Projets

Projets pratiques à réaliser seul ou en équipe pour apprendre à maîtriser les bases de données.

niveau

Même si la structure des sites à réaliser est semblable, chaque élève a reçu des données différentes avec des structures différentes. L'énoncé ci-dessous utilise le terme générique "ENTITÉ" pour faire référence au **concept principal** attribué à chaque élève. Entité peut donc faire référence aussi bien à

- restaurant
- cinéma
- hôtel
- école supérieure ...

... en fonction du contexte.

Énoncé

Ton objectif est de développer un site dynamique en Python avec Flask, Jinja2 et SQLite3 pour afficher et rechercher des entités. Ces entités sont toutes situées dans des communes.

Ces consignes représentent les exigences minimum à réaliser. Tu peux toujours aller plus loin si tu en as envie.

Schéma

Identifie les différentes entités de ton modèle afin de **créer un schéma entité-associations** avec les entités, les associations et les cardinalités.

Pour le schéma, essaie d'aller le plus loin possible dans ta modélisation en identifiant bien les informations redondantes dans les données.

Pour le code Python et la réalisation de la DB, tu peux normalement te contenter des 2 tables principales (communes et ...).

Import initial

Tu vas devoir créer un programme qui permet d'enregistrer les données dans la base de données. Ce programme va devoir convertir le fichier d'origine au format CSV ou JSON et insérer les données dans la base de données. Pour ce faire, tu vas pouvoir utiliser ChatGPT ou un autre programme d'intelligence artificielle. Tu devras bien lui demander de respecter le schéma de données que tu as analysé. Sinon, il aura tendance à ne créer qu'une seule table, ce qui n'est pas correct.

Dans un premier temps, il n'est pas nécessaire de comprendre parfaitement le code généré par l'IA. Mais d'un point de vue logique la première étape consiste à:

1. **créer le schéma**, donc à créer les tables
2. **insérer les données**

Tu verras que l'intelligence artificielle devrait d'abord insérer certaines entités en vérifiant si elles n'existent pas déjà en base de données. Elle effectuera ensuite des requêtes afin de relier les entités entre elles à l'aide des clés primaires et étrangères.

Tu devras pouvoir identifier l'endroit où ton entité de référence est insérée afin de pouvoir générer le slug (voir plus loin).

Site Web Dynamique

Le site consiste en:

- Un menu de navigation et un pied de page sur toutes les pages
- une page d'accueil
- une page avec la liste des communes
- une page pour afficher les informations d'une entité
- une page de recherche d'entités
- une API de recherche de communes

Templates

Organise tes templates correctement, en utilisant l'héritage.

URLs

> Attention: pour générer des chemins vers tes entités, tu vas devoir utiliser ce que l'on appelle des `SLUGS`. Ces slugs doivent être **générés au moment de l'import** des données sur base du nom de l'entité.

Un **slug** est la partie d'une URL qui est spécifique à la page/ressource que l'on publie et dont on a enlevé tous les caractères spéciaux et remplacé les espaces par des "-".

Exemple: `École de cinéma à Paris` --> `ecole-de-cinema-a-paris`

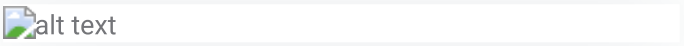
Tes urls ressembleront donc à `/cinemas/cinema-l-etoile`.

Pour faire simple et efficace, utilise le module "slugify" en pièce jointe:

```
# Exemple d'utilisation
text = "École de cinéma à Paris"
slug = slugify(text)
print(slug)  # Résultat : "ecole-de-cinema-a-paris"
```

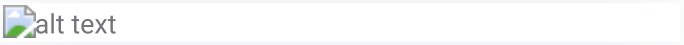
Menu de navigation

Le menu doit permettre de **naviguer sur ton site** et contenir un **champ de recherche** qui pointe vers la page de recherche.



Utilise une `navbar` de Bootstrap.

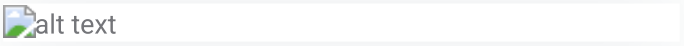
Page d'accueil



La page d'accueil doit contenir:

- le **type d'entité** proposé sur le site. Ex: "Vous trouverez sur ce site..."
- le nombre d'entités disponibles en db
- 4 entités sélectionnées au hasard affichées dans un **list group** Bootstrap (https://getbootstrap.com/docs/5.0/components/list-group/): "[Nom] à [Ville]" (avec liens).

Page "Liste des communes"



Cette page affiche la liste complète des communes avec le nombre d'entités qu'elle contient. La liste est affichée par ordre décroissant du nombre d'entités et ensuite par ordre alphabétique.

L'utilisateur doir pouvoir cliquer sur le nom d'une commune pour se rendre sur la page correspondante.

Utlise les class bootstrap suivantes: `btn`, `btn-success`, `badge`. Exemple

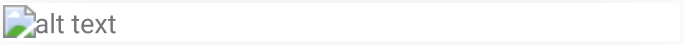
```
<button type="button" class="btn btn-primary">
  Notifications <span class="badge text-bg-secondary">4</span>
</button>
```

Page "Entités"

Cette page affiche toutes les entités dans une **table** Bootstrap, triées par ordre alphabétique.

Pour des raisons de performance, tu peux limiter les résultats aux 50 premières entités.

Page "Recherche d'entités"

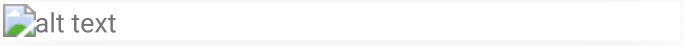


Cette page affiche toutes les entités qui correspondent à la recherche de l'utilisateur sous forme de **table** Bootstrap.

La recherche doit renvoyer toutes les entités dont le nom ou le nom de la commune **contiennent** le mot recherché, triées par ordre alphabétique.

Pour des raisons de performance, tu peux limiter les résultats aux 50 premières entités.

Page "Fiche Entité"



Cette page doit afficher les informations principales de l'entité:

- nom
- adresse
- commune
- - d'autres infos pertinentes en fonction de l'entité (site web, email...)

API de recherche de communes

Chemin: `/api/search`

Cette API est une route qui renvoie exclusivement des **données brutes, sans HTML**.

S'il y a une seule lettre, la route doit renvoyer la liste de toutes les communes qui commencent par cette lettre. Exemple: "J" --> **J**odoigne, **J**emappe... S'il y a plusieurs lettres, la route doit renvoyer toutes les communes qui contiennent la chaine de caractères. Exemple: "cour" --> Rac**cour**, In**cour**t...

Les communes doivent être triées par ordre alphabétique.

Votre route DOIT SIMPLEMENT retourner la **liste des noms des communes** (par exemple `return communes`):

```
def recherche(critere):

    communes = get_communes(critere) # Ex: ["Incourt", "Jodoigne", "Rixensar

    return communes

```

Astuces

Caractères spéciaux

Certains fichiers de données contiennent des caractères spéciaux encodés sous la forme `\u00EF19`. C'est notamment le cas des caractères accentués. A ma connaissance, ChatGPT ne sait pas comment les traiter correctement.

Lors de mes tests, j'ai trouvé que le plus simple est de remplacer tous ces caractères à la main directement dans le fichier de données avec VS Code avant de l'importer. Sélectionne le code à remplacer, `ctrl-f2` (curseurs multiples) et tape le caractère de remplacement.

Executer les requêtes SQL en python

Le plus simple est de créer 2 fonctions qui retournent les résultats d'une requête passée en paramètre:

- une fonction qui renvoie une liste de résultats
- une fonction qui renvoie UN SEUL résultat

> 💡 Idéalement, tu pourrais créer une fonction pour chaque requête et les placer dans un module séparé pour rednre ton code plus lisible.

Retourner plusieurs résultats

```python
# Fonction générique pour exécuter une requête SELECT
def select(query, params=()):
    conn = sqlite3.connect("data/cinemas.db")
    conn.row_factory = sqlite3.Row  # Permet de récupérer les résultats sou
    cursor = conn.cursor()
    cursor.execute(query, params)
    rows = cursor.fetchall()
    conn.close()
    return [dict(row) for row in rows]
```

> Remarque que la connexion est créée et fermée directement dans la fonction.

Récupérer un seul résultat

La fonction `select` renvoie toujours une liste. Quand tu sais que ta requête ne va renvoyer qu'un seul résultat (`count()`...) tu peux utiliser cette version qui utilise `fetchone()` au lieu de `fetchall()`

```python
# Fonction générique pour exécuter une requête SELECT
def select(query, params=()):
    conn = sqlite3.connect("data/cinemas.db")
    conn.row_factory = sqlite3.Row  # Permet de récupérer les résultats sou
    cursor = conn.cursor()
    cursor.execute(query, params)
    rows = cursor.fetchone()
    conn.close()
    return dict(row) if row else None
```

Exemple d'appel de ces fonctions:

```python
age = 18
resultats = select("SELECT * FROM student WHERE age=?", (age,))
```

Requête avec paramètres en Python

Il est possible, et même **conseillé** d'utiliser des requêtes SQL avec paramètres. Pour ce faire, remplace la valeur à rechercher/utiliser par un `?` (appelé **place holder**) et fournis la valeur en paramètre lors de l'exécution de la requête:

```python
# Requête avec un paramètre
student_name = "Alice"
query = "SELECT * FROM students WHERE name = ?"
cursor.execute(query, (student_name,))  # Note la virgule pour le tuple
```

Le paramètre passé à `execute` doit être un tuple (type de variable en Python). S'il n'y a qu'**un seul paramètre** à passer, **il faut ajouter la virgule**.

```python
# Requête avec plusieurs paramètres
age = 20
grade = "A"
query = "SELECT * FROM students WHERE age > ? AND grade = ?"
cursor.execute(query, (age, grade))  # Paramètres dans un tuple
```