Exposer ses données via une API REST sécurisée

Amina MARIE

Solutions d’échange inter-applicatif

Ce module présente différentes méthodes d'échange de données entre systèmes (interne ou publics). Il se focalise particulièrement sur l'utilisation d'APIs en présentant leur fonctionnement et les méthodes courantes pour communiquer avec elles (récupérer ou envoyer des données).

- Partie 1 – Exposer ses données via une API REST sécurisée
- Partie 2 – Requêter des données et construire un mini ETL
- Partie 3 – Utiliser des API Push
- Partie 4 – Publier des données avec un POST

Projet fil rouge – API de gestion de personnages de manga

Objectif global : Construire une API complète de gestion de personnages fictifs (type Olive et Tom) avec des fonctionnalités :

- Exposition sécurisée (GET /personnages)
- Scripts de requêtage
- Simulation d'événements (push)
- Envoi de données à l'API (POST /scores)
- Chaque partie du cours alimente et améliore ce projet en y ajoutant une brique fonctionnelle.

Plan de la 1ère partie

Objectifs pédagogiques

- Comprendre ce qu’est une API et à quoi elle sert
- Identifier les types d’API les plus courants
- Comprendre les méthodes HTTP
- Savoir pourquoi exposer ses données via une API est préférable à un accès direct à une base de données
- Découvrir les bases du développement d'une API REST
- Apprendre les méthodes de sécurisation d’une API

Plan de la 1ère partie

- Qu'est-ce qu'une API ?
- Types d'API
- Méthodes HTTP
- Pourquoi utiliser une API plutôt que donner un accès à la base de données ?
- Développer une API REST simple avec FastAPI
- Sécuriser une API – Authentification par token
- Exercices

Qu’est-ce qu’une API

Définition

Une **API** est une interface qui permet à un logiciel d'en appeler un autre pour obtenir ou envoyer des données, sans en connaître le fonctionnement interne.

Analogie :

Tu es au restaurant. Tu donnes ta commande au serveur (l'API), il transmet à la cuisine (le serveur), puis te ramène ton plat (la réponse).

Exemple d’utilisation d’API

- Quand tu ouvres Google Maps et que tu vois la météo actuelle = l'app utilise une **API météo**.
- Une application mobile qui affiche des produits utilise l’**API du site e-commerce**.

Exemple – API publique

```python
import requests

response = requests.get("https://api.coindesk.com/v1/bpi/currentprice.json")
data = response.json()
print(data["bpi"]["EUR"]["rate"])  # Affiche le prix du Bitcoin en euros
```

Types d’API

3 types d’API

Type	Description	Cas d’usage
REST	Le plus courant, basé sur HTTP	Applis web/mobiles
SOAP	Standard XML, formel	Secteurs bancaires
GraphQL	Flexible, orienté requête	Réseaux sociaux, Applis complexes

Exemples

Exemple REST

```
GET /users/12 → renvoie le profil utilisateur avec l'ID 12
```

Exemple GraphQL

```
query {
  user(id: 12) {
    name
    email
  }
}
```

Les méthodes HTTP

5 méthodes HTTP

Méthode	Action	Exemple d'API REST
GET	Lire une ressource	Liste tous les personnages
POST	Créer une ressource	Ajouter un nouveau personnage
PUT / PATCH	Modifier une ressource (entière ou partiellement)	Change le score d'un perso
DELETE	Supprimer une ressource	Retire un personnage

Exemple avec requests

```
import requests
new_data = {"name": "Ben Becker"}
response = requests.post("http://localhost:8000/personnages", json=new_data)
```

Pourquoi créer une API au lieu de donner un accès direct à la base

Risques avec un accès direct

- Toute l'application cliente a **accès total à la BDD**

  Faille de sécurité

- Impossible de limiter les actions ou les droits

  Trop de droits accordés

- La structure de la base est **exposée**

Avantage d’une API

- **Filtrage** et **contrôle** des accès

  Encapsulation des règles métiers

- **Sécurité** renforcée des accès (tokens, restrictions IP, CORS…)
- Possibilité de **logger, tester et documenter**

Exemple :

Tu peux autoriser une app mobile à consulter les personnages sans lui permettre de les modifier ou de voir les données sensibles.

Développer une API REST simple avec FastAPI

La structure d’un API REST

Qu’est-ce qu’un *endpoint* ?

Un **endpoint** est une "porte d'entrée" de ton API. C’est une **URL** (comme /produits, /clients, /articles) qui répond à une méthode HTTP (GET, POST…).

GET /livres → obtenir la liste des livres

GET /livres/2 → obtenir le livre avec l’ID 2

POST /livres → ajouter un nouveau livre

Organisation minimale pour un projet API

```
mon_api/
├── main.py              ← Point d'entrée de l'API
├── personnages.json     ← Données statiques (simulées)
├── .gitignore           ← Fichiers à ignorer
└── README.md            ← Instructions (facultatif)
```

Dans main.py, on retrouve :

- la création de l'app (app = FastAPI())
- les routes (@app.get(...))
- éventuellement des validations

Bonnes pratiques dès le début

- Nommer clairement les fonctions : get_livres, add_utilisateur, etc.
- Retourner **des listes ou des dictionnaires JSON**
- Ajouter des commentaires pour chaque bloc de code
- Tester très souvent avec Swagger (URL : /docs)

Choisir sa bibliothèque pour créer une API en Python

- Flask

Avantages :

- Très simple à démarrer
- Documentation abondante

Inconvénients :

- Pas de documentation automatique
- Moins moderne (pas de typage, ni gestion native des requêtes)

Choisir sa bibliothèque pour créer une API en Python

- FlaskAPI

Avantages majeurs :

- Swagger intégré automatiquement
- Gestion automatique des erreurs
- Typage clair avec str, int, etc.
- Gestion simplifiée des paramètres (Query, Header, Body)

Choisir sa bibliothèque pour créer une API en Python

- Flask

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/films", methods=["GET"])
def get_films():
    return jsonify([{"id": 1, "titre": "Dune"}])
```

- FastAPI

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/films")
def get_films():
    return [{"id": 1, "titre": "Dune"}, {"id": 2, "titre": "Tenet"}]
```

Les étapes

- Créer un fichier main.py
- Installer FastAPI + Uvicorn : pip install fastapi uvicorn
- Exemple – API de recettes

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/recettes")
def get_recettes():
    return [
        {"id": 1, "nom": "Tarte aux pommes"},
        {"id": 2, "nom": "Curry de légumes"}
    ]
```

Les étapes

- Lancer le serveur:

  uvicorn main:app --reload

- Accède à http://localhost:8000/recettes

Documentation Swagger

Une fois l'API en route :

- Accède à http://localhost:8000/docs (Swagger UI)
- Ou à http://localhost:8000/redoc

Tu y trouveras :

- Tous les endpoints exposés
- Les paramètres attendus
- Les formats de réponse

Sécuriser une API – Authenfication par token

Protéger certains endpoints

Principe :

Un utilisateur doit envoyer un **token d'accès** via un header HTTP. Si le token est invalide, l'accès est refusé.

```python
from fastapi import FastAPI, Header, HTTPException

app = FastAPI()

@app.get("/devis")
def lire_devis(token: str = Header(...)):
    if token != "MYSECRET":
        raise HTTPException(status_code=401, detail="Accès refusé")
    return [{"id": 1, "montant": 480}, {"id": 2, "montant": 230}]
```

Authentification par token

Pourquoi un token ?

Imagine une API qui **affiche des devis ou des données personnelles.**
Sans sécurité, n'importe qui pourrait y accéder.

Étape	Côté client	Côté API
1	L'utilisateur s'identifie	Un token lui est envoyé
2	Il stocke ce token	…
3	Il l'envoie dans l'en-tête HTTP	L'API vérifie le token à chaque requête

Exemple dans FastAPI

Tester avec Swagger (champ token) ou Postman (header token: MON_TOKEN_SECRET)

```python
from fastapi import Header, HTTPException

@app.get("/secrets")
def lire_infos_secretes(token: str = Header(...)):
    if token != "MON_TOKEN_SECRET":
        raise HTTPException(status_code=401, detail="Accès non autorisé")
    return {"données": "très confidentielles"}
```

CORS – Autoriser les requêtes venant du navigateur

Quand une application web (ex: React, Angular…) veut interroger ton API, **CORS** (Cross-Origin Resource Sharing) doit être configuré.

Une API locale ou sur un autre domaine est bloquée par le navigateur **pour des raisons de sécurité**. CORS permet de dire : “j’autorise tel domaine à m’appeler”.

```python
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],  # App React, par exemple
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

CORS – Comment tester ?

Crée une petite page HTML/JS qui appelle l’API avec fetch

Sans CORS : tu verras une erreur dans la console du navigateur

Avec CORS activé : la requête passe

Restriction par adresse IP

FastAPI ne bloque pas les IPs par défaut. Pour le faire :

Option 1 : Utiliser un middleware personnalisé (Python)

```python
@app.middleware("http")
async def bloquer_ips(request: Request, call_next):
    client_ip = request.client.host
    if client_ip not in ["127.0.0.1", "192.168.0.1"]:
        raise HTTPException(status_code=403, detail="IP interdite")
    return await call_next(request)
```

Option 2 : Le faire côté serveur (Nginx, Apache)

Bloquer certaines IPs ou n’autoriser que des plages précises

Exercices

Exercice 1 – Créer une API REST

Objectif : afficher une liste statique de personnages fictifs

À faire :

- Crée un fichier main.py
- Initialise une app FastAPI
- Crée un endpoint GET /personnages
- Retourne une liste fixe de personnages (au moins deux)

Guides utiles :

- Tu peux t'inspirer de la structure de l'API de recettes.
- Utilise le serveur local (uvicorn) pour tester.

Exercice 2 – Sécurise l’accès avec un token

Objectif : seuls les utilisateurs avec le bon token accèdent aux personnages.

À faire :

- Ajoute un paramètre token via Header(...)
- Vérifie sa valeur
- Retourne une erreur 401 sinon

Guides utiles :

- Réutilise la logique vue dans l’API de devis
- Teste via Postman ou Swagger

Exercice 3 – Ajoute CORS

Objectif : rendre ton API accessible depuis une app frontend

Guides utiles :

- Reprends la configuration de l’exemple CORS
- Teste avec un front en HTML+JS ou React

Annexes

a) Comprendre la dépendance Header(...) dans FastAPI

Lorsqu'on écrit dans une fonction d'API : def ma_fonction(token: str = Header(...)):

FastAPI va chercher automatiquement un **header HTTP nommé token** dans la requête, et l'injecter dans la fonction.

Ce mécanisme fait partie du **système de dépendance de FastAPI** : la fonction ne gère pas manuellement les requêtes HTTP, FastAPI s'en charge pour vous.

Concrètement :

- Le client envoie un header HTTP : token: SECRET123
- FastAPI lit ce header et le passe à ta fonction Python.

Tu peux afficher ce que FastAPI a récupéré : print(f"Token reçu : {token}")

Annexes

b) Comprendre HTTPException

Si un problème se produit (ex : token invalide), on peut **arrêter la fonction** en renvoyant une **erreur standardisée** au client.

💻 **Exemple :**

```
raise HTTPException(status_code=401, detail="Token invalide")
```

Cela renvoie une **réponse HTTP** comme :

```
{ "detail": "Token invalide" }
```

et le **code HTTP 401 Unauthorized**.

Annexes

b) Comparaison avec une condition classique

Sans HTTPException	Avec HTTPException
On continue l'exécution même si le token est mauvais	On interrompt la fonction immédiatement
Moins clair pour l'utilisateur	Plus explicite, respect du protocole HTTP

Merci