SQL & API REST - Cours complet pour devenir opérationnel

Ce support a été conçu pour apprendre **SQL + API REST avec Python** depuis zéro, avec une vision claire, synthétique, et directement exploitable en contexte professionnel (ESN, industrie, tech).

🔹 1. SQL - Base de données relationnelle

✅C'est quoi ?

Un système pour **stocker, organiser, et interroger des données** structurées dans des **tables**.

✅Pourquoi on l'utilise ?

Pour stocker durablement les infos (machines, utilisateurs, projets...), les retrouver, les trier, les croiser, les modifier.

✅Comment ça fonctionne ?

Une base SQL contient :

- **Tables** (ex: `machines`)
- **Colonnes** = attributs (ex: `name`, `sector`)
- **Lignes** = enregistrements (valeurs concrètes)
- **Clés primaires (PK)** : identifiants uniques
- **Clés étrangères (FK)** : liens entre tables

✅Commandes essentielles (DDL & DML)

Commande	Rôle	Exemple
`CREATE TABLE`	Créer une table	`CREATE TABLE IF NOT EXISTS machines (...)`
`INSERT`	Ajouter une ligne	`INSERT INTO machines (...) VALUES (...)`
`SELECT`	Lire des lignes	`SELECT * FROM machines`
`UPDATE`	Modifier une ligne	`UPDATE machines SET sector = '1A' WHERE name = 'M001'`
`DELETE`	Supprimer une ligne	`DELETE FROM machines WHERE name = 'M001'`

✅Types SQL (et Python associé)

Type SQL	Type Python	Description
`TEXT`	`str`	Chaine de caractères
`INTEGER`	`int`	Nombre entier
`BOOLEAN`	`bool` (0/1)	Vrai/faux
`REAL`	`float`	Nombre décimal

🔹 2. `sqlite3` - Utiliser SQL en Python

✅C'est quoi ?

Un module standard Python pour manipuler une base de données **SQLite** (locale, fichier `.db`).

✅Pourquoi on l'utilise ?

- Zéro installation serveur (tout tient dans un fichier)
- Léger, rapide, parfait pour prototyper ou outils locaux

✅Comment ça marche ?

```
import sqlite3

with sqlite3.connect("machineMonitor.db") as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM machines WHERE in_service = ?", (1,))
    rows = cursor.fetchall()
```

✅Décomposition :

- `import sqlite3` : charge le module
- `connect(...)` : ouvre la base (crée le fichier si besoin)
- `with ... as conn` : garantit que la base sera bien fermée (même en cas d'erreur)
- `conn.cursor()` : crée un "curseur" pour exécuter les commandes SQL
- `execute(...)` : envoie une commande SQL à la base
- `?` : paramètre de sécurité anti-injection (remplacé par les valeurs à droite)
- `fetchall()` : récupère les résultats

🔹 3. API REST - Avec FastAPI

✅C'est quoi ?

Une **interface web standardisée** pour permettre à des applications clientes (navigateur, frontend, app mobile, etc.) d'accéder aux données.

✅Pourquoi ?

- Pour **communiquer** entre frontend et backend
- Pour **exposer** une base SQL de façon contrôlée

✅Comment ça fonctionne ?

- Serveur HTTP (FastAPI) expose des **routes** : `/machines`, `/logs`, etc.
- Chaque route accepte un **verbe HTTP** : `GET`, `POST`, `PUT`, `DELETE`
- La réponse est en JSON

✅Installation

```
pip install fastapi uvicorn pydantic
```

✅Exemple de base

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Machine(BaseModel):
    name: str
    sector: str

@app.get("/machines")
def readMachines():
    return [{"name": "M001", "sector": "1A"}]
```

✅Décomposition

- `FastAPI()` : crée ton app
- `@app.get("/machines")` : décore une fonction = route `GET /machines`
- `BaseModel` : valide les données entrantes/sortantes automatiquement
- `uvicorn main:app --reload` : lance le serveur de dév

🔹 4. `BaseModel` (Pydantic)

✅C'est quoi ?

Un modèle de donnée à structure **fortement typée**, validé automatiquement.

✅Pourquoi ?

- Garantit que l'entrée utilisateur est correcte
- Documente la structure de donnée (JSON attendu)
- Convertit automatiquement les types (str → bool, etc.)

✅Exemple

```
class Log(BaseModel):
    uuid: str
    machineName: str
    type: str
    project: str
```

Toute requête ou réponse passant par ce modèle sera **validée**.

🔹 5. Décorateurs FastAPI (`@app.get`, `@app.post`, etc.)

✅C'est quoi ?

Un **marqueur Python** qui transforme une fonction en **route HTTP** dans FastAPI.

✅Pourquoi ?

Pour dire :

- Cette fonction = répond à une requête `GET`
- Sur l'URL `/machines`

✅Construction

```
@app.get("/machines")
def readMachines():
    ...
```

- `@app.get(...)` = GET
- `@app.post(...)` = POST
- `@app.put(...)` = PUT
- `@app.delete(...)` = DELETE

Derrière, FastAPI gère le routing, la doc, la validation...

🔹 6. `curl` - Envoyer une requête HTTP en ligne de commande

✅C'est quoi ?

Un outil terminal pour tester une API.

✅Pourquoi ?

- Pour tester les routes sans interface graphique
- Pour simuler des appels frontend/backend

✅Commandes courantes

```
# GET (lecture)
curl http://localhost:8000/machines

# POST (création)
curl -X POST http://localhost:8000/machines \
     -H "Content-Type: application/json" \
     -d '{"name": "M002", "sector": "1A"}'

# DELETE
curl -X DELETE http://localhost:8000/machines/M002
```

✅Explication

- `-X` : méthode HTTP (POST, DELETE...)
- `-H` : header (ici on dit qu'on envoie du JSON)
- `-d` : body (le contenu à envoyer)

🔹 7. Structure pro à connaître

Fichier / dossier	Rôle
`main.py`	Point d'entrée FastAPI
`models.py`	Pydantic BaseModel
`sqlLib.py`	Fonctions de base de données
`core.py`	Logique métier (transformations, filtres...)
`test_*.py`	Fichiers de test
`README.md`	Doc projet

Tu es maintenant prêt à développer un backend complet avec validation, persistance SQL, documentation auto, et test API via curl ou Swagger.