diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Requêtage One-to-Many avec SQLModel

Sommaire

Différence entre une relation simple et une relation avec `Relationship`

En SQLModel, on peut relier deux tables en utilisant :

1. **Uniquement une clé étrangère** (`foreign_key="..."`)
2. **Une clé étrangère + des objets liés** via `Relationship(...)`

Exemple :

```
team_id: Optional[int] = Field(default=None, foreign_key="team.id")
team: Optional["Team"] = Relationship(back_populates="heroes")
```

> ⚠ Le typage de la classe 'Team' est entre guillemets pour éviter les problèmes de référence circulaire.
> Le typage `Team` est bien effectif avec sa classe.

Sans `Relationship`

- On stocke juste un `team_id` dans `Hero`
- Pour accéder à l'équipe d'un héros, il faut faire une requête manuelle (ex : `join` entre `Hero` et `Team`)
- Pas de navigation automatique (`hero.team` ou `team.heroes` impossible)

Avec `Relationship`

- On relie les objets Python entre eux : `hero.team` devient accessible
- La relation est **navigable dans les deux sens** grâce à `back_populates`
- Cela permet de manipuler **plus facilement** les liens entre entités, sans requêtes complexes

Modèle avec `Relationship` et `back_populates`

```
from sqlmodel import SQLModel, Field, Relationship
from typing import Optional, List

class Team(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    headquarters: str

    heroes: List["Hero"] = Relationship(back_populates="team",
passive_deletes="all")

class Hero(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    secret_name: str
    age: Optional[int] = None

    team_id: Optional[int] = Field(default=None, foreign_key="team.id",
ondelete="RESTRICT")
    team: Optional[Team] = Relationship(back_populates="heroes")
```

Explication des champs :

- `team_id` : champ SQL pour la relation, obligatoire pour la base
- `team` : relationship Many-to-One vers `Hero`
- `heroes` : relationship One-to-Many vers `Team`
- `back_populates` : permet à SQLModel de **lier automatiquement les deux champs 'Relationship / back_populates'**

Pourquoi utiliser `passive_deletes` et `ondelete` ?

Dans une relation entre deux tables SQL, la **suppression** d'un parent (ex : une équipe) peut entraîner différents comportements vis-à-vis des enfants (ex : ses héros).

C'est ce que contrôlent les paramètres :

- `ondelete` : côté **SQL / base de données**
- `passive_deletes` : côté **SQLModel / Python**

◇ `ondelete="RESTRICT"`

Ce paramètre est utilisé dans :

```
team_id: Optional[int] = Field(default=None, foreign_key="team.id",
ondelete="RESTRICT")
```

Il définit le comportement de la base de données **si on tente de supprimer une équipe qui est encore liée à un ou plusieurs héros**.

☑ Ce que fait `"RESTRICT"` :

- Empêche la suppression si des héros existent encore
- Assure une **intégrité stricte** (pas de héros orphelins)
- Nécessite que l'on **détache manuellement** tous les héros avant de supprimer l'équipe

❌ Ce que `"RESTRICT"` évite :

- Suppressions accidentelles en cascade
- Incohérences silencieuses

◇ `passive_deletes="all"`

Ce paramètre est utilisé dans :

```
heroes: List["Hero"] = Relationship(back_populates="team", passive_deletes="all")
```

Il dit à SQLAlchemy / SQLModel de **ne pas aller chercher tous les objets liés** pour mettre à jour les relations manuellement lors d'une suppression. Cela suppose que **la base est déjà configurée** pour s'en occuper, via `ondelete`.

☑ Ce que ça permet :

- Meilleures performances : pas de SELECT pour charger tous les enfants liés
- Laisse la base de données gérer la cohérence
- Code plus simple et rapide

> ⚠ Ce paramètre **n'est utile que si la base fait respecter** `ondelete`. Si vous n'avez pas bien défini l'action dans la base, cela peut provoquer des erreurs ou des incohérences.

Alternatives possibles

Voici un résumé des options que l'on peut utiliser dans `ondelete` :

Valeur	Effet
`RESTRICT`	🔒 Empêche la suppression si des enfants existent
`SET NULL`	Remplace la clé étrangère par `NULL` chez les enfants
`CASCADE`	Supprime aussi tous les enfants automatiquement
`NO ACTION`	Laisse la base décider (souvent équivalent à `RESTRICT`)
`SET DEFAULT`	Remplace par une valeur par défaut (rarement utilisé)
...	Autres options spécifiques à la base de données

> 🎯 Dans un projet pédagogique ou en API publique, **RESTRICT** est souvent préférable pour éviter des suppressions involontaires.

Exemples de manipulation CRUD

Créer une équipe avec des héros

```
hero_1 = Hero(name="Black Lion", secret_name="Trevor Challa")
hero_2 = Hero(name="Princess Sure-E", secret_name="Sure-E")

wakaland = Team(name="Wakaland", headquarters="Capital City", heroes=[hero_1,
hero_2])

with Session(engine) as session:
    session.add(wakaland)
    session.commit()
```

Ajouter un héros à une équipe existante

```
with Session(engine) as session:
    team = session.exec(select(Team).where(Team.name == "Wakaland")).one()
    new_hero = Hero(name="Dr. Weird", secret_name="Steve Weird", age=36)
    team.heroes.append(new_hero)
    session.add(team)
    session.commit()
```

Lire tous les héros d'une équipe

```
with Session(engine) as session:
    team = session.exec(select(Team).where(Team.name == "Wakaland")).one()
    for hero in team.heroes:
        print(hero.name)
```

Retirer un héros de son équipe

```
with Session(engine) as session:
    hero = session.exec(select(Hero).where(Hero.name == "Black Lion")).one()
    hero.team = None
    session.add(hero)
    session.commit()
```

Supprimer une équipe

```
with Session(engine) as session:
    team = session.exec(select(Team).where(Team.name == "Wakaland")).one()
    session.delete(team)
    session.commit()
```

> Grâce à `passive_deletes="all"`, les héros de l'équipe ne sont pas supprimés. Leur champ `team_id` devient simplement `NULL`.