Utiliser des schémas de données

Dans FastAPI, la gestion des données reçues et renvoyées est facilitée par **Pydantic**, une bibliothèque qui utilise les annotations de types Python pour **valider** et **sérialiser** les données automatiquement.
Pydantic permet de définir des modèles de données (appelés `BaseModel`) qui servent à valider les entrées et sorties de l'API, tout en générant une documentation interactive.

Sommaire

Qu'est-ce qu'un modèle Pydantic (`BaseModel`) ?

Un modèle Pydantic est une classe Python qui hérite de `BaseModel`. Il définit la structure des données attendues, avec :

- Des attributs typés (ex : `name: str`, `age: int`)
- Une validation automatique à la création d'une instance
- Une conversion facile en JSON

Exemple simple

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int
```

- Ici, `User` attend un objet avec un nom (`str`) et un âge (`int`).
- Si on crée `User(name="Alice", age="30")`, Pydantic convertira `"30"` en entier automatiquement.
- Si on oublie un champ ou si le type est incompatible, une erreur sera levée.

À quoi ça sert dans FastAPI ?

- **Validation des données d'entrée** (ex : corps JSON d'une requête)
- **Documentation automatique** (les schémas apparaissent dans Swagger UI)
- **Sérialisation des réponses** (FastAPI convertit automatiquement en JSON)

Pourquoi passer ensuite à SQLModel ?

Pydantic gère la validation et la sérialisation, mais il ne s'occupe pas de la base de données.

SQLModel étend Pydantic en ajoutant la gestion des tables et des interactions avec une base SQL, tout en conservant la simplicité et la validation des données.

On va donc maintenant voir comment créer des modèles SQLModel qui seront à la fois nos schémas de validation et nos modèles de base de données.

Définir une table avec SQLModel et créer ses schémas associés

1. Exemple simple de table `User`

```
from sqlmodel import SQLModel, Field

class User(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str
    email: str
```

- `id` est la clé primaire, optionnelle car auto-générée par la base.
- `name` et `email` sont des champs obligatoires.

2. Création des schémas Pydantic dédiés

Pour manipuler les données dans différentes situations (création, mise à jour, lecture), on crée plusieurs schémas basés sur la table `User`.

Cette organisation claire permet :

- de contrôler les champs obligatoires et optionnels selon les cas d'usage,
- d'éviter de manipuler directement la table SQLModel dans toutes les fonctions,
- d'avoir une validation adaptée et une documentation claire.

Schéma de base (abstrait) — commun aux autres

```
class UserBase(SQLModel):
    name: str
    email: str
```

Schéma pour la création (`POST`) — sans `id`

```
class UserCreate(UserBase):
    pass  # héritage direct, tous champs requis
```

Schéma pour la mise à jour (`PUT`/`PATCH`) — tous optionnels

```python
class UserUpdate(SQLModel):
    name: str | None = None
    email: str | None = None
```

Schéma pour la lecture/réponse (`GET`) — avec `id`

Comme le schéma de lecture possède un `id` et qu'il est identique à celui de la table `User`, on peut utiliser directement le modèle SQLModel pour la réponse ou créer un schéma spécifique :

```python
class UserRead(UserBase):
    id: int
```