Documentation : Apprenez à programmer des applications Python avec le framework Flask

1. Flask : c’est quoi ?

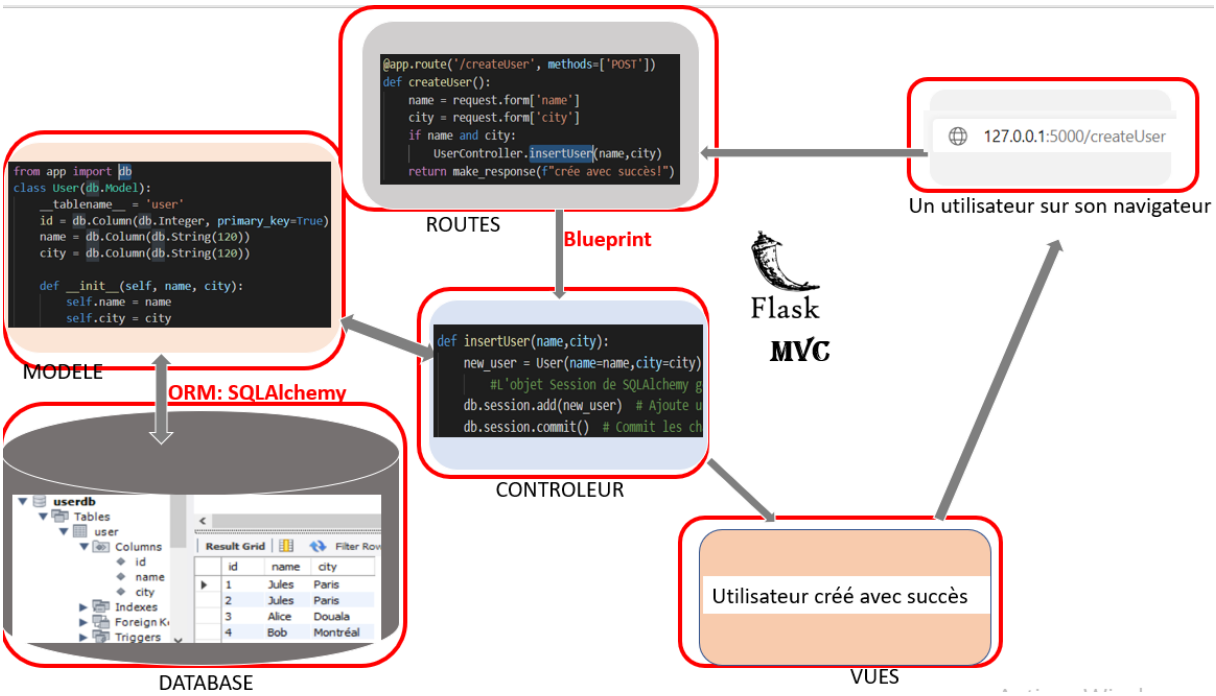
Flask est un micro **framework open-source** de développement web en **Python**. Il est classé comme **microframework** car il est très léger. Flask a pour objectif de garder un noyau simple mais extensible. Il n'intègre pas de système d'authentification, pas de couche d'abstraction de base de données, ni d'outil de validation de formulaires. Cependant, de nombreuses extensions permettent d'ajouter facilement des fonctionnalités

Il fournit des outils et des fonctionnalités utiles qui facilitent la création d'applications Web en Python. Il offre de la flexibilité aux développeurs et constitue un cadre plus accessible pour les nouveaux développeurs, car vous pouvez créer rapidement une application Web à l'aide d'un seul fichier Python.

2. C’est quoi le modèle MVC ?

L'approche **Modèle**-Vue-Contrôleur est une approche de conception de systèmes informatiques permettant de diviser en modules logiques une application comportant des interfaces graphiques, des données et de la logique (c'est-à-dire du contrôleur).

Voici l’architecture de l’application que nous allons traiter dans ce document.



1-Un utilisateur demande à afficher une page en saisissant une URL.

2-Les routes aiguillent les requêtes vers le contrôleur.

3-Le contrôleur reçoit cette demande, la traite et utilise les modèles pour récupérer toutes les données nécessaires, les organise et les envoie vers la vue.

4-La vue utilise ensuite ces données pour afficher la page Web finale présentée à l'utilisateur dans son navigateur.

3. Flask SQLAlchemy

Flask-SQLAlchemy nous fournit un **ORM** pour modifier les données d'application en créant facilement des modèles définis. Quelle que soit la base de données de votre choix, Flask-SQLAlchemy garantira que les modèles que nous créons en Python se traduiront par la syntaxe de la base de données que nous avons choisie.

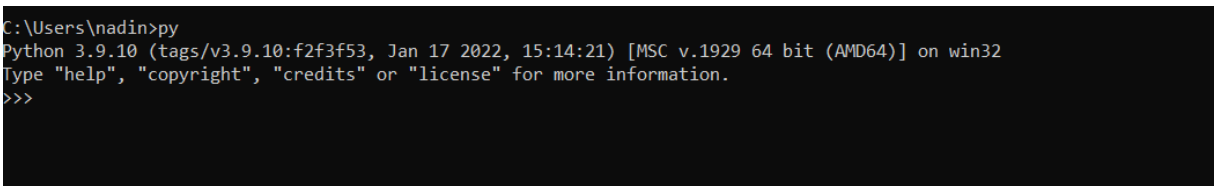
4. Télécharger Python

https://www.python.org/downloads/windows/

Naviguez vers le répertoire d'installation C:\Users*Username*\AppData\Local\Programs\Python\Pythonxx

Et double-cliquez sur python.exe pour vérifier que l'installation est bien faite :

Pour vérifier l'installation, allez sur l'invite de commande et saisissez «py » pour voir la version



Vérifier que pip est installé, c'est un outil de gestion de packages. Il permet de télécharger les paquets python

5. Configurer les variables d'environnement pour python et pip

6. Création d'un environnement virtuel

Les packages logiciels Python sont installés par défaut sur l'ensemble du système. Par conséquent, chaque fois qu'un seul package spécifique à un projet est modifié, il change pour tous vos projets Python. Avoir des environnements virtuels séparés pour chaque projet est la solution la plus simple.

Installer virtualnv :

1. Ouvrez le menu Démarrer et tapez "cmd".
2. Sélectionnez l'application Invite de commandes.
3. Tapez la commande pip suivante dans la console :

```
pip install virtualenv
```

Créer et activer l'environnement virtuel

```
cd flask-app

virtualenv --python venv

.\venv\Scripts\activate
```

Installer Flask

```
pip install flask
```

```
python -c "import flask; print(flask.__version__)"
```

7. Premiers pas avec l'application Flask

Créez un fichier bonjour.py

```
from flask import Flask


app = Flask(__name__)



@app.route('/')
def bonjour():
   return 'Bonjour tout le monde !'
```

Indiquer à Flask où trouver l'application

```
set FLASK_APP=bonjour.py
```

Exécuter en mode développement

```
set FLASK_ENV=development
```

Lancer Flask

```
flask run
```

Note : Utilisez la variable d'environnement FLASK_APP pour pointer la commande vers votre application. Définissez FLASK_ENV=development pour qu'il s'exécute avec le débogueur et le reloader. N'utilisez jamais cette commande pour déployer publiquement, utilisez un serveur WSGI de production tel que Gunicorn, uWSGI, etc.

8. Utilisation du code HTML

Créer une application avec le fichier de base app.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
   return render_template('index.html')
```

Créer un dossier **templates** dans **flask-app** et y créer un fichier **index.html** avec le contenu suivant :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Mon application Flask</title>
</head>
<body>
   <h1>Bonjour tout le monde et Bienvenu !! </h1>
</body>
</html>
```

Créer un dossier static et un sous-dossier css-style et ajouter y un fichier `style.css`

```
h1 {
    border: 4px #eee solid;
    color: blue;
    text-align: center;
    padding: 15px;
}
```

Modifier le fichier index.html pour y ajouter le chemin d'accès au style css

```
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename= 'css-style/style.css') }}">
    <title> Mon application Flask </title>
</head>
```

La fonction d'aide `url_for()` permet de générer l'emplacement du fichier.

9. Création d'une connexion à une base de données mysql

Configuration de la chaine de connexion

Note : Utilisez pip install flask_mysqldb si jamais il n'est pas installé.

```
from flask import Flask,render_template, request
from flask_mysqldb import MySQL

app = Flask(__name__)
#Configuration de la base de données
app.config['MYSQL_HOST'] = 'localhost'
app.config['MYSQL_USER'] = 'root'
app.config['MYSQL_PASSWORD'] = 'root123'
app.config['MYSQL_DB'] = 'customermodels'
#Création d'une instance de connexion
mysql = MySQL(app)
```

Configuration du curseur

Juste avec la configuration ci-dessus, nous ne pouvons pas interagir avec les tables de base de données. Pour cela, nous avons besoin de ce qu'on appelle un curseur.

```
cursor = mysql.connection.cursor()
```

Cursor permet donc à Flask d'interagir avec les tables de base de données. Il peut analyser les données de base de données, exécuter différentes requêtes SQL et supprimer les enregistrements de table.

Utilisation de Cursor pour l'exécution d'une requête

La requête va être exécutée sur la table **customers**.

Liste des propriétés de la table **customers** :

```
customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
```

```
state
postalCode
country
salesRepEmployeeNumber
creditLimit
```

```
#Créer un cursor de connexion
        cursor = mysql.connection.cursor()
        #Exécuter la requete qui insère un client dans la base de données
        cursor.execute("INSERT IGNORE INTO customers
(customerNumber,customerName,contactFirstName,phone,addressLine1,addressLine2,
city,state,postalCode,country,salesRepEmployeeNumber,creditLimit) VALUES
(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)",(customerNumber,customerName,contactFir
stName,phone,addressLine1,addressLine2,city,state,postalCode,country,salesRepE
mployeeNumber,creditLimit))

#Enregistrer l'action sur la DB
 mysql.connection.commit()
        cursor.close()
```

Modifier le fichier index.html pour ajouter le formulaire de création d'un « customers »

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename= 'css-
style/style.css') }}">
    <title>Mon application Flask</title>
</head>
<body>
   <h1>Bonjour tout le monde et Bienvenu !! </h1>

   <form action="/créer" method = "POST">
    <p>customerNumber <input type = "integer" name = "customerNumber" /></p>
    <p>customerName <input type = "text" name = "customerName" /></p>
    <p>contactLastName <input type = "text" name = "contactLastName" /></p>
    <p>contactFirstName <input type = "text" name = "contactFirstName" /></p>
    <p>phone <input type = "text" name = "phone" /></p>
    <p>addressLine1 <input type = "text" name = "addressLine1" /></p>
    <p>addressLine2 <input type = "text" name = "addressLine2" /></p>
    <p>city <input type = "text" name = "city" /></p>
    <p>state <input type = "text" name = "state" /></p>
    <p>postalCode <input type = "text" name = "postalCode" /></p>
    <p>country <input type = "text" name = "country" /></p>
```

```
    <p>salesRepEmployeeNumber <input type = "integer" name =
"salesRepEmployeeNumber" /></p>
    <p>creditLimit <input type = "decimal" name = "creditLimit" /></p>
    <p><input type = "submit" value = "Submit" /></p>
 </form>
   <p>Ceci est une application Flask qui utilise la fonction
render_template... </p>

</body>
</html>
```

Dans le fichier app.y, ajouter la fonction qui récupère les données du formulaire à l'aide de l'objet request et fait une insertion en base de données.

```
@app.route('/créer', methods = ['POST', 'GET'])
def créer():

    if request.method == 'POST':
        customerNumber = request.form['customerNumber']
        customerName = request.form['customerName']
        contactFirstName = request.form['contactFirstName']
        phone = request.form['phone']
        addressLine1 = request.form['addressLine1']
        addressLine2 = request.form['addressLine2']
        city = request.form['city']
        state = request.form['state']
        postalCode = request.form['postalCode']
        country = request.form['country']
        salesRepEmployeeNumber = request.form['salesRepEmployeeNumber']
        creditLimit = request.form['creditLimit']
        #Créer un cursor de connexion
        cursor = mysql.connection.cursor()
        #Exécuter la requete qui insère un client dans la base de données
        cursor.execute("INSERT IGNORE INTO customers
(customerNumber,customerName,contactFirstName,phone,addressLine1,addressLine2,
city,state,postalCode,country,salesRepEmployeeNumber,creditLimit) VALUES
(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)",(customerNumber,customerName,contactFir
stName,phone,addressLine1,addressLine2,city,state,postalCode,country,salesRepE
mployeeNumber,creditLimit))

        mysql.connection.commit()
        cursor.close()
        return f"Client enregistré!!"
```

Exécuter l'application

10. Ajouter à notre application la fonctionnalité qui affiche la liste des clients

Pour afficher la liste des clients, nous allons :

- Ajouter à notre fichier app.py une fonction « afficherClient ». Dans cette fonction, nous implémentons la connexion à la base de données, avec exécution d'une requête qui renvoie la liste des noms des clients. Cette liste est passée en paramètre à la fonction render_template() pour être affichée sur la page html affiche.html.

```
@app.route('/afficherClient')
def afficherClient():
    cursor = mysql.connection.cursor()
    cursor.execute("SELECT customerName FROM customers")
       # output = cur.fetchone()
    mysql.connection.commit()
    output = cursor.fetchall()
    cursor.close()
    return render_template("affiche.html", data = output)
```

- Créer un fichier affiche.html

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename= 'css-
style/style.css') }}">
    <title>Mon application Flask</title>
</head>
<body>
   <h1>Afficher les clients </h1>

   {% block content %}

   <!-- Liste des clients -->

   {% for row in data %}
       <p>{{row[0]}}</p>
   {% endfor %}


   {% endblock %}

</body>
</html>
```

Cette fonction parcours la liste **data** et affiche chaque élément qui représente le nom d'un client en base de données.

11. Réalisation d'une application MVC Flask

Nous allons reprendre notre application précédente, mais en implémentant une architecture MVC(Modèle-Vue-Contrôleur).

Nous allons nous assurer d'avoir l'architecture ci-dessous. Allons dans le dossier du projet et créons les fichiers suivants :

```
app/
|
├── templates/
| └── index.html
  └── affiche.html
|
├── routes/
| └── user_route.py
|
├── models/
| └── User.py
|
├── controllers/
| └── UserController.py
|
├── app.py
└── config.py
```

Etape 1 : Installer le package de l'ORM SQLAlchemy pour flask, appelé flask-sqlalchemy

```
pip install flask-sqlalchemy
```

Etape 2 : Création du model

On crée un objet de classe SQLAlchemy. Il fournit également une classe parent **Model** qui l'utilise pour déclarer un modèle défini par l'utilisateur. Ci-dessous, le modèle User est créé.

```
from flask_sqlalchemy import SQLAlchemy
from app import db
class User(db.Model):
    __tablename__ = 'user'
    id = db.Column(db.Integer, primary_key=True)
```

```
    name = db.Column(db.String(120))
    city = db.Column(db.String(120))

    def __init__(self, name, city):
        self.name = name
        self.city = city
```

Etape 3 : Configuration de la connexion à la base de données

Configuration de la chaîne de connexion à la base de données dans le fichier config.py

Importer d'abord pymsql (l'installer si ce n'est pas encore fait) avec `pip install PyMySql.`

`Le fichier config.py`

```
import os
import pymysql
SECRET_KEY = os.urandom(32)
# Indique le dossier dans lequel scripts s'exécute
basedir = os.path.abspath(os.path.dirname(__file__))
# Activer le mode debug
DEBUG = True
# Connexion à la base de données
SQLALCHEMY_DATABASE_URI = 'mysql+pymysql://root:root123@localhost:3306/userdb'
# désactiver le système d'évènement/warning de Flask-SQLAlchemy
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Etape 4 : Créer la base de données

Le modèle est créé à travers la commande `db.create_all()`

Les différentes méthodes appliquées sur la base de données sont :

```
# Insertion
db.session.add (model object)
```

```
# suppression
db.session.delete (model object)
```

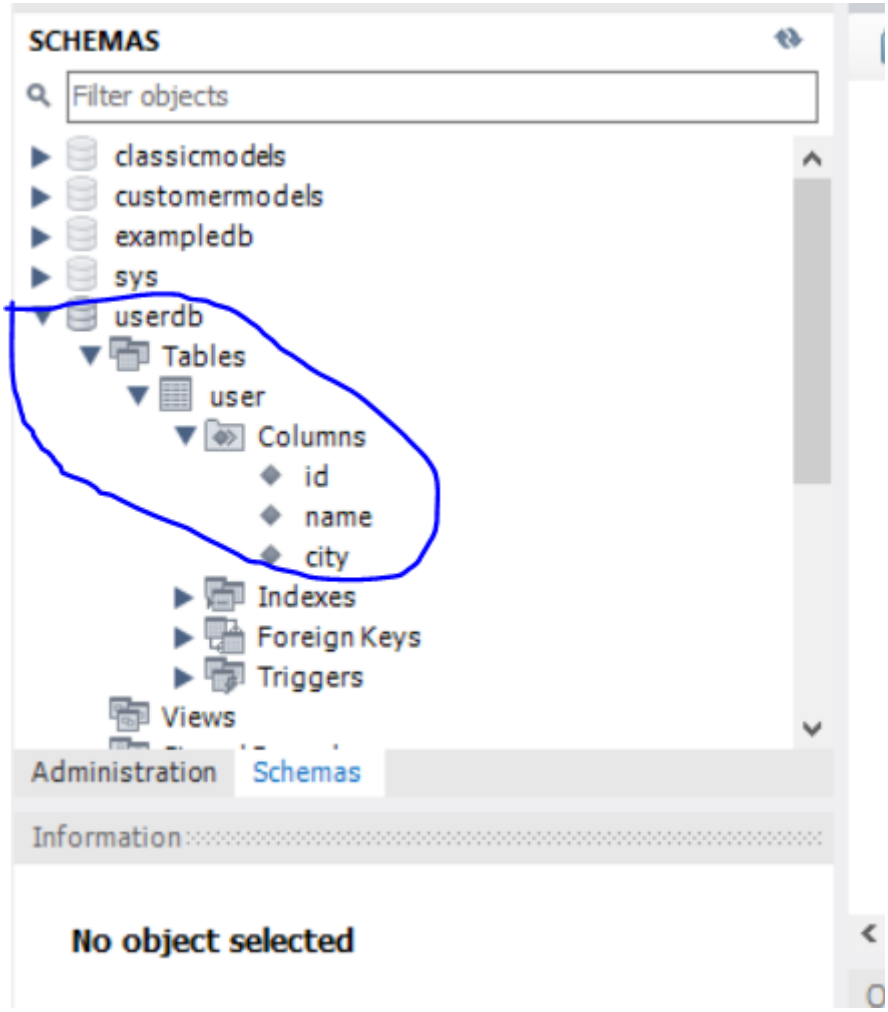
```
# lecture
model.query.all ()
```

Etape 5 : Configuration du fichier d'application `app.py`

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
#associer le fichier de configuration
app.config.from_object('config')
db = SQLAlchemy(app)
#Importer le modèle User
from models.User import User
#Créer le modèle
db.create_all()
if __name__ == '__main__':
    db.create_all()
    app.debug = True
    app.run()
    app.run()
```

Exécuter l'application avec `Flask run` pour créer la base de données, après avoir configuré la variable d'environnement `set FLASK_APP=app.py` Et `set FLASK_ENV=development`.

Résultat en base de données



Etape 6 : définition des fonctions du contrôleur dans UserController.py

Le contrôleur définit l'ensemble des méthodes/fonctions à définir. Par exemple, nous allons définir la fonction qui renvoie la page index, la fonction qui crée un utilisateur, la fonction qui affiche la liste des utilisateurs en base de données.

Il s'agit ici de définir le corps des fonctions qui appelées dans user_route.py. Pour exemple, nous avons la fonction insertUser() qui crée un utilisateur en base de données et la fonction getUser() qui retourne la lise des utilisateurs en base de données.

```
import sys
from flask import render_template, redirect, url_for, request
from models.User import User
from app import db



def getUser():
    user_data = User.query.all()
    for user in user_data:
        print(user.name)
    return user_data


def insertUser(name,city):
    new_user = User(name=name,city=city) #instanciation d'un objet de type
User
        #L'objet Session de SQLAlchemy gère toutes les opérations de
persistance pour les objets ORM.
    db.session.add(new_user)  # Ajoute un nouvel utilisateur à la base de
données
    db.session.commit()  # Commit les changements
```

Etape 7 : définition des routes dans user_route.py

La définition des routes permet de séparer les contrôleurs/fonctions et les routes. Pour ce faire:

- Dans le fichier user_route.py, on importe l'objet app pour pouvoir utiliser le décorateur(@app.route), le contrôleur UserController.py pour pouvoir utiliser les fonctions ou méthodes définies dans ce contrôleur.

- Importer l'objet Blueprint. Il permet de rendre le code encore plus modulable, car on définit les routes ailleurs que dans app.py. De plus, si nous avons plusieurs modèles (exemple : User et Product), on peut créer des fichiers séparés pour les routes liées aux modèles User(user_route.py) et Product(product_route.py)
- Définir les routes associées aux méthodes
- Créer un objet de type Blueprint, ex : `user_bp`. Le paramètre
- Maintenant, pour que l'application puisse retrouver ces routes, l'objet blueprint `user_bp` doit être enregistré dans l'application principale app.py.

Contenu du fichier user_route.py

```
from flask import Blueprint
from flask import render_template, request, make_response
from controllers import UserController
#importer app pour pouvoir utiliser @app.route
from app import app

#créer un objet blueprint pour le modèle User.
user_bp = Blueprint('user_bp', __name__)
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/afficherUser', methods=['GET'])
def afficherUser():
    user_data=UserController.getUser()
    return render_template("affiche.html", data = user_data)

@app.route('/createUser', methods=['POST'])
def createUser():

    #id = request.form['id']
    name = request.form['name']
    city = request.form['city']

    if name and city:
        UserController.insertUser(name,city)
    return make_response(f"crée avec succès!")
```

Voici le code actuel de app.py dans lequel on doit importer l'objet user_bp qui a été créé et enregistrer cet objet dans l'application app.

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)
#associer le fichier de configuration
app.config.from_object('config')

db = SQLAlchemy(app)
#db.init_app(app) #Add this line Before migrate line

from models.User import User
#Créer le modèle
db.create_all()

from routes.user_route import user_bp
#enregistrer l'objet user_bp
app.register_blueprint(user_bp, url_prefix='/users')
print(user_bp)


if __name__ == '__main__':

    app.debug = True
    app.run()
```

Le paramètr url_prefix est optionnel, mais il est utile pour permettre à Blueprint de bien séparer les routes. Par exemple, pour le modèle Product, on aura aussi des routes et en écrivant :

```
app.register_blueprint(product_bp, url_prefix='/products'), Blueprint parvient à faire une distinction parfaite des routes qu'il gère.
```

Etape 9 : Création des vues index.html et affiche.html

Ici, nous allons reprendre les mêmes fichiers que nous avions dans la première partie de ce cours et nous allons les adapter avec les propriétés que nous avons pour la classe User, c'est-à-dire name et city.

Index.html where we create user

```html
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename= 'css-style/style.css') }}">
    <title>Mon application Flask</title>
```

```
</head>
<body>
   <h1>Bonjour tout le monde et Bienvenu !! </h1>

   <form action="/createUser" method = "POST">
    <p>name <input type = "text" name = "name" /></p>
    <p>city <input type = "text" name = "city" /></p>
    <p><input type = "submit" value = "Submit" /></p>
 </form>
   <p>Ceci est une application Flask qui utilise la fonction
render_template... </p>

</body>
</html>
```

Affiche.html : là où nous affichons la liste des utilisateurs

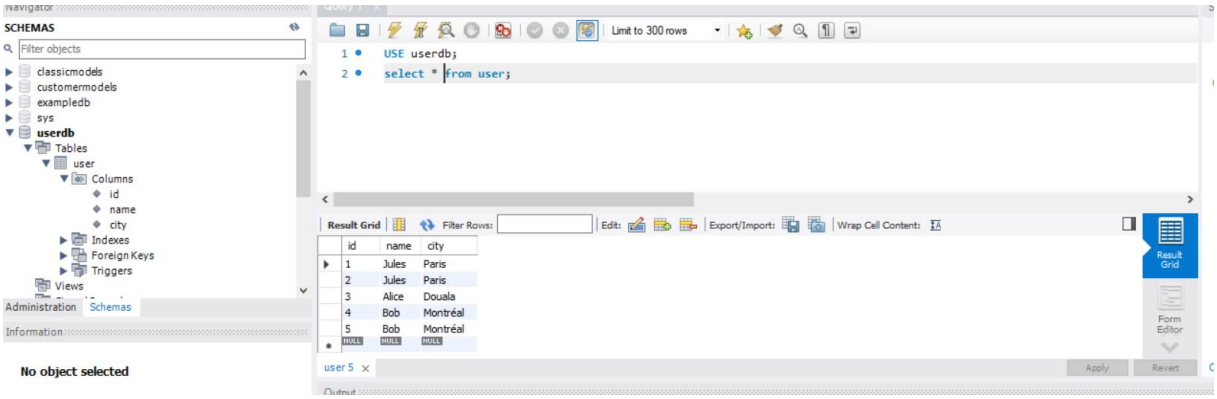
```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="{{ url_for('static', filename= 'css-
style/style.css') }}">
    <title>Mon application Flask</title>
</head>
<body>
   <h1>Afficher les Utilisateurs </h1>

   {% block content %}

   <!-- Liste des clients -->

   {% for row in data %}
       <p>{{row[0]}}</p>
   {% endfor %}


   {% endblock %}

</body>
</html>
```

Executer application

Aller à l'adresse http://127.0.0.1:5000/

Saisir “Bob” et “Montréal” et cliquer sur Submit, ou alors saisissez ce que vous voulez.

Faire ensuite un select sur la base de données pour se rassurer que l’utilisateur a été créé.



Aller à la route http://127.0.0.1:5000/afficherUser pour afficher la liste des utilisateurs.

Vous pouvez ajouter des tests pour éviter qu’un utilisateur ne soit créé deux fois.