Objectifs de la feuille

- Introduction SGBD
- ORM SQLAlchemy
- Modèle
- Création et utilisation de commandes
- Console Flask et Shell

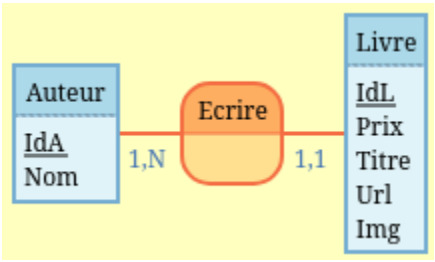
Introduction

Dans le chapitre précédent vous avez appris à installer Flask, organiser votre projet, lancer un serveur avec Flask et à configurer un projet. Avant de voir comment afficher la page index.html, créons une base de données dans laquelle nous implémenterons notre modèle. Et pourquoi ne pas commencer par l'affichage de la page index.html ? Afin de proposer à l'utilisateur des informations, nous devons être en mesure d'en chercher. Or, comment trouver une information qui n'est enregistrée nulle part ? Il est donc préférable de commencer par la création de la base de données et des différents éléments qu'elle contiendra.

Vous savez tous ce qu'est une base de données : un immense système où les données sont organisées en tables, lignes et colonnes. Nous pouvons chercher des données, en créer, en modifier ou en supprimer. Il existe plusieurs types de bases de données. Plus précisément, ce sont plutôt des Systèmes de Gestion de Bases de Données (SGBD) répartis en deux grandes familles :

- SGBD relationnels : les données sont représentées dans des tableaux pouvant être liés les uns avec les autres,
- SGBD NoSQL : les données ne sont pas structurées en tableaux mais autrement : graphe, documents, clé/valeur...

Dans ce cours nous utiliserons SQLite, un SGBD relationnel, mais il en existe d'autres tels que MySql, PostgreSQL, Oracle, etc. J'ai choisi SQLite car il est libre et facile à apprendre. La base de données contient des tables qui elles-mêmes contiennent des champs et des enregistrements. Avant de créer une table, il est important d'en connaître la structure de notre base de données. Voici à quoi ressemblera la base de données de notre projet dont voici le MCD :



Ce MCD se base sur la norme MERISE, un langage graphique permettant de modéliser la structure d'une base de données.

Pour interagir avec un SGBD relationnel, nous utilisons un langage appelé SQL (Structured Query Language). Ce langage permet d'ajouter, modifier ou supprimer des données mais aussi d'interroger la base selon certains critères et faire des recoupements d'information en suivant les relations entre les tables. Une requête SQL peut ressembler à ceci :

```
SELECT * FROM Auteur
```

Ceci affichera tous les enregistrements de la table Auteur. Mais ne vous inquiétez pas : vous n'aurez pas besoin d'apprendre un nouveau langage. Nous allons utiliser Flask pour l'interroger à notre place ! Avant d'interroger une base SQLite avec Flask, vous devez installer le logiciel SQLite en exécutant la commande :

```
$ sudo apt install sqlite3
```

Créez un nouveau document à la racine de votre projet : monApp.db. Il s'agira de notre base de données. C'est tout ! Vous n'avez plus rien à faire. Revenons maintenant à Flask et voyons comment interroger la base de données.

Installation de l'ORM SQLAlchemy

Je vous ai dit précédemment que vous n'aviez pas besoin d'apprendre SQL car Flask intégrait un outil qui le faisait pour vous. J'ai un peu enjolivé la réalité, j'espère que vous ne m'en voudrez pas trop ! Par défaut, Flask ne gère pas les bases de données. Vous devez installer une extension, SQLAlchemy, qui fera le pont entre la base de données et votre application. Plus spécifiquement, vous écrirez des requêtes en Python et SQLAlchemy les traduira en SQL. Puis elle vous renverra les résultats de la requête sous la forme d'objets Python avec lesquels vous pourrez interagir. Un peu comme un traducteur automatique !

C'est ce que nous appelons un ORM (Object Relational Mapping, ou Mapping objet-relationnel en français). Alors, comment utiliser SQLAlchemy ? Commencez par l'installer en utilisant pip :

```
$ pip install flask-sqlalchemy
```

Puis modifiez le fichier config.py en ajoutant ces lignes :

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'monApp.db')
```

Cela permettra à l' ORM de savoir où se situe notre base de données pour pouvoir l'interroger. Puis il faut l'activer dans app.py comme les autres plugins en ajoutant le code suivant :

```
# Create database connection object
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy(app)
db.init_app(app)
```

Ne fonctionne pas avec SQLAlchemy(app)

Avant de connecter l'application à la base de données, créons un premier modèle.

Définition du modèle

Un modèle représente la donnée qui est stockée en base. Il indique à l'ORM la structure que vous souhaitez pour chaque table ainsi que les associations. Vous devez également indiquer le type de chaque champ : est-ce un entier, une chaîne de caractères, un booléen ?
Avec SQLAlchemy, une table SQL correspond à une classe Python ; les lignes de la tables correspondent à des instances de cette classe. Dans nos données, il y a 2 modèles qui nous intéressent : les auteurs et les livres.

La première table Auteur contient 2 champs :

- idA : identifiant unique de l'enregistrement. Il s'agit d'un entier et il est très fortement conseillé de l'indiquer pour tout nouvel enregistrement que vous créez.
- Nom : il s'agit des informations sur l'auteur. Il s'agit d'une chaîne de 100 caractères maximum.

Créez un nouveau fichier models.py dans le dossier monApp. Il contiendra toutes les tables que vous souhaitez créer ainsi que leur structure.Vous allez remplacer le code dans models.py par celui-ci :

```
from .app import db

class Auteur(db.Model):
    idA = db.Column( db.Integer, primary_key=True )
    Nom = db.Column( db.String(100) )

    def __init__(self, Nom):
        self.Nom = Nom
```

Un modèle représente la structure de l'objet qui sera stocké dans la base de données. Dans Flask, le modèle de chaque objet est représenté par une classe dans le fichier models.py. La classe hérite de db.Model et peut donc utiliser les méthodes de la classe parent. Deux parties sont à distinguer dans un modèle :

- la structure de l'enregistrement dans la base de données,
- les attributs d'instance exposés.

L'appel à db.Column crée une nouvelle colonne dans la table. Cela permet de filtrer les résultats d'une recherche par le contenu de cette colonne, en utilisant par exemple la méthode que nous utiliserons plus tard telle que TableName.query.filter_by(column_name=filter).first().

La méthode __init__, elle, construit l'instance. Chaque attribut exposé à cet endroit pourra être modifié par la suite de la même manière que n'importe quelle instance de classe Python. Dans notre exemple, vous pouvez ainsi créer un nouvel élément Auteur dans la base en spécifiant son nom mais pas son id.

La deuxième table Livre contient 5 champs :

- idL : identifiant unique de l'enregistrement. Il s'agit d'un entier et il est très fortement conseillé de l'indiquer pour tout nouvel enregistrement que vous créez.
- Prix : il s'agit du prix du livre. Il s'agit d'un nombre réel
- Titre : il s'agit du titre du livre. Il s'agit d'une chaîne de 255 caractères maximum.
- Url : il s'agit de l'url nous emmenant vers les informations du livre. Il s'agit d'une chaîne de 255 caractères maximum.
- Img : il s'agit du nom de l'image de la couverture du livre. Il s'agit d'une chaîne de 255 caractères maximum.

C'est à vous de jouer !

Je vous invite à créer la seconde classe à l'image de la table Livre dans le fichier models.py

Relations Many-To-One

On aimerait aussi avoir un champ dans chaque livre qui nous permette d'accéder directement à son auteur. De même, on aimerait avoir pour chaque auteur une sorte de champs nous permettant d'accéder à ses livres. Pour cela, SQLAlchemy nous permet de déclarer des "relations." Cette déclaration requiert 2 éléments :

- une foreign key
- la relation utilisant cette foreign key.

Vous ajouterez les champs suivants dans le modèle Livre :

```
auteur_id = db.Column (db.Integer , db.ForeignKey ("auteur.idA") )
auteur = db.relationship ("Auteur", backref =db.backref ("livres", lazy="dynamic") )
```

Création de la base de donnée

Pour l'instant nous n'avons pas encore de base de données. Nous allons écrire un script pour créer la base de données, créer les tables, et les peupler avec notre jeu de données. Nous allons le faire en ajoutant une commande loaddb à notre appli (en plus des commandes run et shell).

Dans un nouveau fichier commands.py dans MonApp, vous allez ajouter le code ci-dessous :

```
import click, logging as lg
from .app import app, db

@app.cli.command()
@click.argument('filename')
def loaddb(filename):
    '''Creates the tables and populates them with data.'''

    #  création de toutes les tables
    db.drop_all()
    db.create_all()

    # chargement de notre jeu de données
    import yaml
    with open(filename, 'r') as file:
        lesLivres = yaml.safe_load(file)

    # import des modèles
    from . models import Auteur,Livre

    # première passe : création de tous les auteurs
    lesAuteurs = {}
    for livre in lesLivres :
        auteur = livre["author"]
        if auteur not in lesAuteurs :
            objet = Auteur(Nom=auteur)
            db.session.add(objet)
            lesAuteurs[auteur] = objet
    db.session.commit ()

    # deuxième passe : création de tous les livres
    for livre in lesLivres :
        auteur = lesAuteurs[livre["author"]]
        objet = Livre(Prix=livre["price"],
                      Titre=livre["title"],
                      Url=livre["url"],
                      Img=livre["img"],
                      auteur_id = auteur.idA)
        db.session.add(objet)
    db.session.commit()
    lg.warning('Database initialized!')
```

Au passage, notez deux nouvelles instructions :

- db.session.add() : Cette méthode permet d'ajouter un enregistrement à la base. En paramètres, vous passez l'instance de l'objet que vous voulez créer.
- db.session.commit() : Chaque création est ajoutée dans une session. Lorsque vous avez terminé d'ajouter des éléments, vous devez indiquer à SQLAlchemy de faire les requêtes dans la base pour finaliser l'opération.

Les sessions de SQLAlchemy permettent de gérer les transactions SQL, autrement dit un ensemble de requêtes. Si l'une d'elles échoue, l'ensemble de la transaction est annulée et aucune requête n'est communiquée à la base. Il s'agit d'un système très utile pour gérer des séquences d'opérations dépendantes les unes des autres.

Dans __init__.py, il faut aussi importer commands.py et models.py pour que nos commandes et nos modèles deviennent connus dès que l'application démarre :

```
from .app import app, db
import monApp.views
import monApp.commands
import monApp.models
```

Vous devriez pouvoir vérifier qu'en effet il y a une nouvelle commande : $ flask

```
Usage: flask [OPTIONS] COMMAND [ARGS]...

  A general utility script for Flask applications.
  An application to load must be given with the '--app' option, 'FLASK_APP'   environment variable, or with a
'wsgi.py' or 'app.py' file in the current  directory.

Options:
  -e, --env-file FILE       Load environment variables from this file, taking precedence over those set by '.env'
                            and '.flaskenv'. Variables set directly in the environment take highest precedence.
                            python-dotenv must be installed.
  -A, --app IMPORT          The Flask application or factory function to load, in the form 'module:name'. Module
                            can be a dotted import or file path. Name is not required if it is 'app', 'application',
                            'create_app', or 'make_app', and can be 'name(args)' to pass arguments.
  --debug / --no-debug      Set debug mode.
  --version                 Show the Flask version.
  --help                    Show this message and exit.

Commands:
        loaddb              Creates the tables and populates them with data.
        routes              Show the routes for the app.
        run                 Run a development server.
        shell               Run a shell in the app context.
```

Pour invoquer cette nouvelle commande, vous devez donc lui passer le chemin vers le fichier contenant les données. Ce fichier est le fichier data.yml qui se trouve dans CELENE. Pour charger du YAML en Python, vous devez installer le package PyYAML. Voici comment faire :

```
$ pip install pyyaml
```

Personnellement, je l'ai mis le fichier yaml dans le sous-répertoire monApp/data/ :

```
$ flask loaddb monApp/data/data.yml
```

Le nouveau fichier myapp.db contenant la base de données a été créé. Si vous voulez recréer la base de données, il suffit d'effacer ce fichier et de ré-invoquer la commande loaddb.

Découverte de la console Flask et du shell

Nous pourrions utiliser le débogueur de Python, mais je vais plutôt vous montrer comment lancer la console interactive de Flask. Vous allez utiliser le shell de Flask pour vous familiariser avec l'utilisation des modèles pour faire des requêtes. Dans votre console, exécutez la commande suivante :

```
$ flask shell
```

Une console s'ouvre :

```
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux
App: monApp.app
Instance: /home/dalaigre/Documents/BUT INFO/R3.01/WEB_FLASK/TutoFlask/instance
>>>
```

Python attend vos instructions. Vous avez accès à quelques objets :

```
>>> app
<Flask 'monApp.app'>

>>> app.config
<Config {'DEBUG': False, 'TESTING': False, …}
```

Vous devez retrouver vos variables du fichier config.py en cherchant bien…Utilisons cette console pour interagir avec la base. Importez les modèles et commençons :

```
>>> from monApp.models import *
>>> Auteur.query.all( )
```

La méthode query.all( ) renvoie tous les enregistrements de la table Auteur. En réponse, nous voyons que nous avons bien une liste d'auteurs. Ceci n'est pas très lisible, alors quittez le shell avec la méthode exit( ), quit( ) ou Ctrl+D. Puis, retournez dans le fichier models.py.
Ajoutez la méthode suivante à la classe Auteur :

```
def __repr__ (self ):
        return "<Auteur (%d) %s>" % (self.idA , self.Nom)
```

et celle-ci à la classe Livre :

```
def __repr__ (self ):
        return "<Livre (%d) %s>" % (self.idL , self.Titre)
```

Retentons le shell : $ flask shell

```
>>> from monApp.models import *
>>> Auteur.query.all( )
[< Auteur(1) Fabio M. Mitchelli >, ...]
```

Ahh ! c'est nettement mieux !
Voyons quels sont les livres coûtant moins de 5 euros :

```
>>> Livre.query.filter (Livre.Prix < 5.0). all( )
[<Livre (8) Obsessions Intimes (Les Chroniques Kr...>, …]
```

Obtenons un livre étant donné son id (clé primaire) :

```
>>> Livre.query.get(8)
<Livre (8) Obsessions Intimes (Les Chroniques Kr...>
```

Servons-nous de la relation Many-to-One pour obtenir les livres de Robin Hobb. Filtrons dans un premier temps la table Auteur:

```
>>> Auteur.query.filter(Auteur.Nom =="Robin Hobb")
<flask_sqlalchemy.query.Query object at 0x7abdb8182ba0>
```

Pas terrible ce résultat. De la même manière que nous avons utilisé la fonction all( ) plus haut pour obtenir l'ensemble des résultats, nous allons utilisé la fonction one( ) nous permettant d'obtenir l'unique résultat de cette requête.

```
>>> Auteur.query.filter(Auteur.Nom =="Robin Hobb").one( )
<Auteur (4) Robin Hobb>
```

Voilà, nous avons l'auteur. Mais nous voulons ses livres. Cela tombe bien, nous avons crée la relation dans le modèle Livre. Regardez dans le fichier models.py, de Livre à Auteur, le chemin retour de cette relation de référence backref porte le nom de "livres". Allons-y, sans oublier la fonction all( ) pour un meilleur affichage :

```
>>> Auteur.query.filter(Auteur.Nom =="Robin Hobb").one().livres.all()
[<Livre (5) Les Cités des Anciens, Tome 8 : Le pu...>, …]
```

Et si nos voulons pour finir les livres de Robin Hobb qui coûtent moins de 7 euros, je rajoute le filtre vu précédemment :

```
>>> Auteur.query.filter(Auteur.Nom =="Robin Hobb").one( ).livres.filter(Livre.Prix < 7).all( )
[<Livre (5) Les Cités des Anciens, Tome 8 : Le pu...>, <Livre (26) Les Cités des Anciens, Tome 7 : Le vo...>]
```

Nous en avons presque terminé avec le shell. Combien avons d'auteurs dans la base ?

```
>>> Auteur.query.count( )
69
```

Nous avons vu comment ajouter des items en utilisant la méthode add( ). Nous avons vu comment trouver un enregistrement en le cherchant via son identifiant unique en utilisant get( ). Voyons maintenant comment modifier un item ou le supprimer. Utile, n'est-ce pas ?

Commençons par ajouter un nouvel auteur :

```
>>> from monApp.models import db, Auteur
>>> db.session.add(Auteur(Nom="Christophe DELAIGRE"))
>>> db.session.commit()
```

Nous avons bien un auteur de plus :

```
>>> Auteur.query.count()
70
>>> Auteur.query.get(70)
<Auteur (70) Christophe DELAIGRE>
```

Nous allons modifier le nom :

```
>>> unAuteur = Auteur.query.get(70)
>>> unAuteur.Nom
'Christophe DELAIGRE'
>>> unAuteur.Nom = "Christophe DALAIGRE"
>>> db.session.commit()
>>> Auteur.query.get(70)
<Auteur (70) Christophe DALAIGRE>
```

Le nom est bien modifié. Bref, supprimons cet auteur qui n'en est pas un !

```
>>> unAuteur=Auteur.query.get(70)
>>> db.session.delete(unAuteur)
>>> db.session.commit()
>>> Auteur.query.count()
69
```

Voilà pour ce TP. Finissons avec un petit résumé de ce que vous avez vu. Une base de données est un ensemble d'informations organisé. Il en existe plusieurs types. Pour communiquer avec, il faut utiliser le langage SQL en faisant des requêtes. Flask intègre un outil qu'on appelle ORM qui permet de transformer des requêtes Python en requêtes SQL. Une table d'une base de données est représentée en Python sous la forme d'une classe dans le modèle. Vous pouvez interagir avec votre base de données en ouvrant une console Flask et en utilisant l'ORM. Votre base de données contient à présent des éléments. Voyons ensemble comment les afficher sur une page web dans le prochain chapitre !