Compte rendu TD4

Auteur : Hugo Soleau

Sommaire

Introduction

Dans ce TD, nous travaillons sur la création d'un serveur local auquel il est possible de requêter sur des données éventuellement importées ou extraites d'ailleurs.
Le but est de permettre à des utilisateurs de se créer un compte, de choisir une licence et d'accéder aux données selon les autorisations accordées par sa licence.

Afin de mener à bien ce travail, nous nous aidons d'un grand nombre de bibliothèques déjà disponibles sur python. Parmi ces biblliothèques, nous retrouvons *fastapi*, *uvicorn*, *jwt*, *bcrypt* afin de nous permettre la construction de notre API et la mise en place de système de sécurité entre autres.
Les bibliothèques typing et pydantic nous permettent quant à elles de jouer avec le format de nos données de façon pratique.
Enfin, pour la gestion des requêtages client et notamment la limitation dans le temps du nombre de requêtes effectuables par nos utilisateurs, nous utilisons différentes bibliothèques de *slowapi*.

```
# TD4_data.py

from typing import Literal
from pydantic import BaseModel, Field
...

# TD4_serv.py

from fastapi import FastAPI, Depends, HTTPException, status, Request
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
import jwt
import uvicorn
import bcrypt
from typing import Optional
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded
...

# TD4_cli.py
```

```
import requests
...
```

Note sur l'exécution du programme

Le serveur se démarre par l'exécution de la commande **uvicorn TD4_serv:app --reload** à partir du dossier source.

L'écriture des requêtes client se fait depuis le fichier *TD4_req.py*. Ce fichier est distinct du fichier *TD4_cli.py* dans lequel sont rédigées les requêtes employables par les utilisateurs.

Les données

Afin de pouvoir travailler, nous définissons les caractéristiques de nos données et de nos utilisateurs.

Les données financières

Les données qui intéresseront nos utilisateurs ici seront des données de RSI, MACD et de cryptomonnaies. Pour faire, simple, nous concevons un dictionnaire regroupant des valeurs fictives de RSI et MACD associées à des cryptomonnaies. Nous stockons dans ce même dictionnaire les noms de nos cryptomonnaies.

```
#Données de test variées
indicators = {
    "coins": ["BTC", "ETH", "ADA", "USD"],
    "RSI": [{
            "value": 42.7,
            "date": "2023-11-23",
            "crypto": "BTC"},
        {
            "value": 35.6,
            "date": "2023-12-23",
            "crypto": "BTC"},
        {
            "value": 37.9,
            "date": "2023-11-23",
            "crypto": "ETH"}],

    "MACD": [{
        "value": 0.0045,
        "date": "2023-12-23",
        "crypto": "ETH"},
        {
            "value": 0.0035,
            "date": "2023-10-23",
            "crypto": "BTC"},
        {
            "value": 0.0026,
            "date": "2023-11-23",
            "crypto": "ETH"}]
}
```

Dans un second dictionnaire, nous stockons les noms de nos monnaies et leurs symboles. Le but de notre démarche est uniquement de nous exercer sur différents formats et de varier les accès ensuite qu'auront nos utilisateurs.

```
#Données des monnaies
coins = {"coin" : [{"name": "BTC",
         "symbol" : "btc"},
         {"name": "ETH",
          "symbol" : "eth"},
         {"name": "ADA",
          "symbol" : "ada"},
         {"name": "USD",
          "symbol" : "usd"}]
    }
```

Les données utilisateurs

Afin de réguler le flux d'individus accédant à nos données et de gérer la visibilité de ces dernières, nous concevons un modèle de données afin de stocker en interne les comptes de ceux-ci.
Nous concevons alors une classe *User* à partir de la classe *BaseModel*. nous définissons nos utilsiateurs par les attributs usuels du nom, prénom, pseudo, adresse email, mot de passe. A ces attributs, nous ajoutons un attribut de licence qui est restreint aux valeurs "étudiant", "premium" et "entreprise". Cette distinction nous permettra notamment à définir des individus avec des types de requêtage différent selon la licence. Nous ajoutons également un attribut d'activation afin de pouvoir bannir éventuellement certains comptes utilisateurs.

```
#Modèle de notre utilisateur avec ses attributs
class User(BaseModel):
    nom: str
    prenom: str
    pseudo: str
    email: str
    password: str
    disabled: bool = False
    licence: Literal["étudiant", "premium", "entreprise"]

class UserInDB(User):
    hashed_password: str

#Modèle de licence permettant la modification de la valeur de la licence
utilisateur
class LicenceUpdate(BaseModel):
    licence: Literal["étudiant", "premium", "entreprise"] = Field(...,
description="Nouvelle licence de l'utilisateur")
```

Nb : On emploie la fonction *Field* dans l'intégration de notre modificateur de licence afin de pouvoir commenter tout changement de licence et ainsi permettre à l'utilisateur l'obtention d'un retour d'information de la bonne modification de celle-ci.

On conçoit un jeu de données utilisateurs pour notre utilisation.

```
#Données de nos utilisateurs
users = [
    {
        "nom": "Dupont",
        "prenom": "Jean",
        "pseudo": "jdupont",
        "email": "jean.dupont@example.com",
        "password" : "password_1",
        "disabled": False,
        "licence": "entreprise"
    },
    {
        "nom": "Martin",
        "prenom": "Alice",
        "pseudo": "amartin",
        "email": "alice.martin@example.com",
        "password" : "password_2",
        "disabled": False,
        "licence": "étudiant"
    },
    ...]
```

Le serveur

Création serveur et connexion

Pour la création de notre serveur API, nous utilisons la fonction *FasTAPI* que nous munissons d'un système de sécurité par identification via un token au moyen du schéma OAuth2.

```
app = FastAPI()
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

Nous définissons ensuite un ensemble de méthodes afin de permettre la création et la vérfication de notre token d'accès.

```
#Fonction de création du token d'accès
def create_access_token(data: dict):
    encoded_jwt = jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

#Fonction de vérification du token de l'utilisateur
```

```
def verify_token(token: str, credentials_exception):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        email: str = payload.get("sub")
        if email is None:
            raise credentials_exception
        return email
    except jwt.PyJWTError:
        raise credentials_exception

#Identification de l'utilisateur actuel via son token
async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid authentication credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    email = verify_token(token, credentials_exception)
    user = next((u for u in users if u['email'] == email), None)
    if user is None:
        raise credentials_exception
    return UserInDB(**user)
```

Puisque notre système de sécurité fonctionne par l'emploi de mots de passe hashés, nous définissons une fonction de hashage et une fonction de vérification qui compare les versions hashées de ceux-ci.

```
#Méthode de hashage du mot de passe utilisateur pour la mémorisation sécurisée
def hash_password(password: str) -> str:
    return bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()

#Vérifie les mots de passe par comparaison
def verify_password(plain_password: str, hashed_password: str) -> bool:
    return bcrypt.checkpw(plain_password.encode(), hashed_password.encode())
```

Il est important de noter que le hashage nous permet entre autres un stockage sécurisé du mot de passe.

Requêtes et accès

Avant de passer à l'écriture des commandes dans notre API, nous concevons un limiteur afin de gérer le nombre de requêtes effectuables par nos utilisateurs.

```
limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)
```

On définit ensuite une série de commandes *post*, *put* et *get* afin de permettre à nos utilisateurs de naviguer sur leur comtpe et entre les données.
Pour commencer, nous définissons des commandes de création de compte (*signup*) et de connexion (*login*).

```
@app.post("/token")
@limiter.limit("10/minute", key_func=rate_limit_key, error_message="Limite de
requêtes atteinte")
async def login(request: Request, form_data: OAuth2PasswordRequestForm =
Depends()):
    user = next((u for u in users if u['pseudo'] == form_data.username), None)
    if user and verify_password(form_data.password, user['hashed_password']) and
not user['disabled']:
        access_token = create_access_token(data={"sub": user["email"]})
        return {"access_token": access_token, "token_type": "bearer"}
    raise HTTPException(status_code=400, detail="Informations d'identification
incorrectes ou compte désactivé")

@app.post("/signup")
@limiter.limit("10/minute", key_func=rate_limit_key, error_message="Limite de
requêtes atteinte")
#Création de compte et ajout à la base de données
async def signup(user: User):
    # Vérifie si l'utilisateur existe déjà
    existing_user = next((u for u in users if u['email'] == user.email), None)
    if existing_user:
        raise HTTPException(status_code=400, detail="L'utilisateur existe déjà")

    # Hashe le mot de passe
    hashed_password = bcrypt.hashpw(user.password.encode(),
bcrypt.gensalt()).decode()

    # Crée un nouvel utilisateur
    new_user = {
        "nom": user.nom,
        "prenom": user.prenom,
        "pseudo": user.pseudo,
        "email": user.email,
        "hashed_password": hashed_password,
        "disabled": user.disabled,
        "licence": user.licence
    }

    # Ajoutez le nouvel utilisateur à la liste des utilisateurs
    users.append(new_user)
```

On limite à 10 le nombre de fois que ces requêtes sont appelables par minute afin de réduire le risque d'intrusion à partir d'un logiciel recherchant des identifiants et également de limiter la création automatisée de comptes factices.

Par ailleurs, on définit une méthode *put* pour permettre aux utilisateurs de changer leur niveau de licence.

```
@app.put("/user/licence")
#Modification de la licence de l'utilisateur
async def update_licence(request: Request, licence_update: LicenceUpdate,
```

```
current_user: UserInDB = Depends(get_current_user)):
    user = next((u for u in users if u['email'] == current_user.email), None)
    if user is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Utilisateur non trouvé")

    user['licence'] = licence_update.licence
    return {"message": "Licence mise à jour avec succès"}
```

Enfin, on définit une série de méthodes de lecture de nos données dont on limite le nombre à 10 requêtes par minute pour les utilisateurs non authentifiés via la fonction *rate_limit_key*.

```
#Fonction de limitation selon l'authentification
def rate_limit_key(request: Request) -> str:
    authorization: str = request.headers.get("Authorization")
    token = authorization.split()[1] if authorization else None

    if token:
        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
            email = payload.get("sub")
            user = next((u for u in users if u['email'] == email), None)
            if user:
                return f"licence:{user['licence']}"  # Clé basée sur la licence de
l'utilisateur
        except jwt.PyJWTError:
            pass

    return "unauthenticated"  # Clé pour les utilisateurs non authentifiés
```

En outre, on ajoute dans nos fonctions de lecture, la condition que les utilisateurs avec une licence étudiante ne peuvent pas récupérer les informations relatives au RSI et au MACD de nos monnaies.

```
@app.get("/indicators/rsi")
@limiter.limit("10/minute", key_func=rate_limit_key, error_message="Limite de
requêtes atteinte")
#Récupération et lecture des données RSI par l'utilisateur
async def read_rsi(request: Request, current_user: User =
Depends(get_current_user), data = indicators):
    user = next((u for u in users if u['email'] == current_user.email), None)
    if user and user['licence'] != "étudiant":
        return {f"data : {data['RSI']}"}
    raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Accès
interdit pour les utilisateurs avec une licence étudiante")


@app.get("/indicators/macd")
@limiter.limit("10/minute", key_func=rate_limit_key, error_message="Limite de
requêtes atteinte")
```

```
#Récupération et lecture des données MACD par l'utilisateur
async def read_macd(request: Request, current_user: User =
Depends(get_current_user), data = indicators):
    user = next((u for u in users if u['email'] == current_user.email), None)
    if user and user['licence'] != "étudiant":
        return {f"data : {data['MACD']}"}
    raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Accès
interdit pour les utilisateurs avec une licence étudiante")


@app.get("/coins/coin")
@limiter.limit("10/minute", key_func=rate_limit_key, error_message="Limite de
requêtes atteinte")
#Récupération et lecture des données des monnaies par l'utilisateur (peut stipuler
une monnaie précise)
async def read_coin(request: Request, name: Optional[str] = None, data = coins):
    if name:
        filtered_coins = [coin for coin in coins['coin'] if coin['name'] == name]
        return {f"{filtered_coins}"}
    else :
        return {f"{data['coin']}"}
```

Nb : Nous permettons à travers la fonction de lecture des monnaies de récupérer les informations spécifiques à une monnaie

Le client

Finalement, nosu définissons dans un fichier à part (**TD4_cli.py**) les fonctions appelant les requêtes créées en amont et affichant leur résultat dans la console.

```
#Fonction de récupération du token d'authentification de l'utilisateur
def get_jwt_token(username, password, BASE_URL):
    auth_url = f"{BASE_URL}/token"
    response = requests.post(auth_url, data={"username": username, "password":
password})
    if response.status_code == 200:
        return response.json()["access_token"]
    else:
        print("Erreur:", response.status_code, response.text)
        raise ValueError("Authentification échouée")


#Fonction pour mettre à jour la licence de l'utilisateur
def update_user_licence(token, new_licence, BASE_URL):
    headers = {"Authorization": f"Bearer {token}"}
    url = f"{BASE_URL}/user/licence"
    data = {"licence": new_licence}
    response = requests.put(url, headers=headers, json=data)
    if response.status_code == 200:
        return response.json()
    else:
```

```
        print("Erreur lors de la mise à jour de la licence:",
response.status_code, response.text)
        return None


#Fonction pour obtenir la licence de l'utilisateur
def get_user_licence(token, BASE_URL):
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.get(f"{BASE_URL}/user/licence", headers=headers)
    if response.status_code == 200:
        return response.json()
    else:
        print("Erreur lors de la récupération de la licence:",
response.status_code, response.text)
        return None


#Fonction pour obtenir les données d'un indicateur financier
def get_indicator_data(endpoint, token, BASE_URL):
    headers = {"Authorization": f"Bearer {token}"}
    response = requests.get(f"{BASE_URL}/{endpoint}", headers=headers)
    if response.status_code == 200:
        return response.json()
    else:
        print("Erreur:", response.status_code, response.text)
        return None


#Fonction pour obtenir les données d'une monnaie
def get_coin_data(endpoint, BASE_URL, name=None):
    url = f"{BASE_URL}/{endpoint}"
    if name:
        url += f"?name={name}"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()
    else:
        print("Erreur:", response.status_code, response.text)
        return None
```

Nb : Il est important de noter qu'un utilisateur n'a pas besoin de s'identifier avec la fonction *get_jwt_token* pour pouvoir effectuer des requêtes.