I Introduction

Le but de ce projet est de construire une application web permettant d'interroger une base de données.

1 Installer Flask

Pour gérer le serveur web, on va utiliser le module Python « Flask ».
Il s'agit donc dans un premier temps d'installer le module Flask.
Flask permet de mettre en place un serveur web et c'est aussi un frameworks :
Wikipédia : *Un framework est un ensemble d'outils et de composants logiciels organisés conformément à un plan d'architecture et des patterns, l'ensemble formant ou promouvant un « squelette » de programme, un canevas.*

2 Premiers exemples

1. Créer un fichier `run.py` et coller le code suivant.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello world !"

if __name__ == "__main__":
    pass #écrire la commande app.run() à la place de pass
```

   Attention : Dans le programme, écrire `app.run()` à la place de `pass`. Je suis obligé de ne pas lancer la commande pour pouvoir compiler mon document.
2. Dans une console, se déplacer dans le répertoire contenant `run.py` et exécuter `python3 run.py` pour démarrer le service.
   Dans la console, on devrait avoir confirmation du démarrage du service, et son url.
3. Dans un navigateur, ouvrir l'adresse donnée dans la console (port inclus). C'est à priori `http://127.0.0.1:5000/`

II Organiser le projet

On ne va pas tout coder dans un seul fichier. On va donc dès maintenant organiser l'arborescence du projet.
Il faut notamment séparer la partie requête sql qui représente l'application (le modèle) de la partie affichage dans le navigateur (les vues), le tout étant organisé et mis en oeuvre par un contrôleur.

1 Organisation des répertoires

Dans le répertoire contenant `run.py`, créer un répertoire nommé `Projet`.
Dans le répertoire `Projet` créer les répertoires suivant :

- `static` : qui contiendra tout ce qui n'est pas géré dynamiquement (feuille de style, script, images,...)
- `db` : qui contiendra la BDD.
- `templates` : qui contiendra les fichiers HTML
- `modele` : qui contiendra les fichiers contenant les requêtes SQL

Dans le répertoire `static`,

- Créer un répertoire `CSS`
- et un répertoire `js`

Dans `CSS` créer un fichier, pour l'instant vide `style.css`

2 Création de la structure

1. Dans le répertoire `Projet` créer un fichier `__init__.py`.
   Ce fichier permet à Python de considérer le répertoire `Projet` comme un paquet (et de faire des importations, voir plus loin).
2. Dans le répertoire `Projet` créer un fichier `control.py` (qui contiendra les routes et les actions à engager) et y copier

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello world !"
```

3. Modifier le fichier `run.py` pour qu'il ne contienne plus que l'appel à l'application.

```
from Projet import app

if __name__ == "__main__":
    pass #Ecrire a nouveau app.run()
```

4. Ouvrir `__init__.py` situé à la racine de `Projet` et ajouter

```
from .control import app
```

   C'est cela qui permet l'importation de app depuis `run.py`.
   Le point devant control représente « ici » (comme dans les autres chemins)

Tester à nouveau en relançant `run.py`

3 Afficher une vraie page html

La fonction `index` dans le contrôleur renvoie le contenu de la page qui va être affiché dans le navigateur lorsque le client appelle la page à la racine du site.
Toutes les pages du site doivent être correctement formées (au point de vue html) et l'on ne va pas écrire tout le contenu de la page html textuellement après le `return` dans `control`.

1. Dans le répertoire `templates`, créer un fichier `index.html` de contenu

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <link rel="stylesheet" type="text/css"
             href="{{url_for('static',filename='CSS/style.css')}}">
        <script src="{{url_for('static',filename='js/mesFonctions.js')}}"></script>
    </head>
    <body>
        <p>Bonjour</p>

    </body>
</html>
```

2. Pour faire afficher le fichier `index.html` lors d'une requête à la racine, on va utiliser dans `control.py` la fonction `render_template` du module flask.
   Il faut donc l'importer.
   Dans `control.py`, compléter `from flask import Flask` par `from flask import Flask, render_template`
   Dans `control.py`, remplacer `return "Hello world !"` par `return render_template('index.html')`

4 Création de la base de données

a Python et sqlite

Consulter le document r_Python_et_sqlite.

b Création de la bibliothèque

En Python ou à l'aide de SQLiteBrowser, créer le fichier `livres.db` à partir du fichier `livres.sql` et le placer dans le répertoire `db`.

5 Créer de nouvelle route et interroger la BDD

Pour pouvoir afficher la liste des livres, il y a 4 choses à réaliser qui seront détaillées dans a, b, c et d.

1. Mettre un lien ou un bouton sur la page d'accueil pointant sur une nouvelle route que l'on doit définir dans le fichier `control.py`
2. Définir la route dans `control.py`
3. Obtenir la liste des livres dans la base de données : Ce sera le travail d'un modèle, une fonction que l'on définira dans le fichier `modele.py` et qui sera appelée dans `control.py`
4. Dans `control.py`, récupérer les données et les transmettre à une template pour affichage.

a Mettre un lien sur la page d'accueil

Pour éviter du dupliquer du code HTML, on va utiliser le moteur de template Jinja

1. Renommer le fichier `index.html` en `base.html`
2. Éditer `base.html` et entre les balises `body`, insérer :
   `{% block content %}{% endblock %}`

3. Ouvrir un fichier index.html l'éditer en :
   ```
   {% extends 'base.html' %}

   {% block content %}
   <a href='/afficher/livre/'>Afficher les livres</a>
   {% endblock %}
   ```
   Redémarrer le serveur et consulter le site.
   Le controlleur donne l'instruction d'afficher `index.html`
   `index.html` charge la page `base.html` mais en remplaçant le contenu du bloc de nom content par ce qui a été précisé dans la page `index.html`.
   On peut définir de nombreux blocs si l'on veut.
   Pour modifier la structure de toutes les page, par exemple en rajoutant un header et un footer, il suffit de modifier `base.html`.

b Définir la route

Dans le fichier `control.py` rajouter après la définition de route « / » et de la fonction `index` :

```
@app.route('/afficher/livre/')
def afficher_livre():
    liste_livre = selectionner_livre()
    return render_template('affiche_livre.html', liste_des_livres = liste_livre)
```

et rajouter dans les importations :
`from .modele.model import selectionner_livre`
Avec liste_des_livres = liste_livre on aura accès à la variable liste_des_livres lors de la création du template affiche_livre.html

c Définir selectionner_livre() dans le fichier `modele.py`

1. Dans le répertoire `modele`, créer le fichier `model.py`.
   Dans ce fichier on va coder les fonctions d'interrogation de la BDD
2. Crée aussi un fichier vide `__init__.py` pour que le répertoire modele soit considéré comme un paquet.
3. Dans `model.py`, on doit donc importer `sqlite3` et définir une fonction de connexion à la base.

   ```
   import sqlite3

   def ouvrir_connexion():
       cnx = None
       try:
           cnx = sqlite3.connect('projet/db/livres.db')
       except BaseException as e:
           print(e)
       return cnx
   ```

   Le chemin démarre à projet car l'application est exécutée à partir de `run.py` et c'est donc à partir de cet endroit que sont définis les chemins.
4. On définit la fonction selectionner_livre() :

   ```
   def selectionner_livre():
       cnx = ouvrir_connexion()
       cur = cnx.cursor()
       cur.execute("SELECT * FROM livre")
       rows = cur.fetchall()
       cnx.close()
       return rows
   ```

d Définir le template affiche_livre.html

1. Dupliquer `index.html` et renommer le clone en `affiche_livre.html`
2. Modifier le contenu du bloc content en :
   `<p>{{liste_des_livres}}</p>`
   Le fait d'entourer par une double accolade permet d'afficher le contenu de la variable.
   Tester en redémarrant le serveur.
3. ça marche mais on va faire mieux. Jinja contient des structure itérative, des structures alternatives.
4. Modifier le bloc content de `affiche_livre.html` en :

   ```
   <table>
       <tr><th>Titre</th><th>Éditeur</th><th>Année</th><th>ISBN</th></tr>
   {% for ligne in liste_des_livres %}
   <tr><td>{{ligne[0]}}</td><td>{{ligne[1]}}</td><td>{{ligne[2]}}</td><td>{{ligne[3]}}</td></tr>
   {% endfor %}
   </table>
   ```

5. Tester.

Jinja permet beaucoup de choses, voir la documentation.

III Mettre à jour la base de données

Ceci est une version très simplifié pour insérer des données dans la base. Dès que l'on utilise des données fournies par l'utilisateur, il faut se méfier et les vérifier afin de ne pas corrompre la base. Il devrait y avoir des contrôles de formulaire (par exemple en javascript avant d'envoyer les données à la base, ce que nous n'allons pas faire.

a Créer le formulaire

On va réaliser un formulaire afin de pouvoir inscrire un nouveau livre.

1. Dans le content block de `index.html`, ajouter :
   `<a href = /ajouter/livre/>Ajouter un livre à la base</a>`
2. Dans `control.py` ajouter la route

   ```
   @app.route('/ajouter/livre/')
   def ajouter_livre():
       return render_template('ajouter_livre.html')
   ```

   qui va afficher le formulaire de saisie de nouveau livre.
3. Créer dans template un fichier `ajouter_livre.html` et le compléter comme ci-dessous :
   Voir pour mieux comprendre le document html_formulaire.
   De plus, les label permettent de rattacher un texte à un élément de formulaire.

   ```
   {% extends 'base.html' %}

   {% block content %}
   <h1>Ajouter un livre à la base</h1>
   <form action="/admin/enregistrer\_livre" method="POST">
       <fieldset>
           <legend>Saisie d'un livre</legend>
           <label for='idTitre'>Titre</label>
           <input type="text" name="titre" id="idTitre"><br>
           <label for='idEditeur'>Editeur</label>
           <input type="text" name="editeur" id="idEditeur"><br>
           <label for='idAnnee'>Année</label>
           <input type="text" name="annee" id="idAnnee"><br>
           <label for='idISBN'>ISBN</label>
           <input type="text" name="isbn" id="idISBN"><br>
           <input type="submit" name="envoyer">
       </fieldset>
   </form>
   {% endblock %}
   ```

Tester l'affichage de la page du formulaire.
Lorsque l'utilisateur clique sur le bouton « envoyer » la route `/admin/enregistrer_livre/` est appelée avec 4 couples de clés valeurs dans le corps de la requête. Les clés sont les noms des éléments du formulaire et les valeurs les valeurs saisies.

b Récupérer les données et les insérer dans la base

Là aussi, avant d'insérer les données, on devrait les vérifier

1. Définir la route `@app.route('/admin/enregistrer_livre/', methods=['POST'])`

   ```
   @app.route('/admin/enregistrer_livre/', methods=['POST'])
   def enregistrer_livre():
       resultat = request.form    #permet de récupérer les données POST
                                  #sous forme d'un dictionnaire resultat
       mettre_a_jour_table_livre(resultat)    #appel le modèle à utiliser
       return redirect(url_for('index'))    #redirige sur l'affichage de la page d'index
   ```

Attention :

Penser à ajouter `redirect`, `url_for` et `request` dans les import en provenance de Flask et `mettre_a_jour_table_livre` dans les import en provenance de model

2. Définir la fonction `mettre_a_jour_table_livre`

```
def mettre_a_jour_table_livre(resultat):
    cnx = ouvrir_connexion()
    cur = cnx.cursor()
    cur.execute("INSERT INTO livre VALUES (?, ?, ?, ?)",(resultat['titre'], \
                    resultat['editeur'], resultat['annee'], resultat['isbn']))
    cnx.commit()
    cnx.close()
```

3. Tester d'entrer le livre :
   - Titre : Mon Livre
   - Editeur : Buisson
   - Année : 2022
   - ISBN : 0-2022
4. Consulter la liste des livres pour constater l'entrée.