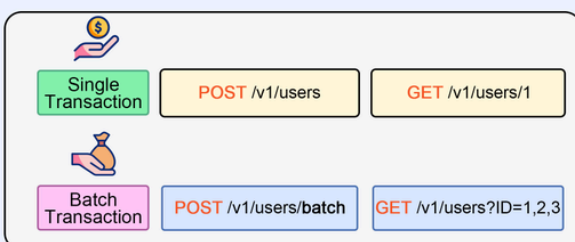
5 Versioning



6 Semantic Paths



7 Batch Processing



8 Query Language

