Architecture Logicielle

heritage SQLAlchemy

SQLAlchemy propose trois formes d'héritage

- **joined table** : la hiérarchie de classes est répartie dans plusieurs tables dépendantes. Chaque table contient les attributs locaux à la classe
- **une seule table** : plusieurs classes sont représentées dans une seule table
- **plusieurs tables indépendantes** : chaque classe est représentée dans des tables indépendantes

Joined table

- Chaque classe a sa propre table. Les tables des classes enfants sont liées à la classe parente
- Lorsque l'on a besoin d'une sous-classe, il est nécessaire de faire une requête sur plusieurs tables, représentant la hiérarchie, et de les assembler avec un join
- Si la classe de base est nécessaire, il n'y aura qu'une seule requête, sur la table de base

- Un attribut particulier dans la classe parente est choisi pour être le **champ discriminateur**, distinguant les différentes sous-classes

```
class Employee(db.Model):

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))
    type = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "employee",
        "polymorphic_on": type,
    }
```

```
__mapper_args__ = {
    "polymorphic_identity": "employee",
    "polymorphic_on": type,
}
```

- L'attribut `type` est le champ discriminateur dans cet exemple. Il est défini par le paramètre `polymorphic_on`. Ce paramètre accepte un nom de colonne
- La valeur à donner à ce paramètre pour cette table est spécifié par la paramètre `polymorphic_identity`. Dans notre exemple, c'est `employee`

Classes enfants :

```
class Engineer(Employee):
    id = db.Column(db.Integer, db.ForeignKey('employee.id'), primary_key=True)
    engineer_name = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "engineer",
    }


class Manager(Employee):
    id = db.Column(db.Integer, db.ForeignKey('employee.id'), primary_key=True)

    manager_name = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "manager",
    }
```

- Chaque classe enfant spécifie la valeur du paramètre de mapping `polymorphic_identity`
- Cette valeur doit être unique dans toutes la hiérarchie des classes
- Dans la table `Employee`, lorsqu'une ligne correspond à un `Engineer`, le champ type est égale à `engineer`, pour une `Manager` ; il est égale à `manager`
- Ce champ permet à l'ORM de savoir à quelle classe la ligne de la table correspond

- Chaque classe enfant doit avoir un attribut qui est une clé étrangère vers la classe parente
- Généralement, c'est la clé primaire
- Cet attribut servira à faire la jointure enter la classe parent et enfant

Il est possible de définir des relations

```
class Company(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))


class Employee(db.Model):

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))
    type = db.Column(db.String(120))
    company_id = db.Column(db.Integer, db.ForeignKey('company.id'))
    company = db.relationship("Company",
        backref=db.backref("employees"))

    __mapper_args__ = {
        "polymorphic_identity": "employee",
        "polymorphic_on": type,
    }
```

Ici entre tous les employés et l'entreprise

Il est possible de définir des relations avec uniquement une classe enfant

```
class Company(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))

class Employee():
...
class Manager(Employee):
    id = db.Column(db.Integer, db.ForeignKey('employee.id'), primary_key=True)

    manager_name = db.Column(db.String(120))
 company_id = db.Column(db.Integer, db.ForeignKey('company.id'))
    company = db.relationship("Company",
        backref=db.backref("managers"))

    __mapper_args__ = {
        "polymorphic_identity": "manager",
    }
```

Ici entre uniquement les managers et l'entreprise

Compléments pour la jointure

- `with_polymorphic` : Il est possible de spécifier comment les tables seront chargées avec la paramètre `with_polymorphic`. En lui donnant la valeur * toutes les classes descendantes seront chargées immédiatement
- `polymorphic_load` : deux valeurs sont possibles
  - `inline` : lors d'une requete sur la classe parente, les champs de la table enfant seront inclus dans la requete SELECT
  - `selectin` :quand les instances de cette chargées, une requête select supplémentaire est nécessaire pour retrouver les champs de cette sous-classe

```
class Employee(db.Model):

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))
    type = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "employee",
        "with_polymorphic" : "*",
        "polymorphic_on": type,
    }
```

```
class Engineer(Employee):
    id = db.Column(db.Integer, db.ForeignKey('employee.id'), primary_key=True)
    engineer_name = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "engineer",
        "with_polymorphic" : "*",
        "polymorphic_load": "inline"
    }


class Manager(Employee):
    id = db.Column(db.Integer, db.ForeignKey('employee.id'), primary_key=True)
    manager_name = db.Column(db.String(120))

    __mapper_args__ = {
        "polymorphic_identity": "manager",
        "with_polymorphic" : "*",
        "polymorphic_load": "inline"
    }
```

https://docs.sqlalchemy.org/en/20/orm/inheritance.html