PYTHON

SQLAlchemy ORM Tutoriel pour les développeurs Python

Voyons comment utiliser SQLAlchemy ORM pour persister et interroger des données avec des applications Python.

Source : https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/
Auteur : https://auth0.com/blog/authors/bruno-krebs/
Traduction : Luc Cotton. dev@pilpoils.be

Table of Contents

TL;DR : Dans cet article, nous allons apprendre comment utiliser SQLAlchemy comme bibliothèque ORM (Object Relational Database) pour communiquer avec les moteurs de bases de données relationnelles. Tout d'abord, nous apprendrons quelques concepts de base de SQLAlchemy (comme les moteurs et les pools de connexions), puis nous apprendrons à mapper les classes Python et leurs relations avec les tables de base de données, et enfin nous apprendrons comment récupérer (interroger) les données de ces tables. Les extraits de code utilisés dans cet article peuvent être trouvés dans ce dépôt GitHub.

"Apprenez à utiliser SQLAlchemy ORM pour persister et interroger les données des applications Python."

SQLAlchemy Introduction

SQLAlchemy est une bibliothèque qui facilite la communication entre les programmes Python et les bases de données. La plupart du temps, cette bibliothèque est utilisée comme un outil de mappage relationnel objet (ORM) qui traduit les classes Python en tables sur des bases de données relationnelles et convertit automatiquement les appels de fonctions en instructions SQL. SQLAlchemy fournit une interface standard qui permet aux développeurs de créer un code agnostique de base de données pour communiquer avec une grande variété de moteurs de base de données.
Comme nous le verrons dans cet article, SQLAlchemy s'appuie sur des modèles de conception communs (comme les Object Pools) pour permettre aux développeurs de créer et d'expédier facilement des applications d'entreprise prêtes pour la production. De plus, avec SQLAlchemy, le code standard pour gérer des tâches telles que les connexions aux bases de données est abstrait pour permettre aux développeurs de se concentrer sur la logique métier.
Avant de plonger dans les fonctionnalités ORM fournies par SQLAlchemy, nous devons apprendre comment fonctionne le noyau. Les sections suivantes présentent les concepts importants que tout développeur Python doit comprendre avant de traiter avec les applications SQLAlchemy.

Python DBAPI

La DBAPI de Python (acronyme de DataBase API) a été créée pour spécifier comment les modules Python qui s'intègrent aux bases de données doivent exposer leurs interfaces. Bien que nous n'interagissions pas directement avec cette API - nous utiliserons SQLAlchemy comme façade - il est bon de savoir qu'elle définit comment les fonctions communes comme `connect`, `close`, `commit`, et `rollback` doivent se comporter. Par conséquent, chaque fois que nous utilisons un module Python qui adhère à la spécification, nous pouvons être assurés que nous trouverons ces fonctions et qu'elles se comporteront comme prévu.
Dans cet article, nous allons installer et utiliser l'implémentation de base de données PostgreSQL la plus populaire disponible : `psycopg`. D'autres pilotes Python communiquent également avec PostgreSQL, mais `psycopg` est le meilleur candidat puisqu'il implémente entièrement la spécification DBAPI et bénéficie du soutien de la communauté.
Pour mieux comprendre la spécification DBAPI, les fonctions dont elle a besoin et le comportement de ces fonctions, jetez un coup d'œil à la proposition d'amélioration Python qui l'a introduite. Aussi, pour en savoir plus sur les autres moteurs de base de données que nous pouvons utiliser (comme MySQL ou Oracle), jetez un œil à la liste officielle des interfaces de base de données disponibles.

Moteurs SQLAlchemy

Chaque fois que nous voulons utiliser SQLAlchemy pour interagir avec une base de données, nous avons besoin de créer un *moteur*. Les moteurs, sur SQLAlchemy, sont utilisés pour gérer deux facteurs cruciaux : *Pool* et *Dialects*. Les deux sections suivantes expliquent ce que sont ces deux concepts, mais pour l'instant, il suffit de dire que SQLAlchemy les utilise pour interagir avec les fonctions DBAPI.
Pour créer un moteur et commencer à interagir avec les bases de données, nous devons importer la fonction `create_engine` de la bibliothèque `sqlalchemy` et lui envoyer un appel :

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://usr:pass@localhost:5432/sqlalchemy')
```

Cet exemple crée un moteur PostgreSQL pour communiquer avec une instance fonctionnant localement sur le port `5432` (celui par défaut). Il définit également qu'il utilisera `usr` et `pass` comme identifiants pour interagir avec la base de données `sqlalchemy.` Notez que la création d'un moteur *ne* se connecte *pas* instantanément à la base de données. Ce processus est reporté au moment où c'est nécessaire (comme lorsque nous soumettons une requête, ou lorsque nous créons ou mettons à jour une ligne dans une table).
Puisque SQLAlchemy s'appuie sur la spécification DBAPI pour interagir avec les bases de données, les systèmes de gestion de bases de données les plus courants sont supportés. PostgreSQL, MySQL, Oracle, Microsoft SQL Server et SQLite sont tous des exemples de moteurs que nous pouvons utiliser avec SQLAlchemy. Pour en savoir plus sur les options disponibles pour créer des moteurs SQLAlchemy, consultez la documentation officielle.

SQLAlchemy Connection Pools

La mise en commun des connexions est l'une des implémentations les plus traditionnelles du modèle de pool d'objets. Les pools d'objets sont utilisés comme caches d'objets pré-initialisés prêts à l'emploi. En d'autres termes, au lieu de passer du temps à créer des objets qui sont fréquemment nécessaires (comme des connexions à des bases de données), le programme récupère un objet existant dans le pool, l'utilise comme il le souhaite et le remet à sa place une fois terminé.
La principale raison pour laquelle les programmes tirent profit de ce modèle de conception est d'améliorer les performances. Dans le cas des connexions aux bases de données, l'ouverture et la maintenance de nouvelles connexions sont coûteuses, prennent du temps et gaspillent des ressources. De plus, ce modèle permet de gérer plus facilement le nombre de connexions qu'une application peut utiliser simultanément.

Il existe différentes implémentations du modèle de pool de connexions disponibles sur SQLAlchemy . Par exemple, la création d'un `Engine` par la fonction `create_engine()` génère généralement un QueuePool. Ce type de pool est configuré avec des valeurs par défaut raisonnables, comme une taille maximale de pool de 5 connexions.
Comme d'habitude, les programmes prêts pour la production doivent remplacer ces valeurs par défaut (pour adapter les pools à leurs besoins), la plupart des différentes implémentations des pools de connexion offrent un ensemble similaire d'options de configuration. La liste suivante montre les options les plus courantes avec leur description :

- `pool_size` : Définit le nombre de connexions que le pool gérera.
- `max_overflow`: Spécifie le nombre de connexions excédentaires (par rapport à la `pool_size`) prises en charge par le pool.
- `pool_recycle` : Configure l'âge maximum (en secondes) des connexions dans le pool.
- `pool_timeout` : Indique combien de secondes le programme attendra avant d'abandonner l'établissement d'une connexion à partir du pool.

Pour en savoir plus sur les pools de connexion sur SQLAlchemy, consultez la documentation officielle.

SQLAlchemy Dialectes

Comme SQLAlchemy est une façade qui permet aux développeurs Python de créer des applications qui communiquent avec différents moteurs de base de données via la même API, nous devons utiliser *Dialects*. La plupart des bases de données relationnelles populaires disponibles sur le marché adhèrent à la norme SQL (Structured Query Language), mais elles introduisent également des variations propriétaires. Ces variations sont les seules responsables de l'existence des *Dialects*.
Par exemple, disons que nous voulons récupérer les dix premières lignes d'un tableau appelé `people`. Si nos données étaient conservées par un moteur de base de données Microsoft SQL Server, SQLAlchemy devrait émettre la requête suivante:

```
SELECT TOP 10 * FROM people;
```

Mais, si nos données étaient persistantes sur l'instance MySQL, alors SQLAlchemy devrait émettre :

```
SELECT * FROM people LIMIT 10;
```

Par conséquent, pour savoir précisément quelle requête émettre, SQLAlchemy doit connaître le type de base de données qu'il traite. C'est exactement ce que font les *Dialects.* Ils font prendre conscience à SQLAlchemy du dialecte dont il a besoin pour parler.

SQLAlchemy comprend la liste suivante de dialectes :

- Firebird
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- Sybase

Les dialectes pour d'autres moteurs de base de données, comme Amazon Redshift, sont supportés comme des projets externes mais peuvent être facilement installés. Consultez la documentation officielle sur SQLAlchemy Dialects pour en savoir plus.

SQLAlchemy ORM

ORM, qui signifie *Object Relational Mapper*, est la spécialisation du modèle de conception du *Data Mapper* qui s'adresse aux bases de données relationnelles comme MySQL, Oracle et PostgreSQL. Comme l'explique Martin Fowler dans l'article, *Mappers* est responsable du déplacement des données entre les objets et une base de données tout en les gardant indépendantes les unes des autres. Comme les langages de programmation orientés objet et les bases de données relationnelles structurent les données de différentes manières, nous avons besoin de code spécifique pour traduire d'un schéma à l'autre.
Par exemple, dans un langage de programmation comme Python, nous pouvons créer une classe `Product` et une classe `Order` pour relier autant d'instances que nécessaire d'une classe à une autre (c'est-à-dire que `Product` peut contenir une liste d'instances `Order` et vice versa). Cependant, sur les bases de données relationnelles, nous avons besoin de trois entités (tables), une pour persister les produits, une autre pour persister les commandes, et une troisième pour relier (par clé étrangère) produits et commandes.
Comme nous le verrons dans les sections suivantes, SQLAlchemy ORM est une excellente *solution de mappage de données* pour traduire les classes Python en tables et déplacer les données entre les instances de ces classes et les lignes de ces tables.

SQLAlchemy Types de données

En utilisant SQLAlchemy, nous pouvons être assurés que nous obtiendrons un support pour les types de données les plus courants trouvés dans les bases de données relationnelles. Par exemple, les booléens, les dates, les heures, les chaînes de caractères et les valeurs numériques ne sont qu'un sous-ensemble des types pour lesquels SQLAlchemy fournit des abstractions. En plus de ces types de base, SQLAlchemy inclut la prise en charge de quelques types spécifiques aux fournisseurs (comme JSON) et permet également aux développeurs de créer des types personnalisés et de redéfinir ceux existants.
Pour comprendre comment nous utilisons les types de données SQLAlchemy pour mapper les propriétés des classes Python en colonnes sur une table de base de données relationnelle, nous allons analyser l'exemple suivant :

```
class Product(Base):
    __tablename__ = 'products'
    id=Column(Integer, primary_key=True)
    title=Column('title', String(32))
    in_stock=Column('in_stock', Boolean)
    quantity=Column('quantity', Integer)
    price=Column('price', Numeric)
```

Dans l'extrait de code ci-dessus, nous définissons une classe appelée `Product` qui a six propriétés. Jetons un coup d'œil à ce que font ces propriétés :

- La propriété `__tablename__` indique à SQLAlchemy que les lignes de la table des `products` doivent être mappées à cette classe.
- La propriété `id` identifie qu'il s'agit de la `primary_key` de la table et que son type est `Integer`.
- La propriété `title` indique qu'une colonne de la table porte le même nom que la propriété et que son type est `String`.
- La propriété `in_stock` indique qu'une colonne de la table porte le même nom que la propriété et que son type est `Boolean`.
- La propriété `quantity` indique qu'une colonne de la table porte le même nom que la propriété et que son type est `Integer`.
- La propriété `price` indique qu'une colonne de la table porte le même nom que la propriété et que son type est `Numeric`.

Les développeurs expérimentés remarqueront que les bases de données relationnelles (habituellement) n'ont pas de types de données avec ces noms exacts. SQLAlchemy utilise ces types comme représentations génériques de ce que les bases de données supportent et utilise le dialecte configuré pour comprendre vers quels types ils se traduisent. Par exemple, sur une base de données PostgreSQL, le titre serait mappé à une colonne `varchar.`

SQLAlchemy Relationship Patterns

Maintenant que nous savons ce qu'est l'ORM et que nous avons étudié les types de données, apprenons à utiliser SQLAlchemy pour mapper les relations entre classes aux relations entre tables. SQLAlchemy supporte quatre types de relations : Un à plusieurs, plusieurs à un, un à un et plusieurs à plusieurs.

Notez que cette section sera un aperçu de tous ces types, mais dans la mise en œuvre de *SQLAlchemy ORM in Practice,* nous allons faire un exercice pratique concernant les classes de mappage dans les tables et pour apprendre comment insérer, extraire et supprimer des données de ces tables.

Le premier type, *One To Many*, est utilisé pour indiquer qu'une instance d'une classe peut être associée à plusieurs instances d'une autre classe. Par exemple, sur un moteur de blog, une instance de la classe `Article` peut être associée à plusieurs instances de la classe `Comment`. Dans ce cas, nous allons mapper les classes mentionnées et leur relation comme suit :

```
class Article(Base):
    __tablename__ = 'articles'
    id = Column(Integer, primary_key=True)
    comments = relationship("Comment")


class Comment(Base):
    __tablename__ = 'comments'
    id = Column(Integer, primary_key=True)
    article_id = Column(Integer, ForeignKey('articles.id'))
```

Le deuxième type, *Many To One*, fait référence à la même relation décrite ci-dessus, mais dans une autre perspective. Pour donner un autre exemple, disons que nous voulons mapper la relation entre les instances de `Tire` (Pneu) et une instance de `Car` (Voiture). Comme beaucoup de pneus appartiennent à une voiture et que cette voiture contient beaucoup de pneus, nous allons mapper cette relation comme suit :

```
class Tire(Base):
    __tablename__ = 'tires'
    id = Column(Integer, primary_key=True)
    car_id = Column(Integer, ForeignKey('cars.id'))
    car = relationship("Car")


class Car(Base):
    __tablename__ = 'cars'
    id = Column(Integer, primary_key=True)
```

Le troisième type, *One To One*, fait référence aux relations où une instance d'une classe particulière ne peut être associée qu'à une instance d'une autre classe, et vice versa. Prenons l'exemple de la relation entre une `Person` et un `MobilePhone`. Habituellement, une personne possède un seul téléphone mobile et ce téléphone n'appartient qu'à cette personne. Pour mapper cette relation sur SQLAlchemy, nous allons créer le code suivant :

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    mobile_phone = relationship("MobilePhone", uselist=False, back_populat
es="person")

class MobilePhone(Base):
    __tablename__ = 'mobile_phones'
    id = Column(Integer, primary_key=True)
    person_id = Column(Integer, ForeignKey('people.id'))
    person = relationship("Person", back_populates="mobile_phone")
```

Dans cet exemple, nous passons deux paramètres supplémentaires à la fonction `relationship`. La première, `uselist=False`, fait comprendre à SQLAlchemy que le `mobile_phone` ne contiendra qu'une seule instance et non un tableau (multiple) d'instances. Le second, `back_populates`, demande à SQLAlchemy de remplir l'autre côté de la carte. La documentation officielle de l'API Relationship fournit une explication complète de ces paramètres et couvre également d'autres paramètres non mentionnés ici.

Le dernier type supporté par SQLAlchemy, *Many To Many*, est utilisé lorsque les instances d'une classe particulière peuvent avoir zéro ou plus d'associations aux instances d'une autre classe. Par exemple, disons que nous sommes en train de mapper la relation entre les instances de `Student` (élève) et les instances de `Class` dans un système qui gère une école. Comme de nombreux élèves peuvent participer à de nombreuses classes, nous pourrions établir la relation comme suit :

```
students_classes_association = Table('students_classes', Base.metadata,
    Column('student_id', Integer, ForeignKey('students.id')),
    Column('class_id', Integer, ForeignKey('classes.id'))
)


class Student(Base):
    __tablename__ = 'students'
    id = Column(Integer, primary_key=True)
    classes = relationship("Class", secondary=students_classes_association
)

class Class(Base):
    __tablename__ = 'classes'
    id = Column(Integer, primary_key=True)
```

Dans ce cas, nous avons dû créer une table de jointures pour maintenir l'association entre les instances de `Student` et les instances de `Class`, car cela ne serait pas possible sans une table supplémentaire. Notez que, pour rendre SQLAlchemy conscient de la table de jointures, nous l'avons passée dans le paramètre `secondary` de la fonction `relationship.`
Les extraits de code ci-dessus ne montrent qu'un sous-ensemble des options de mappage prises en charge par SQLAlchemy. Dans les sections suivantes, nous allons examiner plus en détail chacun des modèles de relations disponibles. En outre, la documentation officielle est une excellente référence pour en savoir plus sur les schémas relationnels sur SQLAlchemy.

SQLAlchemy ORM Cascade

Chaque fois que des lignes d'une table particulière sont mises à jour ou supprimées, les lignes d'autres tables peuvent également subir des modifications. Ces modifications peuvent être de simples mises à jour, appelées mises à jour en cascade, ou des suppressions complètes, appelées suppressions en cascade. Par exemple, disons que nous avons une table appelée `shopping_carts` (caddie d'achat), une table appelée `products,` et une troisième appelée `shopping_carts_products` qui relie les deux premières tables. Si, pour une raison quelconque, nous avons besoin de supprimer des lignes de `shopping_carts`, nous devrons également supprimer les lignes correspondantes de `shopping_carts_products`. Sinon, nous finirons avec beaucoup d'inconsistances et de références insatisfaites dans notre base de données.
Pour rendre ce type d'opération facile à maintenir, SQLAlchemy ORM permet aux développeurs de mapper le comportement en cascade lorsqu'ils utilisent des constructions

`relationship()`. Ainsi, lorsque des opérations sont effectuées sur des objets *parents*, les objets *enfants* sont également mis à jour/supprimés. La liste suivante fournit une brève explication des stratégies en cascade les plus utilisées sur SQLAlchemy ORM :

- `save-update` : Indique que lorsqu'un objet parent est enregistré/mis à jour, les objets enfants sont également enregistrés/mis à jour.
- `delete` : Indique que lorsqu'un objet parent est supprimé, les enfants de cet objet seront également supprimés.
- `delete-orphan` : Indique que lorsqu'un objet enfant perd la référence à un parent, il sera supprimé.
- `merge` : Indique que les opérations `merge()`(fusion) se propagent de parent à enfant.

Si plus d'informations sur cette fonctionnalité sont nécessaires, la documentation de SQLAlchemy fournit un excellent chapitre sur Cascades.

Sessions SQLAlchemy

Les sessions, sur SQLAlchemy ORM, sont la mise en œuvre du modèle de conception de l'unité de travail. Comme l'explique Martin Fowler, une unité de travail est utilisée pour gérer une liste des objets affectés par une transaction métier et pour coordonner l'écriture de ces modifications. Cela signifie que toutes les modifications suivies par Sessions (Unités de Travaux) seront appliquées ensemble à la base de données sous-jacente, ou aucune d'entre elles ne le sera. En d'autres termes, les Sessions sont utilisées pour garantir la cohérence de la base de données.
La documentation officielle de SQLAlchemy ORM sur les Sessions explique comment les changements sont suivis, comment obtenir des sessions et comment créer des sessions ad-hoc. Cependant, dans cet article, nous utiliserons la forme la plus basique de création de session :

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

# create an engine
engine = create_engine('postgresql://usr:pass@localhost:5432/sqlalchemy')

# create a configured "Session" class
Session = sessionmaker(bind=engine)

# create a Session
session = Session()
```

Comme nous pouvons le voir à partir de l'extrait de code ci-dessus, nous n'avons besoin que d'une seule étape pour obtenir les sessions. Nous avons besoin de créer une usine de session qui est liée au moteur SQLAlchemy. Après cela, nous pouvons simplement émettre des appels à cette usine de session pour obtenir nos sessions.

SQLAlchemy dans la pratique

Maintenant que nous avons une meilleure compréhension des éléments les plus importants de SQLAlchemy, il est temps de commencer à le pratiquer. Dans les sections suivantes, nous allons créer un petit projet basé sur `pipenv` - un gestionnaire de dépendances Python - et y ajouter quelques classes. Ensuite, nous allons mapper ces classes aux tables persistantes dans une base de données PostgreSQL et apprendre à interroger les données.

Démarrer le projet du didacticiel

Pour créer notre projet de tutoriel, nous devons avoir Python installé sur notre machine et `pipenv` installé en tant que paquet Python global. Les commandes suivantes installeront `pipenv` et configureront le projet. Ces commandes dépendent de Python, donc assurez-vous de l'avoir installé avant de continuer :

```
# install pipenv globally
pip install pipenv

# create a new directory for our project
mkdir sqlalchemy-tutorial

# change working directory to it
cd sqlalchemy-tutorial

# create a Python 3 project
pipenv --three
```

Exécution de PostgreSQL

Pour pouvoir mettre en pratique nos nouvelles compétences et apprendre à interroger des données sur SQLAlchemy, nous aurons besoin d'une base de données pour soutenir nos exemples. Comme déjà mentionné, SQLAlchemy prend en charge de nombreux moteurs de bases de données différents, mais les instructions qui suivent se concentreront sur PostgreSQL. Il y a plusieurs façons d'obtenir une instance de PostgreSQL. L'un d'eux est d'utiliser un fournisseur de cloud comme Heroku ou ElephantSQL (les deux ont des versions libres). Une autre possibilité est d'installer PostgreSQL localement sur notre environnement actuel. Une troisième option est d'exécuter une instance PostgreSQL dans un conteneur Docker.
La troisième option est probablement le meilleur choix parce qu'elle a les performances d'une instance fonctionnant en local, qu'elle est gratuite pour toujours et qu'il est facile de créer et de détruire des instances Docker. Le seul (petit) inconvénient est que nous devons installer Docker localement.
Après avoir installé Docker, nous pouvons créer et détruire les instances PostgreSQL *dockerisées* avec les commandes suivantes :

```
# create a PostgreSQL instance
docker run --name sqlalchemy-orm-psql \
    -e POSTGRES_PASSWORD=pass \
    -e POSTGRES_USER=usr \
    -e POSTGRES_DB=sqlalchemy \
    -p 5432:5432 \
    -d postgres
# stop instance
docker stop sqlalchemy-orm-psql

# start instance

docker start sqlalchemy-orm-psql

# destroy instance
docker rm sqlalchemy-orm-psql
```

La première commande, celle qui crée l'instance PostgreSQL, contient quelques paramètres qui méritent d'être inspectés :

- `--name` : Définit le nom de l'instance du Docker.
- `-e POSTGRES_PASSWORD` : Définit le mot de passe pour se connecter à PostgreSQL.
- `-e POSTGRES_USER` : Définit l'utilisateur à connecter à PostgreSQL.

- `-e POSTGRES_DB` : Définit la base de données principale (et unique) disponible dans l'instance PostgreSQL.
- `-p 5432:5432` : Définit que le port local `5432` tunnelera les connexions vers le même port dans l'instance Docker.
- `-d postgres` : Définit que cette instance du Docker sera créée sur la base du référentiel officiel PostgreSQL.

Installation des dépendances de SQLAlchemy

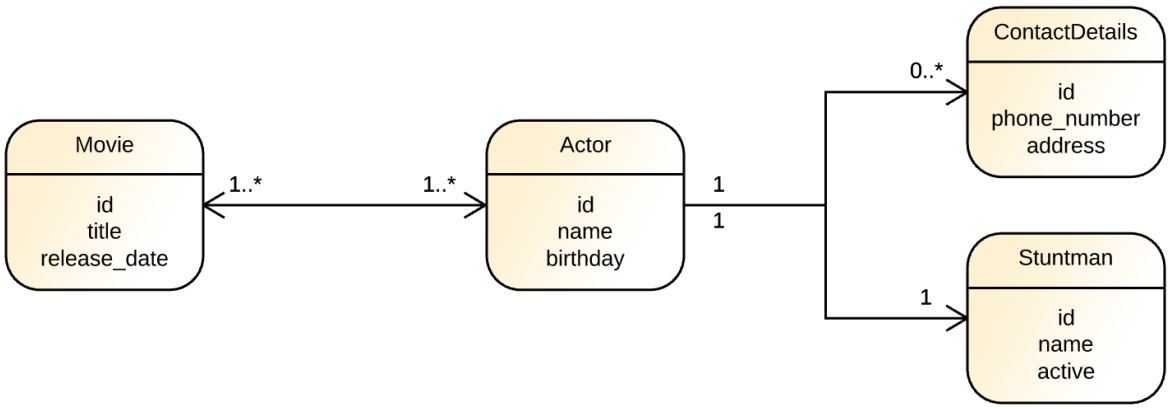
Dans ce tutoriel, nous n'aurons besoin d'installer que deux paquets : `sqlalchemy` et `psycopg2`. La première dépendance fait référence à SQLAlchemy lui-même et la seconde, `psycopg2`, est le pilote PostgreSQL que SQLAlchemy va utiliser pour communiquer avec la base de données. Pour installer ces dépendances, nous utiliserons `pipenv` comme indiqué :

```
# install sqlalchemy and psycopg2
pipenv install sqlalchemy psycopg2
```

Cette commande téléchargera les deux bibliothèques et les rendra disponibles dans notre environnement virtuel Python. Notez que pour exécuter les scripts que nous allons créer, nous devons d'abord générer le shell de l'environnement virtuel. C'est-à-dire, avant d'exécuter `python somescript.py`, nous devons exécuter `pipenv shell`. Sinon, Python ne pourra pas trouver les dépendances installées, car elles sont simplement disponibles dans notre nouvel environnement virtuel.

Classes de Mapping avec SQLAlchemy

Après avoir démarré l'instance PostgreSQL *dockerisée* et installé les dépendances Python, nous pouvons commencer à mapper les classes Python aux tables des bases de données. Dans ce tutoriel, nous allons mapper quatre classes simples qui représentent des films, des acteurs, des cascadeurs et des coordonnées de contact. Le diagramme suivant illustre les caractéristiques de ces entités et leurs relations.



Pour commencer, nous allons créer un fichier appelé `base.py` dans le répertoire principal de notre projet et y ajouter le code suivant :

```
# coding=utf-8

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('postgresql://usr:pass@localhost:5432/sqlalchemy')
Session = sessionmaker(bind=engine)

Base = declarative_base()
```

Ce code crée :

- un moteur SQLAlchemy qui interagira avec notre base de données PostgreSQL *dockerisée,*
- une fabrique de session SQLAlchemy ORM liée à ce moteur,
- et une classe de base pour nos définitions de classes.

Maintenant, créons et mappons la classe `Movie`. Pour ce faire, créons un nouveau fichier appelé `movie.py` et ajoutons-y le code suivant :

```
# coding=utf-8
from sqlalchemy import Column, String, Integer, Date

from base import Base


class Movie(Base):
    __tablename__ = 'movies'

    id = Column(Integer, primary_key=True)
    title = Column(String)
    release_date = Column(Date)

    def __init__(self, title, release_date):
        self.title = title
        self.release_date = release_date
```

La définition de cette classe et de ses caractéristiques de mapping est très simple. Nous commençons par faire en sorte que cette classe étende la classe `Base` définie dans le module `base.py`, puis nous y ajoutons quatre propriétés :

- Un `__tablename__` pour indiquer quel est le nom de la table qui supportera cette classe.
- Un `id` pour représenter la clé primaire dans la table.
- Un `title` de type `String`.
- Une `release_date` du type `Date`.

La classe suivante que nous allons créer et mapper est la classe `Actor.` Créez un fichier appelé `actor.py` et ajoutez-y le code suivant :

```
# coding=utf-8

from sqlalchemy import Column, String, Integer, Date

from base import Base


class Actor(Base):
    __tablename__ = 'actors'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    birthday = Column(Date)

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

La définition de cette classe est assez similaire à la précédente. Les différences sont que l'`Actor` a un `name` au lieu d'un `title`, un `birthday` au lieu d'une `release_date`, et qu'il pointe vers une table appelée `actors` au lieu de `movies`.

Comme beaucoup de films peuvent avoir beaucoup d'acteurs et vice-versa, nous aurons besoin de créer une relation de *Many To Many* entre ces deux classes. Créez cette relation en mettant à jour le fichier `movie.py` comme suit :

```
# coding=utf-8

from sqlalchemy import Column, String, Integer, Date, Table, ForeignKey
from sqlalchemy.orm import relationship

from base import Base

movies_actors_association = Table(
    'movies_actors', Base.metadata,
    Column('movie_id', Integer, ForeignKey('movies.id')),
    Column('actor_id', Integer, ForeignKey('actors.id'))
)


class Movie(Base):
    __tablename__ = 'movies'

    id = Column(Integer, primary_key=True)
    title = Column(String)
    release_date = Column(Date)
    actors = relationship("Actor", secondary=movies_actors_association)

    def __init__(self, title, release_date):
        self.title = title
        self.release_date = release_date
```

La différence entre cette version et la précédente est que :

- nous avons importé trois nouvelles entités : `Table`, `ForeignKey`, et `relationship`;
- nous avons créé une table `movies_actors_association` qui relie des rangées `actors` et des rangées de `films`;
- et nous avons ajouté la propriété `actors` à `Movie` et configuré la `movies_actors_association` comme table intermédiaire.

La prochaine classe que nous allons créer est `Stuntman`. Dans notre tutoriel, un `Actor` particulier n'aura qu'un `Stuntman` et ce `Stuntman` ne travaillera qu'avec cet `Actor`. Cela signifie que nous devons créer la classe `Stuntman` et une relation *One to One* entre ces classes. Pour ce faire, créons un fichier appelé `stuntman.py` et ajoutons-y le code suivant :

```
# coding=utf-8

from sqlalchemy import Column, String, Integer, Boolean, ForeignKey
from sqlalchemy.orm import relationship, backref

from base import Base


class Stuntman(Base):
    __tablename__ = 'stuntmen'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    active = Column(Boolean)
    actor_id = Column(Integer, ForeignKey('actors.id'))
    actor = relationship("Actor", backref=backref("stuntman", uselist=Fals
e))

    def __init__(self, name, active, actor):
        self.name = name
        self.active = active
        self.actor = actor
```

Dans cette classe, nous avons défini que la propriété `actor` fait référence à une instance d'`Actor` et que cet acteur obtiendra une propriété appelée `stuntman` qui n'est pas une liste (`uselist=False`). C'est-à-dire, chaque fois que nous chargeons une instance de `Stuntman`, SQLAlchemy chargera et remplira également l'`Actor` associé à ce cascadeur.

La quatrième et dernière classe que nous allons mapper dans notre tutoriel est `ContactDetails`. Les exemples de cette classe tiendront un `phone_number` et une `addresse` d'un `Actor` particulier, et un `Actor` pourra avoir de nombreux `ContactDetails` associés. Par conséquent, nous devrons utiliser le modèle de relation *Many To One* pout établir cette association.

Pour créer cette classe et cette association, créons un fichier appelé `contact_details.py` et ajoutons le code source suivant :

```
# coding=utf-8

from sqlalchemy import Column, String, Integer, ForeignKey
from sqlalchemy.orm import relationship

from base import Base


class ContactDetails(Base):
    __tablename__ = 'contact_details'

    id = Column(Integer, primary_key=True)
    phone_number = Column(String)
    address = Column(String)
    actor_id = Column(Integer, ForeignKey('actors.id'))
    actor = relationship("Actor", backref="contact_details")

    def __init__(self, phone_number, address, actor):
        self.phone_number = phone_number
        self.address = address
        self.actor = actor
```

Comme nous pouvons le voir, la création d'une association *Many To One* est un peu similaire à la création d'une association *One to One.* La différence est que dans ce dernier cas, nous avons demandé à SQLAlchemy de ne pas utiliser de listes. Cette instruction finit par limiter l'association à une seule instance au lieu d'une liste d'instances.

Données persistantes avec SQLAlchemy ORM

Maintenant que nous avons créé nos classes, créons un fichier appelé `inserts.py` et générons quelques instances de ces classes pour qu'elles persistent dans la base de données. Dans ce fichier, ajoutons le code suivant :

```
# coding=utf-8

# 1 - imports
from datetime import date

from actor import Actor
from base import Session, engine, Base
from contact_details import ContactDetails
from movie import Movie
from stuntman import Stuntman

# 2 - generate database schema
Base.metadata.create_all(engine)

# 3 - create a new session
session = Session()

# 4 - create movies
bourne_identity = Movie("The Bourne Identity", date(2002, 10, 11))
furious_7 = Movie("Furious 7", date(2015, 4, 2))
pain_and_gain = Movie("Pain & Gain", date(2013, 8, 23))

# 5 - creates actors
matt_damon = Actor("Matt Damon", date(1970, 10, 8))
dwayne_johnson = Actor("Dwayne Johnson", date(1972, 5, 2))
mark_wahlberg = Actor("Mark Wahlberg", date(1971, 6, 5))

# 6 - add actors to movies
bourne_identity.actors = [matt_damon]
furious_7.actors = [dwayne_johnson]
pain_and_gain.actors = [dwayne_johnson, mark_wahlberg]

# 7 - add contact details to actors
matt_contact = ContactDetails("415 555 2671", "Burbank, CA", matt_damon)
dwayne_contact = ContactDetails("423 555 5623", "Glendale, CA", dwayne_joh
nson)
dwayne_contact_2 = ContactDetails("421 444 2323", "West Hollywood, CA", dw
ayne_johnson)
mark_contact = ContactDetails("421 333 9428", "Glendale, CA", mark_wahlber
g)

# 8 - create stuntmen
matt_stuntman = Stuntman("John Doe", True, matt_damon)
dwayne_stuntman = Stuntman("John Roe", True, dwayne_johnson)
```

```
mark_stuntman = Stuntman("Richard Roe", True, mark_wahlberg)

# 9 - persists data
session.add(bourne_identity)
session.add(furious_7)
session.add(pain_and_gain)

session.add(matt_contact)
session.add(dwayne_contact)
session.add(dwayne_contact_2)
session.add(mark_contact)

session.add(matt_stuntman)
session.add(dwayne_stuntman)
session.add(mark_stuntman)

# 10 - commit and close session
session.commit()
session.close()
```

Ce code est divisé en 10 sections. Allons les inspecter :

1. La première section importe les classes que nous avons créées, le moteur SQLAlchemy, la classe Base, la fabrique de session et la `date` du module `datetime.`
2. La deuxième section demande à SQLAlchemy de générer le schéma de base de données. Cette génération se produit à partir des déclarations que nous avons faites en créant les quatre classes principales qui composent notre tutoriel.
3. La troisième section extrait une nouvelle session de la fabrique de session.
4. La quatrième section crée trois instances de la classe `Movie.`
5. La cinquième section crée trois instances de la classe `Actor.`
6. La sixième section ajoute des acteurs aux films. A noter que le film *Pain & Gain* fait référence à deux acteurs : *Dwayne Johnson* et *Mark Wahlberg*.
7. La septième section crée des instances de la classe `ContactDetails` et définit à quels acteurs ces instances sont associées.
8. La huitième section définit trois cascadeurs et définit également à quels acteurs ces cascadeurs sont associés.
9. La neuvième section utilise la session en cours pour enregistrer les films, les acteurs, les coordonnées et les cascadeurs créés. Notez que nous n'avons pas explicitement sauvé les acteurs. Ceci n'est pas nécessaire car SQLAlchemy, par défaut, utilise la stratégie de `save-update` en cascade.
10. La dixième section commit la session en cours dans la base de données et la ferme.

Pour exécuter ce script Python, nous pouvons simplement lancer la commande `python inserts.py` (n'oublions pas de lancer `pipenv shell` d'abord) dans le répertoire principal de notre base. L'exécuter créera cinq tables dans la base de données PostgreSQL et remplira ces tables avec les données que nous avons créées. Dans la section suivante, nous apprendrons comment interroger ces tables.

Interrogation des données avec SQLAlchemy ORM

Comme nous le verrons, l'interrogation des données avec SQLAlchemy ORM est très simple. Cette bibliothèque fournit une API intuitive et fluide qui permet aux développeurs d'écrire des requêtes faciles à lire et à maintenir. Sur SQLAlchemy ORM, toutes les requêtes commencent par un objet de requête qui est extrait de la session en cours et qui est associé à une classe mappée particulière. Pour voir cette API en action, créons un fichier appelé `queries.py` et ajoutons-y le code source suivant :

```
# coding=utf-8

# 1 - imports
from actor import Actor
from base import Session
from contact_details import ContactDetails
from movie import Movie

# 2 - extract a session
session = Session()

# 3 - extract all movies
movies = session.query(Movie).all()

# 4 - print movies' details
print('\n### All movies:')
for movie in movies:
    print(f'{movie.title} was released on {movie.release_date}')
print('')
```

L'extrait de code ci-dessus - qui peut être exécuté avec `python queries.py`,- montre combien il est facile d'utiliser SQLAlchemy ORM pour interroger des données. Pour récupérer tous les films de la base de données, il nous suffisait de récupérer une session depuis la fabrique de session, de l'utiliser pour obtenir une requête associée à `Movie`, puis d'appeler la fonction `all()` sur cet objet de requête. L'API de requête fournit des douzaines de fonctions utiles comme `all()`. Dans la liste suivante, nous pouvons voir une brève explication sur les plus importantes :

- `count()` : Retourne le nombre total de lignes d'une requête.
- `filter()` : Filtre la requête en appliquant un critère.
- `delete()` : Supprime de la base de données les lignes correspondant à une requête.
- `distinct()` : Applique une instruction distincte à une requête.
- `exists()` : Ajoute un opérateur existant à une sous-requête.
- `first()` : Renvoie la première ligne d'une requête.
- `get()` : Renvoie la ligne référencée par le paramètre clé primaire passé en argument.
- `join()` : Crée une jointure SQL dans une requête.
- `limit()` : Limite le nombre de lignes retournées par une requête.

- `order_by()` : Définit un ordre dans les lignes retournées par une requête.

Pour explorer l'utilisation de certaines de ces fonctions, ajoutons le code suivant au script `queries.py` :

```
# 1 - imports
from datetime import date

# other imports and sections...

# 5 - get movies after 15-01-01
movies = session.query(Movie) \
    .filter(Movie.release_date > date(2015, 1, 1)) \
    .all()

print('### Recent movies:')
for movie in movies:
    print(f'{movie.title} was released after 2015')
print('')

# 6 - movies that Dwayne Johnson participated
the_rock_movies = session.query(Movie) \
    .join(Actor, Movie.actors) \
    .filter(Actor.name == 'Dwayne Johnson') \
    .all()

print('### Dwayne Johnson movies:')
for movie in the_rock_movies:
    print(f'The Rock starred in {movie.title}')
print('')

# 7 - get actors that have house in Glendale
glendale_stars = session.query(Actor) \
    .join(ContactDetails) \
    .filter(ContactDetails.address.ilike('%glendale%')) \
    .all()

print('### Actors that live in Glendale:')
for actor in glendale_stars:
    print(f'{actor.name} has a house in Glendale')
print('')
```

La cinquième section du script mis à jour utilise la fonction `filter()` pour récupérer uniquement les films qui ont été publiés après le 1er janvier 2015. La sixième section montre comment utiliser `join()` pour récupérer les instances de `Movie` auxquelles l'`Actor` Dwayne Johnson a participé. La septième et dernière section, montre l'utilisation des fonctions `join()` et `ilike()` pour récupérer les acteurs qui ont des maisons à Glendale.

Exécuter la nouvelle version du script (`python queries.py`) maintenant aura pour résultat la sortie suivante :

```
### All movies:
The Bourne Identity was released on 2002-10-11
Furious 7 was released on 2015-04-02
No Pain No Gain was released on 2013-08-23

### Recent movies:
Furious 7 was released after 2015

### Dwayne Johnson movies:
The Rock starred in No Pain No Gain
The Rock starred in Furious 7

### Actors that live in Glendale:
Dwayne Johnson has a house in Glendale
Mark Wahlberg has a house in Glendale
```

Comme on peut le voir, l'utilisation de l'API est simple et génère un code lisible. Pour voir les autres fonctions supportées par l'API de requête et leur description, consultez la documentation officielle.

"L'interrogation des données avec SQLAlchemy ORM est simple et intuitive."

Sécurisation des API Python avec Auth0

Sécuriser les API Python avec Auth0 est très facile et apporte de nombreuses fonctionnalités intéressantes. Avec Auth0, il suffit d'écrire quelques lignes de code pour obtenir :

- Une solution solide de gestion des identités, incluant l'authentification unique
- Gestion des utilisateurs
- Support pour les fournisseurs d'identité sociale (comme Facebook, GitHub, Twitter, etc.)
- Fournisseurs d'identité d'entreprise (Active Directory, LDAP, SAML, etc.)
- Notre propre base de données d'utilisateurs

Par exemple, pour sécuriser les API Python écrites avec Flask, nous pouvons simplement créer un décorateur `requires_auth` :

```
# Format error response and append status code

def get_token_auth_header():
    """Obtains the access token from the Authorization Header
    """
    auth = request.headers.get("Authorization", None)
    if not auth:
        raise AuthError({"code": "authorization_header_missing",
                         "description":
                             "Authorization header is expected"}, 401)

    parts = auth.split()

    if parts[0].lower() != "bearer":
        raise AuthError({"code": "invalid_header",
                         "description":
                             "Authorization header must start with"
                             " Bearer"}, 401)
    elif len(parts) == 1:
        raise AuthError({"code": "invalid_header",
                         "description": "Token not found"}, 401)
    elif len(parts) > 2:
        raise AuthError({"code": "invalid_header",
                         "description":
                             "Authorization header must be"
                             " Bearer token"}, 401)

    token = parts[1]
    return token

def requires_auth(f):
    """Determines if the access token is valid
    """
    @wraps(f)
    def decorated(*args, **kwargs):
```

```
        token = get_token_auth_header()
        jsonurl = urlopen("https://"+AUTH0_DOMAIN+"/.well-known/jwks.json"
)
        jwks = json.loads(jsonurl.read())
        unverified_header = jwt.get_unverified_header(token)
        rsa_key = {}
        for key in jwks["keys"]:
            if key["kid"] == unverified_header["kid"]:
                rsa_key = {
                    "kty": key["kty"],
                    "kid": key["kid"],
                    "use": key["use"],
                    "n": key["n"],
                    "e": key["e"]
                }
        if rsa_key:
            try:
                payload = jwt.decode(
                    token,
                    rsa_key,
                    algorithms=ALGORITHMS,
                    audience=API_AUDIENCE,
                    issuer="https://"+AUTH0_DOMAIN+"/"
                )
            except jwt.ExpiredSignatureError:
                raise AuthError({"code": "token_expired",
                                 "description": "token is expired"}, 401)
            except jwt.JWTClaimsError:
                raise AuthError({"code": "invalid_claims",
                                 "description":
                                     "incorrect claims,"
                                     "please check the audience and issuer"
}, 401)
            except Exception:
                raise AuthError({"code": "invalid_header",
                                 "description":
                                     "Unable to parse authentication"
                                     " token."}, 400)

            _app_ctx_stack.top.current_user = payload
            return f(*args, **kwargs)
        raise AuthError({"code": "invalid_header",
                         "description": "Unable to find appropriate key"},
400)
    return decorated
```

Ensuite, utilisez-le dans nos points finaux :

```
# Controllers API

# This doesn't need authentication
@app.route("/ping")
@cross_origin(headers=['Content-Type', 'Authorization'])
def ping():
    return "All good. You don't need to be authenticated to call this"

# This does need authentication
@app.route("/secured/ping")
@cross_origin(headers=['Content-Type', 'Authorization'])
@requires_auth
def secured_ping():
    return "All good. You only get this message if you're authenticated"
```

Pour en savoir plus sur la sécurisation des *API Python* avec Auth0, consultez ce tutoriel. En plus des tutoriels pour les technologies dorsales (comme Python, Java et PHP), la page Web *Auth0 Docs* propose également des tutoriels pour les *applications mobiles/natives* et les *applications mono-page*.

Prochaines étapes

Nous avons parcouru beaucoup de chemin dans cet article. Nous avons appris les concepts de base de SQLAlchemy comme les moteurs, les pools de connexion et les dialectes. Ensuite, nous avons appris comment SQLAlchemy traite des sujets ORM tels que les modèles de relations, les stratégies en cascade et l'API de requête. En fin de compte, nous avons appliqué ces connaissances dans un petit exercice. En résumé, nous avons eu la chance d'apprendre et de pratiquer les pièces les plus importantes de SQLAlchemy et SQLAlchemy ORM. Dans le prochain article, nous allons utiliser ces nouvelles compétences pour implémenter des API RESTful avec Flask, le microframework Python pour le web. Restez à l'écoute !