TUTO : Sécuriser un endpoint FastAPI avec un jeton JWT (Bearer Token)

Objectif

À la fin de ce TP, tu seras capable de :

- réer un **endpoint** `/token` qui génère un **JWT signé**.
- Protéger un **endpoint** `/animaux/noms` à l'aide de ce jeton.
- Vérifier le **payload** du JWT pour authentifier la requête.

Étape 1 — Comprendre les bases

Qu'est-ce qu'un JWT ?

Un **JSON Web Token** (JWT) est une chaîne de caractères composée de **trois parties encodées en Base64** :

```
header.payload.signature
```

Exemple :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhbGljZSIsImV4cCI6MTcwNzQ3MjkzNn0.k4AhHV2
```

- **Header** → type de token + algorithme (`HS256`)
- **Payload** → les données transportées ("claims") ex : utilisateur, expiration, rôles
- **Signature** → garantit l'intégrité du token (calculée avec une clé secrète)

Ressources :

- Introduction à JWT – Auth0
- JWT.io — visualiser un token

Qu'est-ce qu'un Bearer Token ?

Un **Bearer Token** est un jeton transmis dans le header HTTP pour prouver l'identité d'un utilisateur :

```
Authorization: Bearer <ton_token_jwt>
```

Ressources :

- Swagger — Bearer Authentication

Étape 2 — Créer la structure du projet

Arborescence minimale :

```
.
├── main.py
├── models/
```

```
|   └── animal.py
└── dto/
    └── services.py
```

Vous retrouverez le code exemple sur le github suivant : [API Python]

Étape 3 — Créer l'application FastAPI et la clé de sécurité

Dans `main.py` :

```python
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
from models.animal import Animal
from dto.services import AnimalService

app = FastAPI()

# Clé secrète et algorithme pour signer le JWT
SECRET_KEY = "super_secret_key"  # à stocker dans une variable d'environnement en prod
ALGORITHM = "HS256"
security = HTTPBearer()  # schéma Bearer pour Swagger
```

Les dépendances :

- `FastAPI`, `Depends`, `HTTPException`, `status` : base de l'app et système de dépendances/erreurs.
- `HTTPBearer`, `HTTPAuthorizationCredentials` : **schéma de sécurité** "Bearer token" prêt à l'emploi. Il fait deux choses :
  1. lit l'en-tête `Authorization: Bearer <token>`
  2. refuse la requête (401) si l'en-tête est absent/mal formé.
- `jose.jwt`, `JWTError` : encodage/décodage de JWT + gestion d'erreurs.

Les déclarations :

- `app` : instance FastAPI.
- `SECRET_KEY` : clé de **signature** du token (doit être stockée en **variable d'environnement** en prod).
- `ALGORITHM` : algo de signature du JWT (HS256 = HMAC-SHA256).
- `security = HTTPBearer()` : crée l'objet sécurité utilisé comme **dépendance** pour extraire le token depuis l'en-tête `Authorization`.

Étape 4 — Créer une fonction de vérification du jeton

Cette fonction sera utilisée comme **dépendance** dans les endpoints à protéger.

```python
def verify_jwt_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
    token = credentials.credentials  # extrait la partie après "Bearer"
    try:
        # On décode le JWT avec la clé secrète et l'algorithme
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
```

```
        return payload  # contient les données (ex: {"sub": "alice"})
    except JWTError:
        # Si le token est invalide ou expiré, on renvoie une erreur 401
        raise HTTPException(status_code=401, detail="Jeton invalide ou expiré")
```

- `Depends(security)` : injecte `credentials` **si et seulement si** l'en-tête `Authorization: Bearer …` est présent et bien formé.
- `credentials.scheme` vaudrait `"Bearer"`, `credentials.credentials` contient la chaîne du jeton.
- `jwt.decode(...)` :
  - vérifie l'intégrité (signature) avec `SECRET_KEY`.
  - décode et renvoie le **payload** (dict) si tout est OK.
  - lèvera une `JWTError` si signature invalide, token altéré, horodatage non conforme, etc.
- En cas d'erreur → `401 Unauthorized`.

> *Remarque : le code n'impose pas d'expiration (`exp`). C'est OK pour une démo, mais en prod on recommande d'ajouter `exp` au moment de l'émission et de le vérifier au décodage (la lib `python-jose` le fait automatiquement si `exp` est présent).*

Étape 5 — Créer un endpoint pour générer un token

Cet endpoint simule une connexion : on fournit un nom d'utilisateur et on obtient un jeton signé.

```
@app.post("/token")
def generate_token(username: str):
    data = {"sub": username}
    token = jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
    return {"access_token": token, "token_type": "bearer"}
```

- Endpoint minimal pour générer un JWT :
  - prend un `username` (paramètre de requête ou body x-www-form-urlencoded selon ton appel).
  - construit un payload avec `sub` (subject = identifiant utilisateur).
  - **encode** et **signe** le token avec la clé/algorithme.
  - renvoie un objet conforme aux usages : `{"access_token": "...", "token_type": "bearer"}`.
  - `sub` = "subject" → représente l'identité de l'utilisateur.

> *En prod : on **validerait** vraiment l'utilisateur (mot de passe, base de données), et on **ajouterait** des claims (`exp`, éventuellement `iat`, `aud`, `roles`, etc.).*

Le résultat ressemble à :

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5...",
  "token_type": "bearer"
}
```

Étape 6 — Créer le endpoint sécurisé `/animaux/noms`

```
@app.get("/animaux/noms")
def get_animal_names(payload: dict = Depends(verify_jwt_token)):
    noms = AnimalService.get_all_animal_names()
    return {"noms": noms, "user": payload}
```

- `Depends(verify_jwt_token)` : FastAPI appelle cette fonction avant d'exécuter le endpoint.
- Si le token est valide → `payload` est injecté automatiquement.
- Si le token est invalide → réponse 401.

Étape 7 — Tester dans Swagger

1. Lance le serveur :

   ```
   uvicorn main:app --reload
   ```

2. Ouvre http://127.0.0.1:8000/docs

3. Clique sur **POST /token** → "Try it out" → entre un `username`

4. Récupère la valeur du champ `"access_token"`.

5. Clique sur le bouton *Authorize * → colle ton token (sans le mot "Bearer").

6. Teste **GET /animaux/noms** → tu dois recevoir :

   ```
   {
     "noms": ["Chien", "Chat", "Lapin"],
     "user": {"sub": "ton_nom_utilisateur"}
   }
   ```

Résultat final

Code complet :

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
from models.animal import Animal
from dto.services import AnimalService

app = FastAPI()

SECRET_KEY = "super_secret_key"
ALGORITHM = "HS256"
security = HTTPBearer()

def verify_jwt_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
    token = credentials.credentials
```

```
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        raise HTTPException(status_code=401, detail="Jeton invalide ou expiré")

@app.post("/token")
def generate_token(username: str):
    data = {"sub": username}
    token = jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
    return {"access_token": token, "token_type": "bearer"}

@app.get("/")
async def read_root():
    return {"Hello": "World"}

@app.get("/animaux/noms")
def get_animal_names(payload: dict = Depends(verify_jwt_token)):
    noms = AnimalService.get_all_animal_names()
    return {"noms": noms, "user": payload}
```

Pour aller plus loin

- FastAPI — Sécurité avec JWT
- Auth0 — Comprendre le JWT
- JWT.io — Visualiseur de token
- Swagger — Authentification Bearer