models.py

Un Modèle dans SQLAlchemy est une classe Python qui représente une table de votre base de données. Chaque attribut de la classe correspond à une colonne de la table.

Le fichier models.py contiendra la représentation pythonique des tables de notre base de données. Les classes de ce fichier seront utilisées lorsque les utilisateurs de l'API interrogereront la base de données.

```
"""SQLAlchemy models"""
from sqlalchemy import Column, Integer, String, Float, ForeignKey
from sqlalchemy.orm import relationship # permet des relations de clé
étrangère entre les tables.
from database import Base

class Movie(Base):
    __tablename__ = "movies"

    movieId = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
    genres = Column(String)

    ratings = relationship("Rating", back_populates="movie")
    tags = relationship("Tag", back_populates="movie")
    link = relationship("Link", uselist=False, back_populates="movie")


class Rating(Base):
    __tablename__ = "ratings"

    userId = Column(Integer, primary_key=True)
    movieId = Column(Integer, ForeignKey("movies.movieId"),
primary_key=True)
    rating = Column(Float)
    timestamp = Column(Integer)

    movie = relationship("Movie", back_populates="ratings")


class Tag(Base):
    __tablename__ = "tags"

    userId = Column(Integer, primary_key=True)
    movieId = Column(Integer, ForeignKey("movies.movieId"),
primary_key=True)
    tag = Column(String, primary_key=True)
    timestamp = Column(Integer)

    movie = relationship("Movie", back_populates="tags")


class Link(Base):
```

```
    __tablename__ = "links"

    movieId = Column(Integer, ForeignKey("movies.movieId"),
primary_key=True)
    imdbId = Column(String)
    tmdbId = Column(Integer)

    movie = relationship("Movie", back_populates="link")
```

Chaque table est bien représentée avec ses clés primaires, clés étrangères et types.

Explication de la classe Movie

Nous allons maintenant définir la classe `Movie`, qui est la classe Python utilisée pour stocker les données de la table SQLite `movies`. Cette classe est une sous-classe de `Base`, un modèle de base importé depuis le fichier `database.py`.

Nous utilisons l'attribut spécial `__tablename__` pour indiquer à SQLAlchemy que cette classe est associée à la table `movies`. Ainsi, lorsque nous interrogerons SQLAlchemy avec la classe `Movie`, il saura automatiquement qu'il doit récupérer les données de la table `movies`. C'est l'un des principaux avantages d'un ORM : il permet de mapper le code Python à la base de données sous-jacente de manière transparente.

```
class Movie(Base):
    __tablename__ = "movies"
```

Le reste de la définition de la classe `Movie` permet de mapper les colonnes de la base de données aux attributs Python correspondants. Chaque attribut est défini à l'aide de la fonction `Column` fournie par SQLAlchemy :

```
    movieId = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
    genres = Column(String)
```

Voici quelques points à noter sur ces définitions :

- **Les noms des attributs** (`movieId`, `title`, `genres`) correspondent directement aux noms des colonnes dans la base de données.
- **Les types de données** utilisés (`Integer`, `String`) sont fournis par SQLAlchemy et doivent être importés avant d'être utilisés. Ils correspondent aux types SQL sous-jacents dans SQLite.
- **La clé primaire** (`movieId`) est définie avec `primary_key=True`, ce qui permet d'assurer l'unicité des enregistrements et d'optimiser les requêtes.

En plus des colonnes, nous définissons des **relations** entre les tables en utilisant la fonction `relationship()`. Cela permet d'accéder facilement aux données associées sans avoir à écrire des jointures SQL complexes :

```
    ratings = relationship("Rating", back_populates="movie", cascade="all,
delete")
    tags = relationship("Tag", back_populates="movie", cascade="all,
delete")
    link = relationship("Link", back_populates="movie", uselist=False,
cascade="all, delete")
```

Explication des relations :

- **`ratings = relationship("Rating", back_populates="movie")`**
  - Cela établit une relation entre la classe `Movie` et `Rating`.
  - `back_populates="movie"` signifie que chaque objet `Rating` aura aussi un attribut `movie` pointant vers le film correspondant.
- **`tags = relationship("Tag", back_populates="movie")`**
  - De la même manière, cette relation permet de récupérer tous les tags associés à un film.
- **`link = relationship("Link", back_populates="movie", uselist=False)`**
  - Cette relation est un peu différente : `uselist=False` signifie qu'il ne peut y avoir qu'un seul lien (`Link`) pour chaque film (`Movie`).

Grâce à ces relations, nous pourrons écrire du code comme ceci pour récupérer les évaluations d'un film :

```
movie = session.query(Movie).filter_by(movieId=1).first()
print(movie.ratings)  # Affichera toutes les évaluations associées au film
avec ID 1
```

Cela nous permet d'exploiter la puissance de SQLAlchemy pour manipuler les données de manière simple et intuitive.