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.

Résumé du 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)

```python
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)
```

2. *flask* : Serveur web

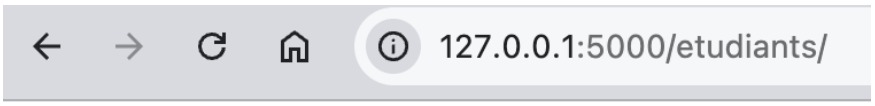
Il faut tout d'abord créer un environnement virtuel pour votre application : https://code.visualstudio.com/docs/python/environments

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

```python
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 de *flask* et de *mysql* :

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) :

```python
@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 ***getEtudiant(id)*** qui reçoit en paramètre l'id de cet étudiant, qui sera appelée par la requête HTTP http://127.0.0.1:5000/etudiants/{id}

```python
@app.route('/v1/etudiants/<int:id>', methods=['GET'])
def getEtudiant(id):
    req = f"SELECT * FROM etudiant WHERE idetudiant = {id}"
    print (req)
    cursor.execute(req)
    row = cursor.fetchone()
    etudiant = {
        "idetudiant": row[0],
        "nom": row[1],
        "prenom": row[2],
        "email": row[3],
        "telephone": row[4]
    }
    return jsonify(etudiant), 200
```

Détail des versions et du travail demandé

(***api_v1.py***) :

Compléter l'API avec les routes et les fonctions permettant de réaliser tous les cas d'utilisation du CRUD (*cheat-sheet #2*)

Rappel des cas d'utilisation :

Requête pour voir tous les étudiants de la table			
Méthode HTTP	**Route**	**Fonction associée**	**Paramètres**
GET	**/v1/etudiants/**	getEtudiants()	N/A
Réponses possibles			
Code HTTP	**Données / Message** en format JSON		
200 **400** **401** **500**	Tableau contenant tous les étudiants Requête invalide Accès non autorisé Échec de connexion à la base de données		

Requête pour voir un seul étudiant			
Méthode HTTP	**Route**	**Fonction associée**	**Paramètres**
GET	**/v1/etudiants/{id}**	getEtudiant(id)	id de l'étudiant dans la route
Réponse			
Code HTTP	**Données / Message** en format JSON		
200 **400** **401** **404** **500**	Détails de l'étudiant en format JSON Requête invalide Accès non autorisé id invalide Échec de connexion à la base de données		

Requête pour ajouter un étudiant			
Méthode HTTP	**Route**	**Fonction associée**	**Paramètres**
POST	**/v1/etudiants/**	addEtudiant()	Détails de l'étudiant en format JSON dans le body
Réponse			
Code HTTP	**Données / Message** en format JSON		
201 **401** **500**	Ajout OK Accès non autorisé Échec de connexion à la base de données		

Requête pour modifier les détails d'un étudiant			
Méthode HTTP	**Route**	**Fonction associée**	**Paramètres**
PUT	**/v1/etudiants/{id}**	updateEtudiant(id)	id de l'étudiant dans la route Détails de l'étudiant en format JSON dans le body
Réponse			
Code HTTP	**Données / Message** en format JSON		
200 **401** **404** **500**	Modification OK Accès non autorisé id invalide Échec de connexion à la base de données		

Requête pour supprimer un étudiant			
Méthode HTTP	**Route**	**Fonction associée**	**Paramètres**
DELETE	**/v1/etudiants/{id}**	deleteEtudiant(id)	id de l'étudiant dans la route
Réponse			
Code HTTP	**Données / Message** en format JSON		
200 **401** **404** **500**	Suppression OK Accès non autorisé id invalide Échec de connexion à la base de données		

Points à respecter :

- Gestion des f-strings en Python :
  - Créer les requêtes SQL en utilisant les ***f-strings***
  - https://www.docstring.fr/glossaire/f-string/

- Versioning (*cheat-sheet #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/)

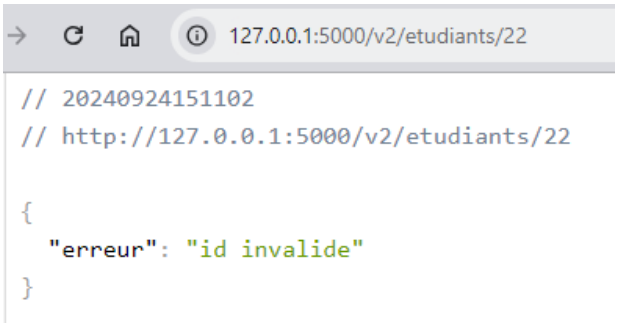
- Codes de retour HTTP (*cheat-sheet # 4*) :
  - Utiliser correctement les codes de retour HTTP

(***api_v2.py***) :

1. Gestion des erreurs :
    a. Écrire une nouvelle version de l'API (v2) qui gère correctement les erreurs possibles (ex : id inexistant)
    https://www.youtube.com/watch?v=sNJYSPNJIAc

- Affichage d'une erreur non gérée :

- Affichage d'une erreur gérée :



- Code modifié :

```python
@app.route('/v2/etudiants/<int:id>', methods=['GET'])
def getEtudiant(id):
    req = f"SELECT * FROM etudiant WHERE idetudiant = {id}"
    print (req)
    try :
        cursor.execute(req)
        row = cursor.fetchone()
        etudiant = {
            "idetudiant": row[0],
            "nom": row[1],
            "prenom": row[2],
            "email": row[3],
            "telephone": row[4]
        }
        return jsonify(etudiant), 200
    except TypeError :
        return jsonify({'erreur':'id invalide'}), 404
```

2. Créer un utilisateur ***extern_user*** avec le mot de passe ***bt5@c13l972*** pour tout hôte (%) ayant les privilèges nécessaires pour les requêtes **SELECT**, **INSERT**, **UPDATE**, et **DELETE**

3. Tester les accès de cet utilisateur en exécutant l'API depuis un poste à partir de son adresse IP dans le réseau local (dissociation du serveur de l'API et du serveur de bases de données)

4. Sauvegarder les fichiers Python et la base de données sur Github

(***api_v3.py***) :

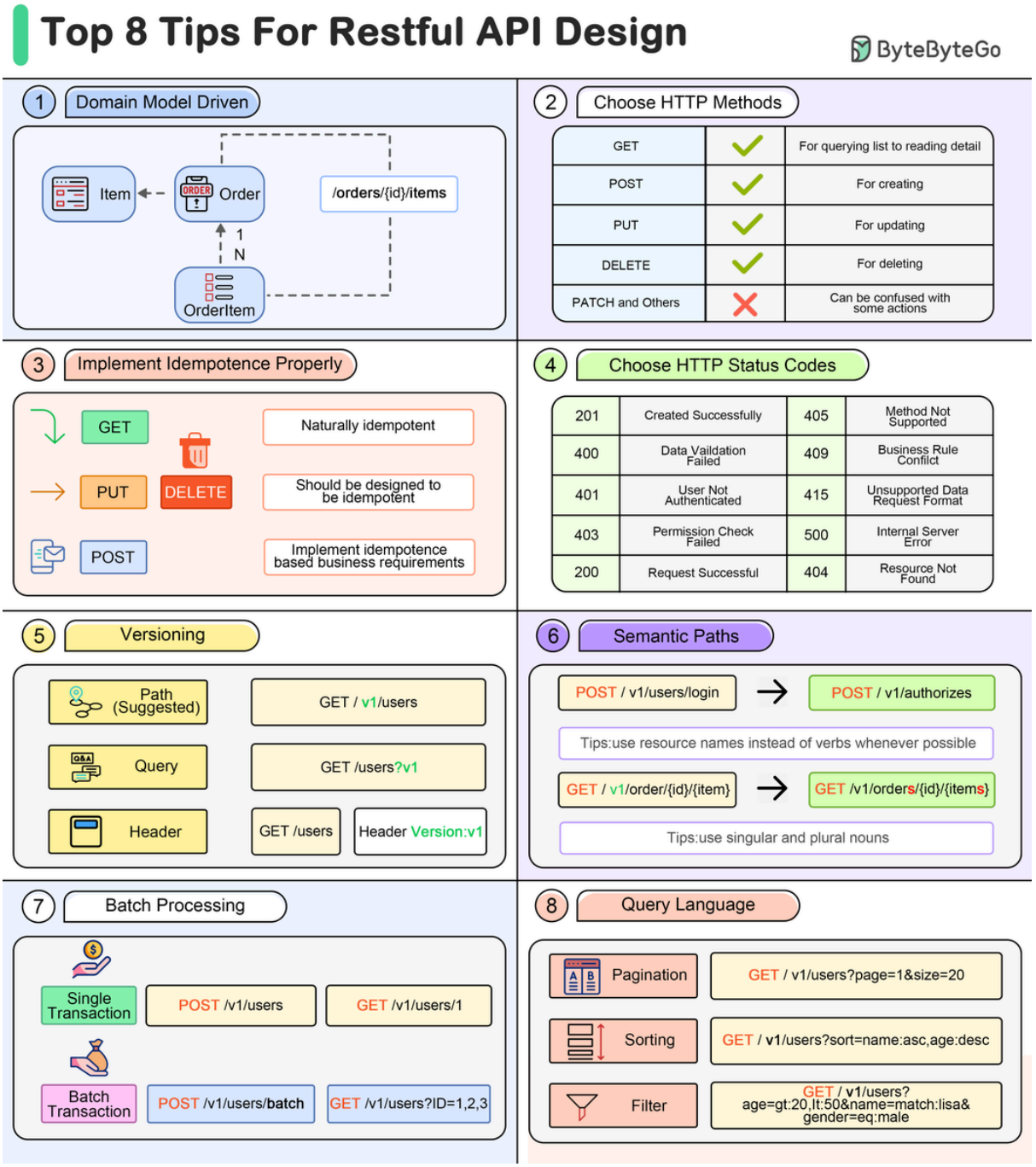
1. Créer une nouvelle table user dans la base ciel2025 :
2. Créer un utilisateur (login: user1 / password: 123456)
3. Créer dans le code de l'api une nouvelle route /login/ avec la méthode POST qui lance la fonction login()
4. Tester avec Postman la récupération des identifiants
5. Rechercher dans la table user le mot de passe de l'utilisateur
6. Ajouter le code permettant de tester le résultat de la recherche
7. Transférer toutes les fonctions de requêtage dans un fichier db.py et les transformer en méthodes d'une classe **Database**

Rédiger un rapport contenant :

- Une description du projet à l'aide d'un diagramme de cas d'utilisation
- Une description de l'API avec la présentation détaillée des requêtes HTTP (utiliser Swagger : https://swagger.io/)
- Un récapitulatif des 3 versions de l'API
- Un diagramme de classes pour la classe **Database** généré par **pyreverse**
- Une fiche des tests de l'API réalisés avec **Postman** par un autre étudiant
- Une conclusion comprenant les améliorations possibles de l'API en terme de **cybersécurité**

*** À compléter ***

Cheat-sheet de conception d'API :



Cheat-sheet des commandes SQL :
https://www.rameshfadatare.com/cheat-sheet/cheat-sheet-for-mysql-database-commands/

*** Améliorations finales ***

- 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

- Automatisation des tests

- Injections SQL

- Injections XSS

- Historique des versions :
    - v1 :
        - Tous les **cas d'utilisation** sont implémentés
        - Utilisation du **versioning**
        - Utilisation d'un **code HTTP** et d'un message de retour appropriés
        - Utilisation des **f-strings**

    - v2 :
        - Gestion des erreurs avec **try** / **except**
        - Gestion des **privilèges**
        - Début des sauvegardes avec **Github**

    - v3 :
        - Gestion des utilisateurs identifiés avec **username / password**
        - Passage à la programmation modulaire et à la **POO**