diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exercices – API REST avec FastAPI et SQLModel

Ces exercices ont pour objectif de vous faire pratiquer la conception et l'implémentation d'une API REST complète en utilisant FastAPI et SQLModel en vu du projet `Développement d'application informatique`.

Chaque exercice est divisé en plusieurs parties. Certaines sont théoriques et doivent être réalisées avant toute écriture de code, afin de bien préparer la suite.

L'enchaînement des exercices suit une progression logique :

- on commence par des concepts simples et isolés,
- puis on ajoute progressivement de nouveaux concepts, fonctionnalités et bonnes pratiques

Sommaire

Exercice 1 – Analyse et conception d'une API REST Utilisateurs

Tout d'abord, on va réfléchir à la conception d'une API REST simple pour gérer des utilisateurs.

Partie 1 – Analyse critique d'une API

On vous fournit les endpoints `User` d'une API REST existante :

```
POST /getUser
GET /deleteUser?id=12
POST /updateUser
```

Identifier et expliquer brièvement pourquoi ces endpoints ne respectent pas les bonnes pratiques REST.

Les points suivants doivent notamment être abordés :

- l'utilisation des méthodes HTTP
- la structure des URLs
- la sémantique REST (ressources vs actions)
- le caractère complet ou incomplet des endpoints proposés

Partie 2 – Refonte et conception d'une API REST

Vous devez maintenant **reconcevoir cette API REST correctement** pour gérer des **utilisateurs**.

1. Proposer les endpoints REST permettant de :
    - créer un utilisateur
    - récupérer tous les utilisateurs
    - récupérer un utilisateur unique
    - mettre à jour un utilisateur
    - supprimer un utilisateur
2. Pour chaque endpoint, préciser :
    - la méthode HTTP
    - l'URL
    - le(s) code(s) HTTP attendu (succès/erreur)
3. Donner un exemple de payload JSON utilisé pour la création d'un utilisateur.

> 👉 Un utilisateur est identifié de manière **unique par un `id`**.
> Toute récupération, modification ou suppression d'un utilisateur doit se faire à partir de cet identifiant.

Exercice 2 – Mise en place de l'environnement projet

Partie 1 – Création de l'environnement

1. Créer et activer un environnement virtuel Python dédié au projet.
2. Installer les dépendances nécessaires :
   - FastAPI, SQLModel, Uvicorn (serveur ASGI)
   - Toute autre bibliothèque utile (ex : pymysql, python-dotenv, etc.)
3. Figer les dépendances dans un fichier `requirements.txt`.
4. Mettre en place une structure de projet claire et organisée.

```
mon-projet-api/
├── src/              # code source de l'application
│   ├── main.py       # point d'entrée de l'application
│   ├── models/       # models et schémas SQLModel
│   ├── repositories/ # logique de manipulation des models
│   ├── services/     # logique métier
│   ├── routes/       # endpoints FastAPI
│   ├── conf/         # gestion de la configuration (Ex: .env, bdd, etc.)
│   └── utils/        # fonctions utilitaires
├── tests/
│   ├── conftest.py   # configuration des tests unitaires
│   └── test_*.py     # tests unitaires d'un model jusqu'à l'endpoint
├── .env              # variables d'environnement (ex : BDD credentials)
├── requirements.txt  # liste des dépendances
└── README.md         # documentation du projet
```

Partie 2 – MVP

1. Un endpoint FastAPI `/health` dans `src/main.py` qui retourne un succes avec le message `"status": "ok"`.

```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/health")
async def health_check():
   return {"status": "ok"}
```

2. Lancer l'application avec Uvicorn et vérifier que l'endpoint fonctionne : `uvicorn src.main:app --reload`
3. Accéder à l'interface Swagger via l'URL `/docs` et vérifier que l'endpoint `/health` y est bien listé. http://localhost:8000/docs

Exercice 3 – Développement de l'API REST avec FastAPI

Ensuite, on va implémenter l'API REST conçue précédemment en utilisant FastAPI.

Voici un payload JSON typique utilisé pour créer un utilisateur :

```
{
  "email": "user@test.com",
  "full_name": "John Doe",
  "age": 32,
  "is_active": true
}
```

Partie 1 – Analyse des données

> ⚠ Cette partie réflexion est à réaliser **avant toute écriture de code**.

1. Identifier les champs qui doivent être stockés en base de données.
2. Pour chaque champ, indiquer :
    - le type de données **SQLModel** approprié
    - la clé primaire
    - les contraintes évidentes (unique, nullable, valeurs obligatoires, etc.)

Partie 2 – API et sécurité des données

> ⚠ Toujours sans code, réponse attendue sous forme de points

Expliquer quels **risques potentiels** (fonctionnels et/ou sécurité) peuvent exister si :

- l'API accède directement aux tables de la base de données
- ou exécute des requêtes SQL construites à partir des données reçues des clients

Partie 3 – Modélisation avec SQLModel

1. Dans le dossier `src/models/`, créer un fichier `user.py`.

2. Ensuite, proposer un ou plusieurs **schémas SQLModel** adaptés à cette API, en distinguant si nécessaire

    - `UserBase` *(super-classe abstraite)*
      Champs communs à tous les schémas
    - `UserCreate` *(sous-classe conrete de UserBase)*
      Champs nécessaires à la création (ex : ajout d'un mot de passe)

- `UserPatch` *(sous-classe conrete de UserBase)*
  Champs utilisés pour la mise à jour (tous optionnels)
- `User` *(sous-classe conrete de UserBase)*
  Schéma complet correspondant à la table réelle en base de données

> ❌ Il n'est **pas nécessaire** de créer un schéma `UserDelete`.
> Les schémas ne servent que si un payload est attendu (création, mise à jour, etc.)

> 💡 Il peut parfois être intéressant de faire un `UserGet` (sous-classe de UserBase).
> Ce sera un équivalent au Schéma `User`, mais sans champs sensibles (ex : password)

🧩 Partie 4 – Implémentation FastAPI

1. Dans le dossier `src/routes/`, créer un fichier `user_routes.py`.
2. Ensuite, implémenter les endpoints FastAPI définis dans l'exercice 1 Partie 2 :
   - les schémas SQLModel définis précédemment
   - une logique claire de séparation entre API et base de données
3. Pour finir, importer et inclure les routes dans `src/main.py` afin qu'elles soient accessibles.

> 📌 *Aucune gestion d'authentification, de pagination ou de permissions n'est demandée.* On mettra en place les retours HTTP appropriés (codes et messages) pour les cas de succès et d'erreurs courants (ex : utilisateur non trouvé) plus tard : Exercice 4.

🧩 Partie 5 - Tester avec Swagger

Afin d'éviter les bug et la dette technique, il est important de tester morceau de code au fur et a mesure du développement.

Pour tester les endpoints, comme l'exercice 2, on utilisera l'interface Swagger auto-générée par FastAPI :

1. Lancer l'application FastAPI
2. Accéder à l'interface Swagger via l'URL `/docs`
3. Pour chaque endpoint implémenté :
   - Effectuer une requête de test (avec payload si nécessaire)
   - Vérifier que la réponse est conforme aux attentes

> 💡 Vous pouvez voir vos schémas tout en bas avec les champs obligatoires, optionnelles, les types des champs, leurs contraintes...

🩹 Exercice 4 – Mise en place de l'ORM avec SQLModel

Maintenant que l'API REST de gestion des utilisateurs est en place avec FastAPI, on va intégrer SQLModel pour gérer la persistance des données.

> On gérera le lien entre les 2 dans le prochain exercice.

En vous basant sur schéma `User` qui est la vrai table effective en bdd, réaliser les tâches suivantes :

🧩 Partie 1 – Compréhension du cycle de session

En pseudo-code, décrire les étapes nécessaires pour :

- Créer l'engine de connexion à la base de données
- Ouvrir une session SQLModel
- Récupérer tous les utilisateurs
- Fermer proprement la session

> Le pseudo-code doit faire apparaître clairement ces différentes étapes et les étapes intermédiaires nécessaires
> *(ex : exécution de la requête, récupération des résultats, etc.)*

🧩 Partie 2 – Implémentation en Python

On va implémenter le pseudo-code précédent en Python, en suivant les bonnes pratiques.

1. Dans le dossier `src/conf/`, créer un fichier `setting.py` pour gérer la configuration de l'application.
    - Charger les variables d'environnement depuis le fichier `.env`
    - créer la chaîne de connexion à la base de données
    - ...
2. Dans le dossier `src/conf/`, créer un fichier `session.py`
    - Importe la chaine de connexion depuis `setting.py`
    - Crée l'engine SQLModel
    - ...
3. Dans le dossier `src/repositories/`, créer un fichier `user_repository.py`
    - Importe le model `User`
    - Importe l'engine SQLModel depuis `session.py`
    - Implémente le CRUD complet pour le modèle `User`
    - ...

> 💡 Pour tester votre code, vous pouvez importer `user_repository.py` dans le `src/main.py` et appeler les fonctions de manipulation des utilisateurs.

> 💡 N'oubliez pas le `sqlmodel.metadata.create_all(engine)` dans le main pour créer la table dans la bdd.

🩹 Exercice 5 – Liaison SQLModel & FastAPI + gestion des erreurs

Maintenant que l'on a un SQLModel et un FastAPI fonctionnels, il est temps de les lier ensemble et de gérer les erreurs possibles.

🧩 Partie 1 – Liaison entre FastAPI et SQLModel

Pour faciliter l'injection des dépendances et l'écriture de tests unitaires, nous allons **injecter la session SQLModel directement dans les endpoints** FastAPI.

- La fonction `get_session` ouvrira une session SQLModel et utilisera `yield` pour la maintenir ouverte le temps de l'opération.
- Chaque endpoint recevra la session via `Depends`.

```
from fastapi import Depends
from sqlmodel import Session
from database import get_session

@app.get("/users/{user_id}")
async def get_user(user_id: int, session: Session = Depends(get_session)):
    ...
```

1. Créer la fonction `get_session` dans `src/database/session.py` qui :
    - ouvre une session SQLModel
    - la `yield` pour l'injection
    - ferme correctement la session après utilisation
2. Modifier vos endpoints FastAPI pour **injecter la session SQLModel** via `Depends(get_session)`.
3. Adapter vos fonctions de manipulation (CRUD) des utilisateurs pour **utiliser la session reçue en paramètre**.

🧩 Partie 2 – Mapping des erreurs BDD vers des réponses API

Certaines opérations peuvent échouer. Nous allons gérer deux cas typiques :

1. **GET /users/{user_id}** : l'utilisateur demandé n'existe pas
2. **POST /users** : insertion échouée (ex : email déjà utilisé)

Pour chaque cas :

- Identifier l'erreur métier
- Associer un **code HTTP approprié**
- Définir un **message API clair et compréhensible**

> Exemple attendu pour GET inexistant : Code HTTP : `404` Message : `"Utilisateur non trouvé"`

> Exemple attendu pour POST avec email déjà utilisé : Code HTTP : `409` Message : `"Email déjà utilisé"`

🩹 Exercice 6 – One-to-Many et Many-to-Many avec SQLModel

Dans la continuité des exercices précédents, nous enrichissons le contexte de gestion des **utilisateurs** en ajoutant deux nouvelles entités :

1. **Relation One-to-Many**
   - Un **utilisateur** peut créer plusieurs **tickets**
   - Un **ticket** appartient à un seul **utilisateur**
2. **Relation Many-to-Many**
   - Un **ticket** peut être associé à plusieurs **étiquettes**
   - Une **étiquette** peut être associée à plusieurs **tickets**

🧩 Partie 1 – Modélisation avec SQLModel

1. Créer les **schémas et modèles SQLModel** suivants :
   - `Ticket` : `id`, `titre`, `description`, `id_utilisateur`
   - `Tag` : `id`, `nom`
   - Une table d'association **Many-to-Many** `ticket_tag` entre `Ticket` et `Tag`
2. Définir correctement les relations :
   - `User` → `tickets` (**One-to-Many**)
   - `Ticket` ↔ `Tag` (**Many-to-Many**)
3. Ajouter les contraintes importantes :
   - clés primaires
   - règles de nullabilité adaptées
   - ...

> ⚠ Pour les relations Many-to-Many, **seule la table d'association est attendue**.
> Aucun schéma dédié n'est nécessaire.

🧩 Partie 2 – Concrétisation via l'API

1. Créer les **5 endpoints CRUD** pour :
   - les **tickets**
   - les **étiquettes (tags)**
2. Mettre en place la **logique métier** dans le dossier `src/services/` afin de :
   - gérer correctement les relations entre entités
   - inclure les données liées (utilisateur, étiquettes) dans les réponses lorsque cela est pertinent
3. Tester les endpoints à l'aide de **Swagger UI** afin de vérifier :
   - la création des relations
   - la récupération correcte des données liées
   - le bon fonctionnement global des relations One-to-Many et Many-to-Many