TP n°3 — *SQLAlchemy, Modèles*

Dans ce TP vous allez prendre en main SQLAlchemy qui vous donne une façon très simple et très Pythonesque d'interfacer avec une base de donnée.

Exercice 1. Installation de `flask-sqlalchemy`

Activons notre virtualenv et installons le plugin :

```
$ source venv/bin/activate
(venv) $ pip install flask-sqlalchemy
```

Puis il faut l'activer dans `app.py` comme les autres plugins en ajoutant le code suivant :

myapp/app.py

```
import os.path
def mkpath(p):
    return os.path.normpath(
        os.path.join(
            os.path.dirname(__file__),
            p))

from flask_sqlalchemy import SQLAlchemy
app.config['SQLALCHEMY_DATABASE_URI'] = (
    'sqlite:///'+mkpath('../myapp.db'))
db = SQLAlchemy(app)
```

On importe la variable `db` aussi dans `__init__` pour la rendre dispo directement dans le module `myapp`.

Exercice 2. Mise en place des modèles

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.

Vous allez remplacer le code dans `models.py` par celui-ci (que je ne vous donne que partiellement — et qu'il vous faut donc compléter) :

myapp/models.py

```
from .app import db

class Author(db.Model):
    id         = db.Column(db.Integer, primary_key=True)
    name       = db.Column(db.String(100))

class Book(db.Model):
    id         = db.Column(db.Integer, primary_key=True)
    price      = db.Column(db.Float)
    ...
```

Ajoutez les champs pour `title`, `url` et `img`.

Relations. 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 : (1) une foreign key (2) la relation utilisant cette foreign key. Vous ajoutez les champs suivants dans le modèle `Book` :

```
    author_id = db.Column(db.Integer, db.ForeignKey("author.id"))
    author    = db.relationship("Author",
        backref=db.backref("books", lazy="dynamic"))
```

Requète. Enfin, on peut définir la requète `get_sample` qui renvoit un échantillon de 10 livres :

```
def get_sample():
    return Book.query.limit(10).all()
```

Ajoutez et commitez !

Exercice 3. 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 le nouveau fichier `commands.py`, vous allez ajouter le code ci-dessous :

myapp/commands.py

```python
import click
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.create_all()

    # chargement de notre jeu de données
    import yaml
    books = yaml.load(open(filename))

    # import des modèles
    from .models import Author, Book

    # première passe: création de tous les auteurs
    authors = {}
    for b in books:
        a = b["author"]
        if a not in authors:
            o = Author(name=a)
            db.session.add(o)
            authors[a] = o
    db.session.commit()

    # deuxième passe: création de tous les livres
    for b in books:
        a = authors[b["author"]]
        o = Book(price     = b["price"],
                 title     = b["title"],
                 url       = b["url"]  ,
                 img       = b["img"]  ,
                 author_id = a.id)
        db.session.add(o)
    db.session.commit()
```

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 :

myapp/__init__.py

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

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

```
(venv) $ flask
Usage: flask [OPTIONS] COMMAND [ARGS]...

  A general utility script for Flask applications.

  Provides commands from Flask, extensions, and the application. Load
  application defined in the FLASK_APP environment variable, or from
  wsgi.py file. Setting the FLASK_ENV environment variable to 'develo
  will enable debug mode.

    $ export FLASK_APP=hello.py
    $ export FLASK_ENV=development
    $ flask run

Options:
  --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     Runs a development server.
  shell   Runs a shell in the app context.
```

Utilisation de la nouvelle commande. Voici l'aide automatiquement disponible pour la nouvelle commande :

```
(venv) $ flask loaddb --help
Usage: flask loaddb [OPTIONS] FILENAME

  Creates the tables and populates them with data.

Options:
  --help  Show this message and exit.
```

Pour l'invoquer vous devez donc lui passer le chemin vers le fichier `data.yml`. Personellement, je l'ai mis dans le sous-répertoire `myapp` :

```
(venv) $ flask loaddb myapp/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 réinvoquer la commande `loaddb`.

Nous pouvons inspecter cette base de données par une méthode traditionelle :

```
(venv) $ sqlite3 myapp.db
SQLite version 3.8.7.2 2014-11-18 20:57:56
Enter ".help" for usage hints.
sqlite> select * from author;
1|Fabio M. Mitchelli
...
sqlite> select * from book;
1|19.9|La Compassion du Diable|http://www.amazon.fr/La-Comp...
...
```

Ajoutez et commitez ! (mais pas la base de données !)

Exercice 4. Requètes dans le shell

Vous allez utiliser le shell de flask pour vous familiariser avec l'utilisation des modèles pour faire des requètes :

```
(venv) $ flask shell
>>> from myapp.models import *
>>> Author.query.all()
[<myapp.models.Author object at 0x7f752e93e400>, ...]
```

Ceci n'est pas très lisible, alors quittez le shell et direction le fichier `models.py`.

myapp/models.py

Ajoutez la méthode suivante à la classe `Author` :

```
    def __repr__(self):
        return "<Author (%d) %s>" % (self.id, self.name)
```

et celle-ci à la classe `Book` :

```
    def __repr__(self):
        return "<Book (%d) %s>" % (self.id, self.title)
```

Retentons le shell :

```
(venv) $ flask shell
>>> from myapp.models import *
>>> Author.query.all()
[<Author (1) Fabio M. Mitchelli>, ...]
```

Ahh ! c'est nettement mieux !

Voyons quels sont les livres coûtant moins de 5 euros :

```
>>> Book.query.filter(Book.price < 5.0).all()
[<Book (8) Obsessions Intimes (Les Chroniques Kr...>, ...]
```

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

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

Dans une vue, il est souvent utile d'utiliser `get_or_404` :

```
>>> Book.query.get_or_404(8)
<Book (8) Obsessions Intimes (Les Chroniques Kr...>
>>> Book.query.get_or_404(1000)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/denys/cours/2a-web-serveur/P2/TP1/venv/lib/python3.4/si
    abort(404)
  File "/home/denys/cours/2a-web-serveur/P2/TP1/venv/lib/python3.4/si
    raise self.mapping[code](*args, **kwargs)
werkzeug.exceptions.NotFound: 404: Not Found
```

Lorsque l'objet en question n'existe pas, une réponse HTTP 404 est alors renvoyée (page non trouvée).

Les livres de Robin Hobb coûtant moins de 7 euros :

```
>>> Author.query.filter(
        Author.name=="Robin␣Hobb"
    ).one().books.filter(Book.price < 7).all()
[<Book (5) Les Cités des Anciens, Tome 8 : Le pu...>,
 <Book (26) Les Cités des Anciens, Tome 7 : Le vo...>]
```

Exercice 5.

A vous de jouer ! vous devez maintenant adapter votre application pour utiliser la base de données.