**Модуль 12: Авторизація та Аутентифікація**

**Заняття 2: Авторизація та Аутентифікація**

**Моделі і схеми валідації користувача**

Тепер, коли ми вже знаємо, як функціонує JWT, додамо авторизацію та аутентифікацію в наш RESTful API застосунок.

**Змінюємо моделі**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/intro#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%96)**file: src/database/models.py**

from sqlalchemy import Column, Integer, String, Boolean, func, Table, UniqueConstraint

from sqlalchemy.orm import relationship

from sqlalchemy.sql.schema import ForeignKey

from sqlalchemy.sql.sqltypes import DateTime

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

note\_m2m\_tag = Table(

"note\_m2m\_tag",

Base.metadata,

Column("id", Integer, primary\_key=True),

Column("note\_id", Integer, ForeignKey("notes.id", ondelete="CASCADE")),

Column("tag\_id", Integer, ForeignKey("tags.id", ondelete="CASCADE")),

)

class Note(Base):

\_\_tablename\_\_ = "notes"

id = Column(Integer, primary\_key=True)

title = Column(String(50), nullable=False)

created\_at = Column('created\_at', DateTime, default=func.now())

description = Column(String(150), nullable=False)

done = Column(Boolean, default=False)

tags = relationship("Tag", secondary=note\_m2m\_tag, backref="notes")

user\_id = Column('user\_id', ForeignKey('users.id', ondelete='CASCADE'), default=None)

user = relationship('User', backref="notes")

class Tag(Base):

\_\_tablename\_\_ = "tags"

\_\_table\_args\_\_ = (

UniqueConstraint('name', 'user\_id', name='unique\_tag\_user'),

)

id = Column(Integer, primary\_key=True)

name = Column(String(25), nullable=False)

user\_id = Column('user\_id', ForeignKey('users.id', ondelete='CASCADE'), default=None)

user = relationship('User', backref="tags")

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True)

username = Column(String(50))

email = Column(String(250), nullable=False, unique=True)

password = Column(String(255), nullable=False)

created\_at = Column('crated\_at', DateTime, default=func.now())

avatar = Column(String(255), nullable=True)

refresh\_token = Column(String(255), nullable=True)

Наші моделі створюють три таблиці в базі даних для зберігання нотаток "notes", тегів "tags" та користувачів "users", а також встановлюють відповідні відношення між ними.

Клас Note тепер містить зовнішній ключ user\_id, що встановлює зв'язок один до багатьох між користувачами та нотатками. Рядок user = relationship('User', backref="notes") створює зв'язок між класами Note та User і вказує, що зв'язок є зв'язком один до багатьох (у користувача може бути кілька нотаток, але кожна нотатка належить одному користувачеві). Аргумент backref створює зворотне посилання на клас User, дозволяючи вам отримати доступ до зв'язаних об'єктів Note з об'єкта User. Наприклад, ви можете використовувати user.notes для отримання всіх нотаток, пов'язаних із користувачем.

Клас Tag теж тепер має зовнішній ключ user до таблиці User, що встановлює зв'язок один до багатьох між користувачами та тегами, аналогічно класу Note. Крім того, існує унікальне обмеження на комбінацію імені та ідентифікатора користувача. Оскільки теги повинні бути унікальними, тепер не просто для застосунку, а для користувача ми використовуємо рядок \_\_table\_args\_\_ = (UniqueConstraint('name', 'user\_id', name='unique\_tag\_user')), який додає унікальне обмеження до таблиці тегів. UniqueConstraint вказує, що комбінація стовпців name та user\_id повинна бути унікальною. Це гарантує, що в користувача не може бути кількох тегів з однаковим ім'ям. Аргумент імені надає ім'я для обмеження - unique\_tag\_user, яке можна використовувати для посилання на обмеження SQLAlchemy.

Клас User має стовпці для імені користувача username, електронної пошти email, пароля password, часу створення created\_at, аватару avatar та токена оновлення refresh\_token.

Таблиця note\_m2m\_tag використовується для створення зв'язку багато до багатьох між нотатками і тегами.

**Додаємо схеми валідації**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/intro#%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-%D1%81%D1%85%D0%B5%D0%BC%D0%B8-%D0%B2%D0%B0%D0%BB%D1%96%D0%B4%D0%B0%D1%86%D1%96%D1%97)**file: src/schemas.py**

...

class UserModel(BaseModel):

username: str = Field(min\_length=5, max\_length=16)

email: str

password: str = Field(min\_length=6, max\_length=10)

class UserDb(BaseModel):

id: int

username: str

email: str

created\_at: datetime

avatar: str

class Config:

orm\_mode = True

class UserResponse(BaseModel):

user: UserDb

detail: str = "User successfully created"

class TokenModel(BaseModel):

access\_token: str

refresh\_token: str

token\_type: str = "bearer"

Ми додали у файл src/schemas.py чотири моделі

Pydantic: UserModel, UserDb, UserResponse та TokenModel.

Клас UserModel представляє корисні дані запиту для створення нового користувача. У ньому три поля: ім'я користувача, електронна пошта та пароль.

Поле імені користувача є рядком з мінімальною довжиною 5 і максимальною довжиною 16. Поле електронної пошти представляє рядок. Поле пароля представляє рядок з мінімальною довжиною 6 і максимальною довжиною 10.

Клас UserDb визначає представлення бази даних користувача. Він складається із шести полів, аналогічних полям бази даних. Клас Config вказує, що модель UserDb використовується для представлення моделі ORM.

Клас UserResponse - це модель відповіді, яка містить модель UserDb та поле відомостей detail з рядком "User successfully created".

Клас TokenModel визначає відповідь при отриманні токенів доступу для користувача, що пройшов аутентифікацію. Він має три поля: access\_token, refresh\_token та token\_type. Поля access\_token та refresh\_token є рядками. Поле token\_type є рядком зі значенням за замовчуванням bearer.

**Створюємо репозиторій користувача**

Для роботи з моделлю користувача User в SQLAlchemy ми визначили кілька функцій, які нам знадобляться надалі:

* get\_user\_by\_email ця функція приймає email та сеанс бази даних db та повертає об'єкт користувача з бази даних, якщо він існує з такою адресою електронної пошти.
* create\_user ця функція приймає параметр body, який вже пройшов валідацію моделлю користувача UserModel з тіла запиту, та другий параметр - сеанс бази даних db. Створює нового користувача у базі даних, а потім повертає щойно створений об'єкт User.
* update\_token ця функція приймає об'єкт користувача user, токен оновлення token та сеанс бази даних db. Вона оновлює поле refresh\_token користувача та фіксує зміни у базі даних.

**file: src/repository/users.py**

from libgravatar import Gravatar

from sqlalchemy.orm import Session

from src.database.models import User

from src.schemas import UserModel

async def get\_user\_by\_email(email: str, db: Session) -> User:

return db.query(User).filter(User.email == email).first()

async def create\_user(body: UserModel, db: Session) -> User:

avatar = None

try:

g = Gravatar(body.email)

avatar = g.get\_image()

except Exception as e:

print(e)

new\_user = User(\*\*body.dict(), avatar=avatar)

db.add(new\_user)

db.commit()

db.refresh(new\_user)

return new\_user

async def update\_token(user: User, token: str | None, db: Session) -> None:

user.refresh\_token = token

db.commit()

Тут досить все просто, з нового тільки об'єкт Gravatar, який створює функція create\_user на основі електронної пошти, і отримує URL-адресу аватара з Gravatar API. Після цього створює нового користувача в базі даних з наданими даними та аватаром.

[**Gravatar**](https://en.gravatar.com/)**(https://gravatar.com/)** (Globally Recognized Avatar) - це зручний та безкоштовний сервіс, що дозволяє створити аватар (електронний аватар), який може бути використаний на багатьох сайтах в інтернеті. Ви можете завантажити фото або зображення, яке буде прив'язане до вашої адреси електронної пошти. Потім, коли ви надсилаєте коментар або робите іншу діяльність, яка вимагає відображення вашого аватара, цей аватар відображатиметься на цьому сайті.

Щоб його використовувати, ми повинні додати до проекту пакет [libgravatar](https://pypi.org/project/libgravatar/) (https://pypi.org/project/libgravatar/)

﻿poetry add libgravatar

**Сервіс аутентифікації та авторизації**

Для правильної роботи нашого сервісу необхідно встановити наступні пакети:

poetry add python-jose["cryptography"]

poetry add passlib["bcrypt"]

poetry add python-multipart

Закріпимо, для чого потрібні ці пакети в нашому застосунку:

* python-jose["cryptography"] - це пакет, який надає функціональність для роботи з JSON Web Tokens (JWT) та допомагає створювати безпечні токени аутентифікації та авторизації для REST API
* passlib["bcrypt"] пакет необхідний для хешування паролів користувачів. Хешування паролів необхідно, щоб їх не можна було відновити у вихідний вигляд, навіть, якщо дані витечуть з бази даних.
* python-multipart - цей пакет для роботи з файлами у форматі multipart/form-data, який є основним форматом для завантаження файлів по HTTP, необхідний у цьому випадку для правильної роботи FastAPI.

Створимо модуль src/services/auth.py та всередині визначимо клас служби аутентифікації Auth. Вона має кілька методів для підтримки операцій аутентифікації та авторизації:

* verify\_password перевіряє, чи відповідає простий текстовий пароль хешованому паролю.
* get\_password\_hash хешує пароль за допомогою алгоритму bcrypt.
* create\_access\_token створює веб-токен JWT з областю дії scope, що дорівнює значенню access\_token, який буде використовуватись для авторизації користувача для доступу до захищених ресурсів.
* create\_refresh\_token створює JWT з областю дії refresh\_token, який можна використовувати для оновлення токена доступу access\_token після закінчення терміну його дії.
* decode\_refresh\_token метод декодує токен оновлення refresh\_token для отримання електронної пошти користувача.
* get\_current\_user авторизує користувача, розшифровуючи токен доступу access\_token та, перевіряючи, чи існує користувач у базі даних.

Потім ми створюємо екземпляр класу auth\_service = Auth(), який будемо використовувати у всьому коді для виконання операцій аутентифікації та авторизації.

**file: src/services/auth.py**

from typing import Optional

from jose import JWTError, jwt

from fastapi import HTTPException, status, Depends

from fastapi.security import OAuth2PasswordBearer

from passlib.context import CryptContext

from datetime import datetime, timedelta

from sqlalchemy.orm import Session

from src.database.db import get\_db

from src.repository import users as repository\_users

class Auth:

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

SECRET\_KEY = "secret\_key"

ALGORITHM = "HS256"

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/api/auth/login")

def verify\_password(self, plain\_password, hashed\_password):

return self.pwd\_context.verify(plain\_password, hashed\_password)

def get\_password\_hash(self, password: str):

return self.pwd\_context.hash(password)

# define a function to generate a new access token

async def create\_access\_token(self, data: dict, expires\_delta: Optional[float] = None):

to\_encode = data.copy()

if expires\_delta:

expire = datetime.utcnow() + timedelta(seconds=expires\_delta)

else:

expire = datetime.utcnow() + timedelta(minutes=15)

to\_encode.update({"iat": datetime.utcnow(), "exp": expire, "scope": "access\_token"})

encoded\_access\_token = jwt.encode(to\_encode, self.SECRET\_KEY, algorithm=self.ALGORITHM)

return encoded\_access\_token

# define a function to generate a new refresh token

async def create\_refresh\_token(self, data: dict, expires\_delta: Optional[float] = None):

to\_encode = data.copy()

if expires\_delta:

expire = datetime.utcnow() + timedelta(seconds=expires\_delta)

else:

expire = datetime.utcnow() + timedelta(days=7)

to\_encode.update({"iat": datetime.utcnow(), "exp": expire, "scope": "refresh\_token"})

encoded\_refresh\_token = jwt.encode(to\_encode, self.SECRET\_KEY, algorithm=self.ALGORITHM)

return encoded\_refresh\_token

async def decode\_refresh\_token(self, refresh\_token: str):

try:

payload = jwt.decode(refresh\_token, self.SECRET\_KEY, algorithms=[self.ALGORITHM])

if payload['scope'] == 'refresh\_token':

email = payload['sub']

return email

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail='Invalid scope for token')

except JWTError:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail='Could not validate credentials')

async def get\_current\_user(self, token: str = Depends(oauth2\_scheme), db: Session = Depends(get\_db)):

credentials\_exception = HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Could not validate credentials",

headers={"WWW-Authenticate": "Bearer"},

)

try:

# Decode JWT

payload = jwt.decode(token, self.SECRET\_KEY, algorithms=[self.ALGORITHM])

if payload['scope'] == 'access\_token':

email = payload["sub"]

if email is None:

raise credentials\_exception

else:

raise credentials\_exception

except JWTError as e:

raise credentials\_exception

user = await repository\_users.get\_user\_by\_email(email, db)

if user is None:

raise credentials\_exception

return user

auth\_service = Auth()

Багато чого вам вже повинно бути зрозуміло після прочитання попереднього розділу, але, все ж таки, для закріплення матеріалу, давайте розберемо код детальніше.

Атрибут oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/api/auth/login") - це екземпляр класу OAuth2PasswordBearer, який забезпечує авторизацію по bearer токену. Він потрібний для валідації JWT токена, який буде використовуватися як аутентифікаційні дані користувача. Зверніть увагу, ми вказуємо йому, де в нашому застосунку буде маршрут для аутентифікації tokenUrl="/api/auth/login". І пам'ятайте, що він, відповідно до стандарту, очікує на пару username і password, а ми, замість значення username, будемо підставляти в полі email користувача.

def verify\_password(self, plain\_password, hashed\_password):

return self.pwd\_context.verify(plain\_password, hashed\_password)

def get\_password\_hash(self, password: str):

return self.pwd\_context.hash(password)

Ці два методи слугують для верифікації пароля та хешування пароля відповідно. Метод verify\_password приймає на вхід пароль у відкритому вигляді plain\_password і зашифрований пароль hashed\_password, потім використовує метод verify з класу CryptContext для перевірки збігу відкритого пароля і зашифрованого. Якщо паролі збігаються, повертається True, інакше - False. Метод get\_password\_hash приймає на вхід пароль у відкритому вигляді password та повертає зашифрований пароль, згенерований за допомогою методу hash з класу CryptContext.

async def create\_access\_token(self, data: dict, expires\_delta: Optional[float] = None):

to\_encode = data.copy()

if expires\_delta:

expire = datetime.utcnow() + timedelta(seconds=expires\_delta)

else:

expire = datetime.utcnow() + timedelta(minutes=15)

to\_encode.update({"iat": datetime.utcnow(), "exp": expire, "scope": "access\_token"})

encoded\_access\_token = jwt.encode(to\_encode, self.SECRET\_KEY, algorithm=self.ALGORITHM)

return encoded\_access\_token

Метод create\_access\_token створює токен доступу access\_token і приймає два параметри:

* data - словник, що містить корисні дані для кодування у форматі JWT;
* expires\_delta - необов'язковий параметр, що визначає час життя токена в секундах. Якщо параметр не вказано, час життя за замовчуванням складає 15 хвилин.

Метод починається з копіювання корисних даних data у новий словник to\_encode. Потім обчислюємо час життя токена, і потім оновлюємо to\_encode наступною інформацією:

* iat поточний час у форматі UTC datetime;
* exp час життя токена
* scope рядок, що вказує область дії токена, в цьому випадку вона має значення "access\_token".

Потім метод повертає закодований JWT, використовуючи метод jwt.encode та атрибути класу SECRET\_KEY та ALGORITHM.

async def create\_refresh\_token(self, data: dict, expires\_delta: Optional[float] = None):

to\_encode = data.copy()

if expires\_delta:

expire = datetime.utcnow() + timedelta(seconds=expires\_delta)

else:

expire = datetime.utcnow() + timedelta(days=7)

to\_encode.update({"iat": datetime.utcnow(), "exp": expire, "scope": "refresh\_token"})

encoded\_refresh\_token = jwt.encode(to\_encode, self.SECRET\_KEY, algorithm=self.ALGORITHM)

return encoded\_refresh\_token

Метод create\_refresh\_token аналогічний методу create\_access\_token, але створює токен оновлення refresh\_token, який за замовчуванням має час життя 7 днів і область дії `scope'': "refresh\_token".

async def decode\_refresh\_token(self, refresh\_token: str):

try:

payload = jwt.decode(refresh\_token, self.SECRET\_KEY, algorithms=[self.ALGORITHM])

if payload['scope'] == 'refresh\_token':

email = payload['sub']

return email

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail='Invalid scope for token')

except JWTError:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail='Could not validate credentials')

Цей метод декодує токен оновлення refresh\_token та повертає з корисного навантаження email користувача. Якщо корисне навантаження токена не має області дії, що дорівнює "refresh\_token", воно викликає виняток HTTPException з кодом стану 401 та подробицями detail='Invalid scope for token'. Якщо JWT не може бути декодований, він викликає виняток HTTPException з кодом стану 401 та подробицями detail='Could not validate credentials'.

І нарешті останній метод get\_current\_user використовується для авторизації користувача на основі його токена доступу: access\_token. При цьому ми використовуємо клас OAuth2PasswordBearer для витягування токена із запиту, а потім декодуємо токен payload = jwt.decode(token, self.SECRET\_KEY, algorithms=[self.ALGORITHM]) з використанням атрибутів SECRET\_KETH класу Auth.

**Маршрути аутентифікації**

Після того як сервіс аутентифікації Auth створений, ми додаємо необхідні маршрути для роботи нашого застосунку. Створимо файл src/routes/auth.py та додамо роутер FastAPI router = APIRouter(prefix='/auth', tags=["auth"]) для функцій аутентифікації та авторизації. Нам необхідно визначити три маршрути:

* signup маршрут для реєстрації користувача;
* login маршрут для входу користувача;
* refresh\_token маршрут для оновлення токена доступу;

INFO

Не забудьте підключити новий роутер у головному файлі застосунку main.py

from fastapi import FastAPI

from src.routes import notes, tags, auth

app = FastAPI()

app.include\_router(auth.router, prefix='/api')

app.include\_router(tags.router, prefix='/api')

app.include\_router(notes.router, prefix='/api')

**file: src/routes/auth.py**

from typing import List

from fastapi import APIRouter, HTTPException, Depends, status, Security

from fastapi.security import OAuth2PasswordRequestForm, HTTPAuthorizationCredentials, HTTPBearer

from sqlalchemy.orm import Session

from src.database.db import get\_db

from src.schemas import UserModel, UserResponse, TokenModel

from src.repository import users as repository\_users

from src.services.auth import auth\_service

router = APIRouter(prefix='/auth', tags=["auth"])

security = HTTPBearer()

@router.post("/signup", response\_model=UserResponse, status\_code=status.HTTP\_201\_CREATED)

async def signup(body: UserModel, db: Session = Depends(get\_db)):

exist\_user = await repository\_users.get\_user\_by\_email(body.email, db)

if exist\_user:

raise HTTPException(status\_code=status.HTTP\_409\_CONFLICT, detail="Account already exists")

body.password = auth\_service.get\_password\_hash(body.password)

new\_user = await repository\_users.create\_user(body, db)

return {"user": new\_user, "detail": "User successfully created"}

@router.post("/login", response\_model=TokenModel)

async def login(body: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get\_db)):

user = await repository\_users.get\_user\_by\_email(body.username, db)

if user is None:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid email")

if not auth\_service.verify\_password(body.password, user.password):

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid password")

# Generate JWT

access\_token = await auth\_service.create\_access\_token(data={"sub": user.email})

refresh\_token = await auth\_service.create\_refresh\_token(data={"sub": user.email})

await repository\_users.update\_token(user, refresh\_token, db)

return {"access\_token": access\_token, "refresh\_token": refresh\_token, "token\_type": "bearer"}

@router.get('/refresh\_token', response\_model=TokenModel)

async def refresh\_token(credentials: HTTPAuthorizationCredentials = Security(security), db: Session = Depends(get\_db)):

token = credentials.credentials

email = await auth\_service.decode\_refresh\_token(token)

user = await repository\_users.get\_user\_by\_email(email, db)

if user.refresh\_token != token:

await repository\_users.update\_token(user, None, db)

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid refresh token")

access\_token = await auth\_service.create\_access\_token(data={"sub": email})

refresh\_token = await auth\_service.create\_refresh\_token(data={"sub": email})

await repository\_users.update\_token(user, refresh\_token, db)

return {"access\_token": access\_token, "refresh\_token": refresh\_token, "token\_type": "bearer"}

Функція signup для маршруту /signup обробляє операцію POST. Вона створює нового користувача, якщо користувача з такою електронною поштою не існує. Не може бути в системі два користувача з однаковим email. Якщо користувач з таким email вже існує в базі даних, функція викликає виняток HTTPException з кодом стану 409 Conflict та подробицями detail="Account already exists".

Функція login для маршруту /login обробляє операцію POST. Вона витягує користувача з бази даних з його email, якщо такого користувача немає, то викликається виняток HTTPException з кодом стану 401 та подробицями detail="Invalid email". Після цього виконується перевірка пароля на збіг, якщо паролі не ідентичні, то викликається виняток HTTPException з кодом стану 401 та подробицями detail="Invalid password". Після всіх перевірок генерується пара токенів access\_token та refresh\_token, для відправлення клієнту.

Також оновлюємо токен оновлення у базі даних для користувача.

Функція refresh\_token для маршруту /refresh\_token обробляє операцію GET. Вона декодує токен оновлення refresh\_token та витягує відповідного користувача з бази даних. Потім створює нові токени доступу та оновлення, і також оновлює refresh\_token в базі даних для користувача. Якщо токен оновлення недійсний, то викликається виняток HTTPException з кодом стану 401 та подробицями detail="Invalid refresh token".

**Додаємо авторизацію**

**Змінюємо репозиторії**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/auth#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D1%80%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D1%96%D1%97)Під час роботи з тегами необхідно враховувати, що тепер вони належать конкретному користувачеві. У кожну функцію нашого репозиторію ми додаємо новий параметр user: User з поточним користувачем. І тепер, під час запитів, за допомогою методу filter(Tag.user\_id == user.id) враховуємо належність тегу конкретному користувачеві.

**file: src/repository/tags.py**

from typing import List

from sqlalchemy.orm import Session

from sqlalchemy import and\_

from src.database.models import Tag, User

from src.schemas import TagModel

async def get\_tags(skip: int, limit: int, user: User, db: Session) -> List[Tag]:

return db.query(Tag).filter(Tag.user\_id == user.id).offset(skip).limit(limit).all()

async def get\_tag(tag\_id: int, user: User, db: Session) -> Tag:

return db.query(Tag).filter(and\_(Tag.id == tag\_id, Tag.user\_id == user.id)).first()

async def create\_tag(body: TagModel, user: User, db: Session) -> Tag:

tag = Tag(name=body.name, user\_id=user.id)

db.add(tag)

db.commit()

db.refresh(tag)

return tag

async def update\_tag(tag\_id: int, body: TagModel, user: User, db: Session) -> Tag | None:

tag = db.query(Tag).filter(and\_(Tag.id == tag\_id, Tag.user\_id == user.id)).first()

if tag:

tag .name = body.name

db.commit()

return tag

async def remove\_tag(tag\_id: int, user: User, db: Session) -> Tag | None:

tag = db.query(Tag).filter(and\_(Tag.id == tag\_id, Tag.user\_id == user.id)).first()

if tag:

db.delete(tag)

db.commit()

return tag

Ті самі операції виконуємо для репозиторію роботи з нотатками Note. У кожну функцію додаємо новий параметр user: User з поточним користувачем. І, під час запитів, за допомогою методу filter(Note.user\_id == user.id) враховуємо належність нотатки конкретному користувачеві.

**file: src/repository/notes.py**

from typing import List

from sqlalchemy import and\_

from sqlalchemy.orm import Session

from src.database.models import Note, Tag, User

from src.schemas import NoteModel, NoteUpdate, NoteStatusUpdate

async def get\_notes(skip: int, limit: int, user: User, db: Session) -> List[Note]:

return db.query(Note).filter(Note.user\_id == user.id).offset(skip).limit(limit).all()

async def get\_note(note\_id: int, user: User, db: Session) -> Note:

return db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

async def create\_note(body: NoteModel, user: User, db: Session) -> Note:

tags = db.query(Tag).filter(and\_(Tag.id.in\_(body.tags), Tag.user\_id == user.id)).all()

note = Note(title=body.title, description=body.description, tags=tags, user=user)

db.add(note)

db.commit()

db.refresh(note)

return note

async def remove\_note(note\_id: int, user: User, db: Session) -> Note | None:

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

db.delete(note)

db.commit()

return note

async def update\_note(note\_id: int, body: NoteUpdate, user: User, db: Session) -> Note | None:

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

tags = db.query(Tag).filter(and\_(Tag.id.in\_(body.tags), Note.user\_id == user.id)).all()

note.title = body.title

note.description = body.description

note.done = body.done

note.tags = tags

db.commit()

return note

async def update\_status\_note(note\_id: int, body: NoteStatusUpdate, user: User, db: Session) -> Note | None:

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

note.done = body.done

db.commit()

return note

**Змінюємо маршрути**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/auth#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D0%BC%D0%B0%D1%80%D1%88%D1%80%D1%83%D1%82%D0%B8)У маршрути для роботи з тегами та нотатками необхідно додати авторизацію за допомогою методу get\_current\_user класу Auth. Для кожного маршруту, де необхідна авторизація, потрібно додати параметр за допомогою залежності Depends(auth\_service.get\_current\_user). Параметр current\_user: User = Depends(get\_current\_user) отримує інформацію про поточного користувача з токена доступу access\_token, який ми повинні надати разом із запитом до маршруту.

**file: src/routes/tags.py**

...

@router.get("/", response\_model=List[TagResponse])

async def read\_tags(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

tags = await repository\_tags.get\_tags(skip, limit, current\_user, db)

return tags

@router.get("/{tag\_id}", response\_model=TagResponse)

async def read\_tag(tag\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

tag = await repository\_tags.get\_tag(tag\_id, current\_user, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

@router.post("/", response\_model=TagResponse, status\_code=status.HTTP\_201\_CREATED)

async def create\_tag(body: TagModel, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

return await repository\_tags.create\_tag(body, current\_user, db)

@router.put("/{tag\_id}", response\_model=TagResponse)

async def update\_tag(body: TagModel, tag\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

tag = await repository\_tags.update\_tag(tag\_id, body, current\_user, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

@router.delete("/{tag\_id}", response\_model=TagResponse)

async def remove\_tag(tag\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

tag = await repository\_tags.remove\_tag(tag\_id, current\_user, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

**file: src/routes/notes.py**

...

@router.get("/", response\_model=List[NoteResponse])

async def read\_notes(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

notes = await repository\_notes.get\_notes(skip, limit, current\_user, db)

return notes

@router.get("/{note\_id}", response\_model=NoteResponse)

async def read\_note(note\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

note = await repository\_notes.get\_note(note\_id, current\_user, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.post("/", response\_model=NoteResponse, status\_code=status.HTTP\_201\_CREATED)

async def create\_note(body: NoteModel, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

return await repository\_notes.create\_note(body, current\_user, db)

@router.put("/{note\_id}", response\_model=NoteResponse)

async def update\_note(body: NoteUpdate, note\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

note = await repository\_notes.update\_note(note\_id, body, current\_user, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.patch("/{note\_id}", response\_model=NoteResponse)

async def update\_status\_note(body: NoteStatusUpdate, note\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

note = await repository\_notes.update\_status\_note(note\_id, body, current\_user, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.delete("/{note\_id}", response\_model=NoteResponse)

async def remove\_note(note\_id: int, db: Session = Depends(get\_db),

current\_user: User = Depends(auth\_service.get\_current\_user)):

note = await repository\_notes.remove\_note(note\_id, current\_user, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

INFO

Зверніть увагу, що ми скрізь змінили, відповідно до REST, статус відповіді для операції POST (створення нового ресурсу) з 200 на 201. І в декораторах маршрутів додали для цього status\_code=status.HTTP\_201\_CREATED.

Після виконаних операцій ми отримаємо наступну Swagger документацію для нашого REST API застосунку:



**Підсумкова Swagger документація**

**Перевіряємо роботу застосунку**

Перевіримо базову реалізацію нашого REST API

**Реєструємо нового користувача**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/finish#%D1%80%D0%B5%D1%94%D1%81%D1%82%D1%80%D1%83%D1%94%D0%BC%D0%BE-%D0%BD%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0)Зареєструємо нового користувача з наступними вхідними даними:

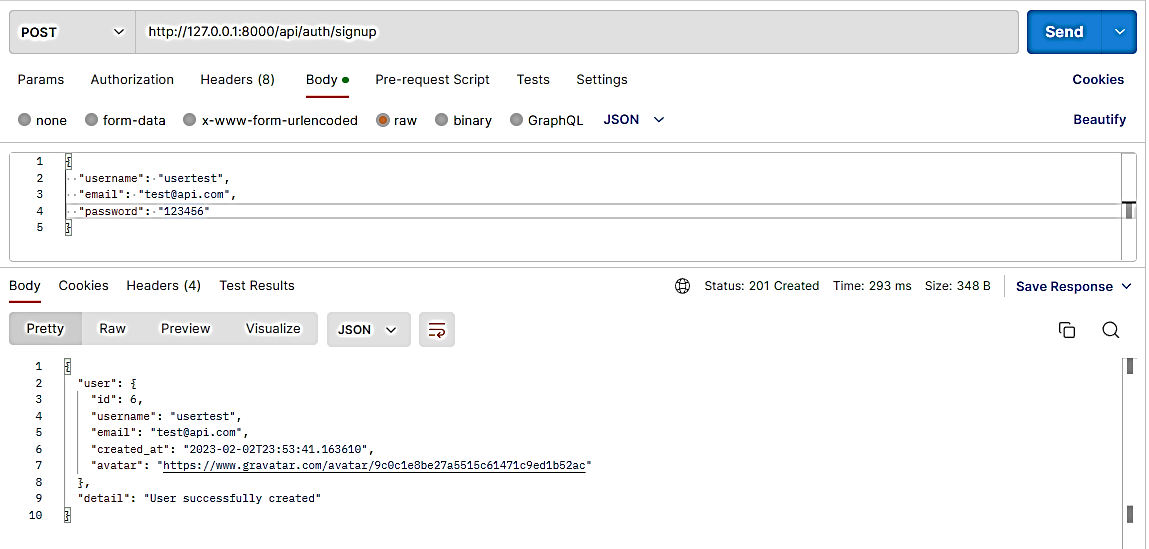
{

"username": "usertest",

"email": "test@api.com",

"password": "123456"

}



**Реєстрація користувача**

Відповідь сервера повинна бути наступного формату (приклад):

{

"user": {

"id": 6,

"username": "usertest",

"email": "test@api.com",

"created\_at": "2023-02-02T23:53:41.163610",

"avatar": "https://www.gravatar.com/avatar/9c0c1e8be27a5515c61471c9ed1b52ac"

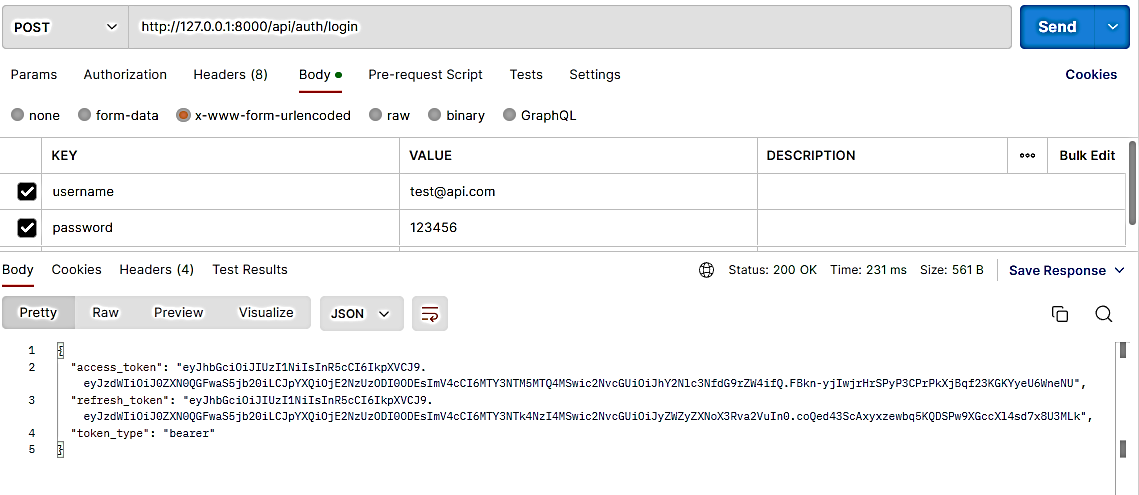
},

"detail": "User successfully created"

}

**Аутентифікація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/finish#%D0%B0%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F)Використовуючи режим імітації відправлення форми x-www-form-urlencoded, за допомогою Postman надсилаємо запит на сервер з полями: username і password. Нагадуємо, що замість username, підставляємо email користувача, оскільки він є унікальним для нашої бази даних.



**Аутентифікація користувача**

Отримуємо відповідь з необхідною для подальшої роботи парою токенів:

{

"access\_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0QGFwaS5jb20iLCJpYXQiOjE2NzUzODI0ODEsImV4cCI6MTY3NTM5MTQ4MSwic2NvcGUiOiJhY2Nlc3NfdG9rZW4ifQ.FBkn-yjIwjrHrSPyP3CPrPkXjBqf23KGKYyeU6WneNU",

"refresh\_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0QGFwaS5jb20iLCJpYXQiOjE2NzUzODI0ODEsImV4cCI6MTY3NTk4NzI4MSwic2NvcGUiOiJyZWZyZXNoX3Rva2VuIn0.coQed43ScAxyxzewbq5KQDSPw9XGccXl4sd7x8U3MLk",

"token\_type": "bearer"

}

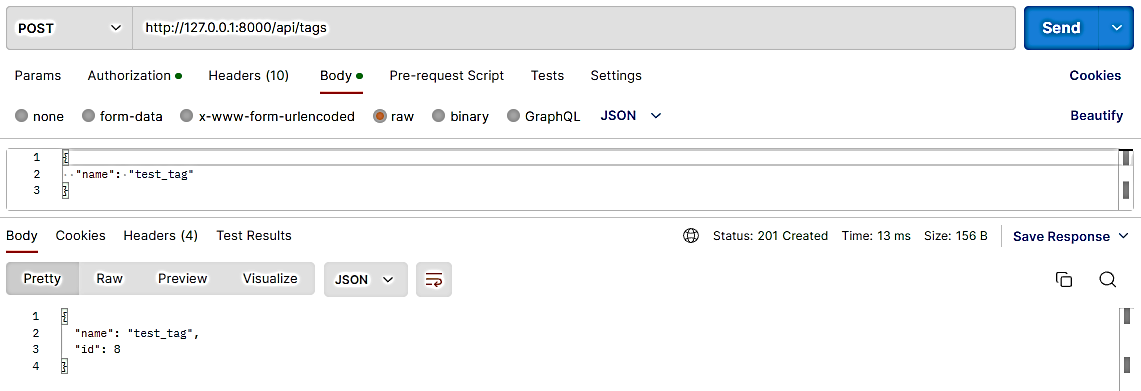
**Створюємо тег та нотатку**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/finish#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D1%82%D0%B5%D0%B3-%D1%82%D0%B0-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D1%83)Створимо тег з ім'ям "test\_tag", передавши в заголовку запиту необхідний access\_token.

{

"name": "test\_tag"

}



**Аутентифікація користувача**

Відповідь сервера:

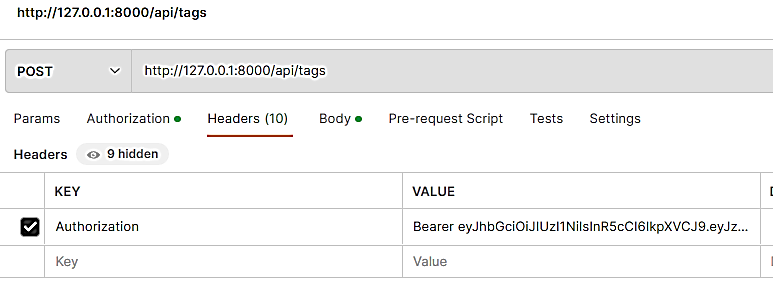
{

"name": "test\_tag",

"id": 8

}

Нагадаємо, що для передачі токена доступу access\_token під час запиту на сервер, його необхідно додати у вкладці Headers програми Postman як на скріншоті нижче:



**Аутентифікація користувача**

Далі, використовуючи id отриманого тегу, створюємо нотатку. Також, передавши в заголовку запиту необхідний access\_token.

{

"title": "Test note",

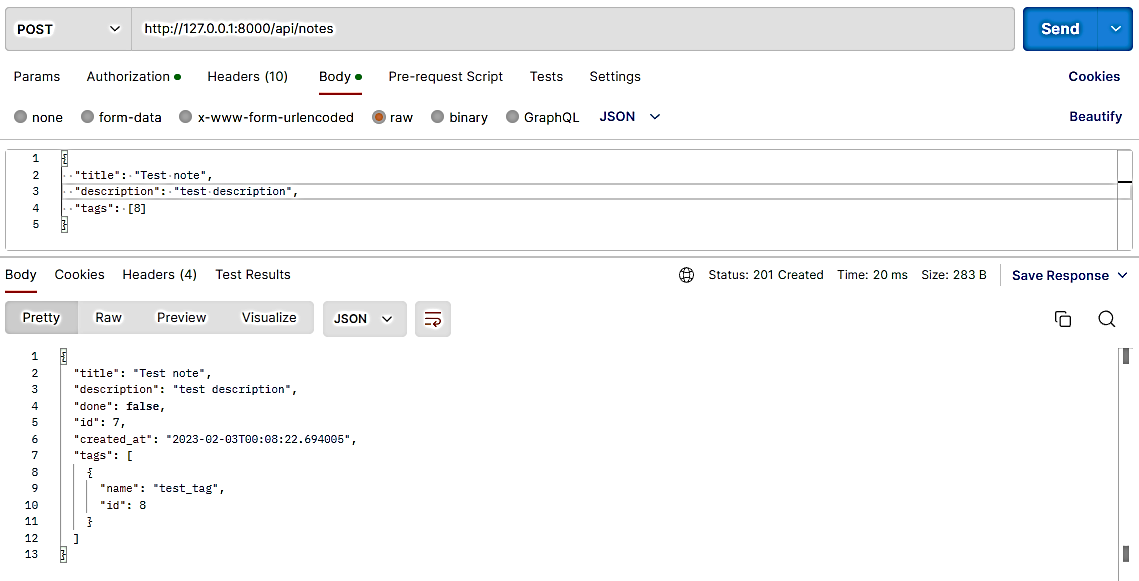
"description": "test description",

"tags": [

8

]

}



**Аутентифікація користувача**

Відповідь сервера:

{

"title": "Test note",

"description": "test description",

"done": false,

"id": 7,

"created\_at": "2023-02-03T00:08:22.694005",

"tags": [

{

"name": "test\_tag",

"id": 8

}

]

}

**Висновок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-02/finish#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)Як бачимо, застосунок працює коректно. Готовий результат повинен бути як у цьому живому прикладі нижче, запустіть та подивіться код. Перевірте самостійно за допомогою Postman працездатність інших маршрутів та операцій:

ПІДКАЗКА

Якщо будете використовувати Postman для роботи з REST API живого прикладу, маршрут реєстрації буде наступним https://fastapi-rest-api-auth.krabaton.repl.co/api/auth/signup. Вам потрібно буде перед кожним маршрутом додати доменне ім'я живого прикладу https://fastapi-rest-api-auth.krabaton.repl.co.

Щоб побачити Swagger документацію нашого застосунку, після запуску прикладу натисніть [посилання](https://fastapi-rest-api-auth.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-Rest-API-Auth)

Ви також можете [**завантажити повний приклад коду**](https://github.com/GoIT-Python-Web/REST-API-AUTH) (https://github.com/GoIT-Python-Web/REST-API-AUTH) розділу для запуску на локальній машині.

**INFO**

Для полегшення живого прикладу було здійснено заміну бази даних на SQLite. Це пов'язано з тим, що сервіс Replit підтримує PostgeSQL лише на платній основі. Також ми не використали міграції Alembic. Сам запуск сервера виконується прямо з коду:

if \_\_name\_\_ == "\_\_main\_\_":

uvicorn.run(app, host="0.0.0.0", port=8000)