Organisation d'un projet FastAPI

Sommaire

Pourquoi structurer un projet ?

Un projet API simple peut commencer dans un seul fichier. Mais dès que l'API grossit, il devient vite **difficile de s'y retrouver** :

- trop de routes entassées
- logique métier mélangée à la configuration
- duplication du code

Avantages d'un découpage modulaire :

- Meilleure **lisibilité**
- **Réutilisation** de code
- Tests plus simples
- Maintenance facilitée
- Collaboration entre développeurs plus fluide

Structure recommandée d'un projet FastAPI

Les demandes sont envoyées par le client (navigateur, mobile, etc.) vers l'API. L'API les intercepte via des **routes** définies dans les **routers**. Les données reçues sont validées à l'aide des **schémas**, puis transmises à la **logique métier** (services). Les **services** nettoient, transforment ou enrichissent les données, puis délèguent l'accès à la base via les **repositories**, qui contiennent la logique SQL. Ces repositories utilisent les **models**, qui représentent les tables de la base. Une fois le traitement terminé, la réponse suit le chemin inverse jusqu'au client.

Pour faciliter la compréhension des interactions de l'environnement, voici un schéma de base pour un projet FastAPI :


API Structure

Voici une structure de base claire et courante :

```
app/
├── main.py               # Point d'entrée de l'application
├── routes/               # Les routes de l'API
├── services/             # La logique métier
├── repositories/         # Accès aux données
├── models/               # Les schémas ORM
├── db/                   # Connexion à la base ou session
│   └── session.py
├── conf/                 # Paramètres globaux (config, sécurité)
│   └── config.py
```

Chaque répertoire a une **responsabilité claire** :

- `routes/` : tout ce qui expose une API
- `models/` : définition des objets métier
- `db/` : gestion de la base de données (connexion, session)
- `conf/` : configuration globale de l'application (ex. variables d'environnement, secrets)
- `repositories/` : pour définir les accès aux données
- `services/` : pour la logique métier

> Les dossiers `services/` et `repositories/` sont optionnels mais recommandés pour une séparation claire entre la logique métier et l'accès aux données.

`APIRouter` : séparer les routes

Afin d'éviter de mélanger tous les models et routes associées, FastAPI propose de **séparer les routes** en modules grâce à `APIRouter`. Cela permet de regrouper les routes par fonctionnalité ou domaine, rendant le code plus lisible et maintenable.

```
# app/routes/user.py
from fastapi import APIRouter

router = APIRouter()

@router.get("/users")
def list_users():
    return [{"name": "Alice"}, {"name": "Bob"}]
```

Puis dans `main.py`, tu **montes le routeur** :

```
# app/main.py
from fastapi import FastAPI
from app.routes import user

app = FastAPI()

app.include_router(user.router)
```

Résultat :

- `/api/users` → route accessible
- Tu peux regrouper toutes les routes liées à un domaine (`/user`, `/product`, etc.)

Autres bonnes pratiques

1. Ajouter des tags sur les routers

Au lieu de mettre le tag dans chaque route, tu peux le mettre directement sur le router. Cela permet de regrouper les routes dans la documentation Swagger.

```
app.include_router(user.router, tags=["users"])
```

2. Ajouter des préfixes

Ajouter un préfixe aux routers permet de regrouper les routes par fonctionnalité ou domaine et simplifier la lecture dans les routes. Par exemple, pour les utilisateurs :

```
app.include_router(user.router, prefix="/api")
```

Cela permet de structurer les routes de manière logique, d'éviter les répétitions ainsi que les conflits de noms.

3. autres bonnes pratiques

- **Utiliser** `__init__.py` dans les dossiers pour rendre les modules importables.
- **Séparer** `schemas/` **et** `models/` si tu utilises une base SQL et pas SQLModel (Pydantic ≠ ORM).
- **Mettre les tests dans un dossier** `tests/`, à la racine du projet.