**Модуль 12: Авторизація та Аутентифікація**

**Заняття 1: Авторизація та Аутентифікація**

**Вступ**

Спочатку розберемося з термінологією. Аутентифікація та авторизація, що це?

Ці два терміни багато хто використовує як взаємозамінні, проте це не зовсім так. Аутентифікація стосується перевірки автентичності користувача: що він той, за кого себе видає. Авторизація стосується визначення того, до чого користувач може отримати доступ у вашому вебзастосунку. Типовий приклад - звичайні користувачі та адміністратор, який теж авторизований, але має доступ не тільки до свого облікового запису, але й до записів інших користувачів. Логічно стає зрозуміло, що спочатку виконується аутентифікація, а потім встановлюється авторизація.

**Механізм сесій**

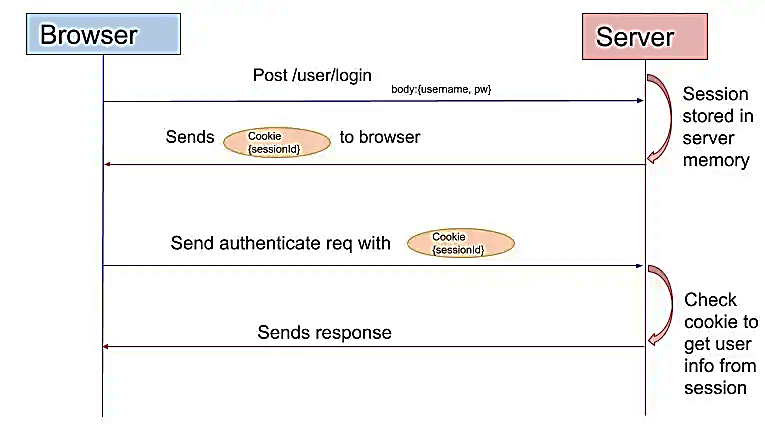
[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/intro#%D0%BC%D0%B5%D1%85%D0%B0%D0%BD%D1%96%D0%B7%D0%BC-%D1%81%D0%B5%D1%81%D1%96%D0%B9)HTTP протокол не зберігає стан запиту і, коли ви переходите сторінками сайту, то сервер не може знати, що це один і той самий користувач. І тут є певна проблема - неможливо залогінитися на сайт, зберегти налаштування сайту при переході між сторінками тощо.

Для таких речей існують [**cookie-файли**](https://en.wikipedia.org/wiki/HTTP_cookie) (<https://www.edu.goit.global/uk/learn/> 15249151/10926565/11865871/training?blockId=20111512) та сесії(сеанси), які дозволяють поверх протоколу HTTP зберігати стан користувача. Сенс cookie-файлу досить простий — сервер відправляє якийсь фрагмент інформації до 4 кб, а браузер зберігає його на стороні клієнта як файл. Існує певний проміжок часу зберігання цього файлу. Що стосується інформації, що зберігається, то часто - це просто якийсь унікальний ідентифікатор для користувача.

Механізм використання cookies для авторизації та створення сесій може бути описаний наступним чином:

1. Користувач вводить свої облікові дані (наприклад, логін та пароль) на сторінці аутентифікації та надсилає їх на сервер.
2. Сервер перевіряє облікові дані і, якщо вони вірні, створює унікальний ідентифікатор сесії (наприклад, випадковий рядок) та зберігає його у базі даних на сервері.
3. Сервер відправляє цей ідентифікатор сесії як cookie клієнту.
4. При кожному наступному запиті клієнта він відправляє цей ідентифікатор сесії як cookie на сервер.
5. Сервер використовує ідентифікатор сесії, щоб визначити конкретного користувача та аутентифікувати його.
6. Коли користувач виходить із системи або сесія закінчується, ідентифікатор сесії видаляється з бази даних і cookie з цим ідентифікатором видаляється з клієнта.

Використання механізму сесій для аутентифікації дозволяє зберігати стан користувача між запитами, що дозволяє користувачеві не вводити свої облікові дані при кожному запиті. Такий процес ми використовували у Django, коли створювали користувачів у попередньому розділі.



**Механізм сесії**

Але цей механізм має ряд недоліків:

1. Збереження стану: Традиційний механізм сесій вимагає збереження стану на сервері, що може бути складно при масштабуванні та розподілі навантажень між серверами.
2. Вразливість до XSS: Якщо програма вразлива до крос-сайтової скриптової атаки (XSS), то зловмисник може отримати доступ до сесійного ID та використовувати його для несанкціонованого доступу до облікового запису користувача.
3. Вразливість до CSRF: Якщо програма не захищена від атаки підробки запиту міжсайтової передачі (CSRF), то зловмисник може виконувати дії від імені аутентифікованого користувача без його відому. Згадайте в шаблонах Django нам доводиться в кожній формі вставляти {% csrf\_token %} для захисту від CSRF атак.
4. Обмежена передача даних: Сесійний ID зазвичай передається між клієнтом і сервером тільки як cookie, що може бути обмежуючим при використанні різних пристроїв або платформ.
5. Обмежена тривалість життя: Сесійні дані можуть бути видалені або закінчуватися залежно від конфігурації сервера, що може призводити до несподіваних втрат даних або переривання з'єднання.

Одним із недоліків використання cookies є те, що вони можуть бути легко підроблені або скопійовані, що може призвести до несанкціонованого доступу до акаунту користувача. Це можна уникнути за допомогою використання безпечних методів зберігання та передачі cookies, таких як зашифровані cookies або використання HttpOnly та Secure flag.

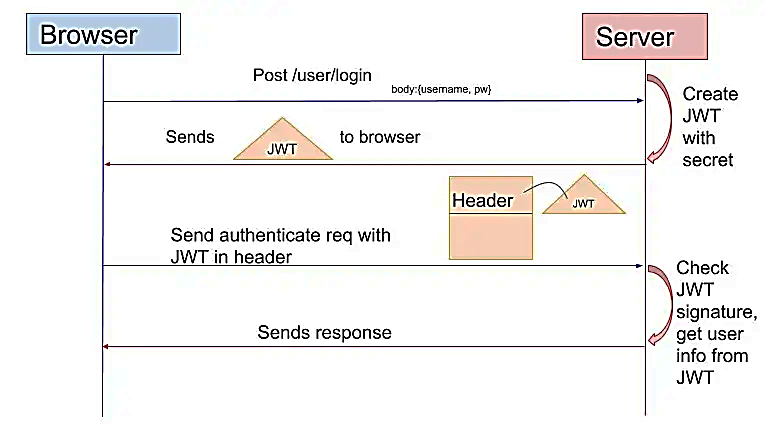
**JSON Web Token (JWT)**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/intro#json-web-token-jwt)

На зміну механізму сесій прийшов JWT.

INFO

[JSON Web Token (JWT)](https://jwt.io/introduction) (https://jwt.io/introduction)- це стандартний формат токена для аутентифікації та авторизації, який використовує JSON для зберігання інформації про користувача та підпису цієї інформації.

Багато веб-застосунків використовують веб-маркер JWT, замість сеансів для перевірки справжності. У застосунку на основі маркерів сервер створює JWT із секретом та відправляє JWT клієнту. Клієнт зберігає JWT (зазвичай у локальному сховищі) і включає JWT у заголовок з кожним запитом. Потім сервер перевіряє JWT з кожним запитом від клієнта та надсилає відповідь.



**Механізм JWT**

Найбільша різниця тут полягає у тому, що стан користувача не зберігається на сервері, оскільки стан зберігається всередині токена на стороні клієнта.

Більшість сучасних вебзастосунків використовують JWT для аутентифікації з багатьох причин, включаючи масштабованість та аутентифікацію мобільних пристроїв.

Ось кілька переваг при використанні JWT як альтернатива традиційним механізмам сесій:

1. Без збереження стану: JWT не вимагає збереження стану на сервері, що дозволяє легко розподіляти навантаження між серверами та полегшує масштабування застосунку.
2. Краща передача даних: JWT може легко передаватися між клієнтом та сервером у тілі запиту або в заголовку, а не тільки як cookie.
3. Безпечніший: JWT має сигнатуру, яка дозволяє перевірити його справжність та уникнути несанкціонованої зміни токена. Також JWT може бути зашифрований для додаткового захисту інформації.
4. Зручність використання: JWT має стандартизований формат, що дозволяє легко інтегрувати його в різні системи та платформи.

Хоча JWT має багато переваг, він також має свої недоліки, такі як неможливість відкликати токен та обмежена можливість оновлення інформації в токені без його повної заміни. Тому використання JWT повинно бути усвідомленим та поєднаним з іншими методами безпеки для максимального захисту застосунку.

**Що таке веб-токени JSON?**

**Опис JWT**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt#%D0%BE%D0%BF%D0%B8%D1%81-jwt)**JSON Web Token (JWT)** у своїй найпростішій формі є безпечним URL-рядком, що містить закодований об'єкт JSON. JWT - це відкритий промисловий стандарт, повністю описаний в [RFC 7519](https://www.rfc-editor.org/rfc/rfc7519) (https://www.rfc-editor.org/rfc/rfc7519), який містить велику кількість деталей, зокрема про те, як JWT вимагає функцію для забезпечення безпеки створеного маркера.

Давайте подивимося на приклад точена із сайту <https://jwt.io/> (https://jwt.io/):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

Зверніть увагу, що токен містить три частини, які розділені крапкою ".": header.payload.signature.

Ці три частини наступні:

* **header** — містить інформацію, яка визначає алгоритм хешування, щоб його можна було використовувати для правильного розшифрування та аутентифікації.

{

"alg": "HS256",

"typ": "JWT"

}

* **payload**— ця частина містить інформацію, яку ви хочете надіслати за допомогою JWT. Зверніть увагу, що payload не захищений і може бути розшифрований без секретного ключа – це звичайне кодування Base64. JWT не призначений для надсилання конфіденційної інформації, такої як паролі або іншої особистої інформації.

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022

}

* **signature** — поєднує закодований header та payload із секретним ключем і надійно кодує це з використанням алгоритму хешування, визначеного в header.

Таким чином, щоразу, коли створюється токен, header залишиться постійним (якщо ви не зміните алгоритм хешування); payload залишиться постійним, якщо, знову ж таки, ми не змінимо його зміст, а ми змінюватимемо його, вказуючи тривалість життя токена; signature шифруватиме ці дві частини інформації на основі алгоритму хешування за допомогою секретного ключа.

**Бібліотека python-jose**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt#%D0%B1%D1%96%D0%B1%D0%BB%D1%96%D0%BE%D1%82%D0%B5%D0%BA%D0%B0-python-jose)python-jose - це бібліотека для роботи з JSON Web Tokens (JWT) на Python. Вона дозволяє створювати, підписувати, перевіряти та дешифрувати JWT.

Встановлення бібліотеки:

pip install python-jose[cryptography]

Створення та перевірка токена:

from jose import jwt

# дані для заповнення токена

payload = {"sub": "1234567890", "name": "John Doe"}

# створення токена з симетричним ключем

encoded = jwt.encode(payload, "secret\_key", algorithm='HS256')

print(encoded)

# перевірка токена

decoded = jwt.decode(encoded, "secret\_key", algorithms=['HS256'])

print(decoded)

Виведення:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIn0.mat7\_4Ab301xPP14vVC0WAlI\_KYPh4rX2vkDnLwrxJM

{'sub': '1234567890', 'name': 'John Doe'}

Якщо у метода decode буде неправильний секретний ключ, то виникне помилка JWSError: "Signature verification failed".

**Базовий приклад JWT авторизації**

JWT (JSON Web Token) авторизація використовує механізм обміну токенами для аутентифікації користувача. Розглянемо загальний алгоритм роботи JWT авторизації:

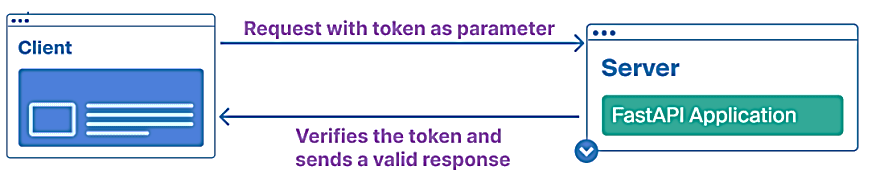
1. Клієнт відправляє запит на отримання токена, передаючи свої облікові дані (ім'я користувача username і пароль password) на сервер.
2. Сервер перевіряє облікові дані і, якщо вони вірні, створює JWT токен та відправляє його клієнту.



**Отримання JWT токена**

3.Клієнт зберігає JWT токен і в подальшому використовує його для аутентифікації при зверненні до ресурсів сервера. Щоразу клієнт відправляє JWT токен на сервер у заголовку запиту.

4.Сервер перевіряє справжність JWT токена, використовуючи підпис і дані в токені. Якщо токен валідний, то сервер обробляє запит та повертає ресурс клієнту. Якщо токен недійсний або минув термін його дії, то сервер повертає помилку.



**Запит до API за допомогою JWT токена**

**Реалізація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F)Для цього прикладу потрібний наступний набір пакетів:

python = "^3.10"

fastapi = "^0.89.1"

uvicorn = "^0.20.0"

fastapi-jwt-auth = "^0.5.0"

python-jose = {extras = ["cryptography"], version = "^3.3.0"}

sqlalchemy = "^1.4.46"

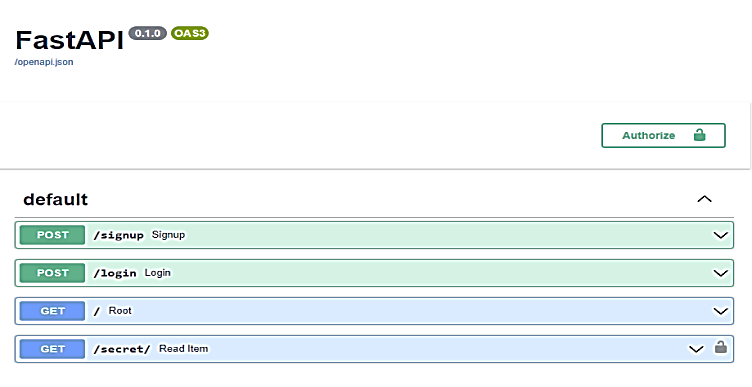
passlib = {extras = ["bcrypt"], version = "^1.7.4"}

python-multipart = "^0.0.5"

Сам приклад ми розіб'ємо на три файли:

* app.py файл застосунку з маршрутами;
* db.py підключення до бази даних та моделі;
* auth.py методи роботи з аутентифікацією та авторизацією;

І реалізуємо маршрути як на малюнку знизу:



**Swagger документація прикладу**

У нас буде маршрут для реєстрації у нашому застосунку /signup. Також реалізуємо маршрут для аутентифікації /login. Цей маршрут повинен буде приймати email та пароль, та повинен повертати нам JWT токен: access\_token. На маршрут / ми можемо потрапити в будь-якому випадку, а от маршрут /secret - тільки використовуючи access\_token.

**Підключення до бази даних**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%D0%BF%D1%96%D0%B4%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D0%B1%D0%B0%D0%B7%D0%B8-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

**db.py**

from sqlalchemy import create\_engine, Column, String, Integer

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./sql\_app.db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}

)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

class User(Base):

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

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

email = Column(String(150), nullable=False, unique=True)

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

Base.metadata.create\_all(bind=engine)

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

Цей код створює базу даних SQLite у файлі sql\_app.db за допомогою SQLAlchemy. Функція create\_engine створює двигун для бази даних SQLite, вказаної в константі SQLALCHEMY\_DATABASE\_URL. Змінна SessionLocal створюється як фабрика сесій, яка використовується для створення нових сесій.

Функція declarative\_base створює базовий клас Base для оголошених моделей, який використовується для визначення моделі User. Модель User має три стовпці: id, email та password. Атрибут \_\_tablename\_\_ встановлений у "users" і використовується як ім'я таблиці в базі даних.

Функція Base.metadata.create\_all створює таблицю в базі даних із використанням моделі User. Для прикладу немає необхідності використовувати механізми міграції.

Функція get\_db є залежністю, що дозволяє коду використовувати сесію бази даних. Функція створює нову сесію під час її виклику та повертає сесію, потім закриває сесію після закінчення роботи.

З нового тут напевно лише аргумент connect\_args у функції create\_engine. Він використовується для зазначення аргументу з'єднання, який говорить SQLite відключити перевірку на той самий потік.

SQLite - легковаговий двигун бази даних, призначений для вбудовування в інші застосунки. Одним з обмежень SQLite є те, що за замовчуванням він дозволяє лише одне з'єднання з базою даних з одного потоку. Тому, якщо ви спробуєте відкрити з'єднання з базою даних SQLite з двох різних потоків, одне зі з'єднань не буде вдалим.

Параметр check\_same\_thread використовується для відключення цього обмеження, що дозволяє відкривати кілька з'єднань з різних потоків до однієї бази даних. Передаючи {"check\_same\_thread": False} як connect\_args, застосунок може відкривати кілька з'єднань з однією і тією самою базою даних SQLite з різних потоків без помилок.

**Аутентифікація та створення токенів**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%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-%D1%82%D0%B0-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%BE%D0%BA%D0%B5%D0%BD%D1%96%D0%B2)

**auth.py**

from datetime import datetime, timedelta

from typing import Optional

from fastapi import Depends, HTTPException

from passlib.context import CryptContext

from fastapi.security import OAuth2PasswordBearer

from sqlalchemy.orm import Session

from jose import JWTError, jwt

from starlette import status

from db import get\_db, User

class Hash:

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

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)

SECRET\_KEY = "secret\_key"

ALGORITHM = "HS256"

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

# define a function to generate a new access token

async def create\_access\_token(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({"exp": expire})

encoded\_jwt = jwt.encode(to\_encode, SECRET\_KEY, algorithm=ALGORITHM)

return encoded\_jwt

async def get\_current\_user(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, SECRET\_KEY, algorithms=[ALGORITHM])

email = payload["sub"]

if email is None:

raise credentials\_exception

except JWTError as e:

raise credentials\_exception

user: User = db.query(User).filter(User.email == email).first()

if user is None:

raise credentials\_exception

return user

Цей код визначає кілька функцій та класів для обробки аутентифікації користувача та генерації токенів доступу з використанням JWT та класу OAuth2PasswordBearer з бібліотеки FastAPI.

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

OAuth2PasswordBearer - це клас, який надається бібліотекою FastAPI, який дозволяє легко реалізувати аутентифікацію на основі механізму [**OAuth2**](https://oauth.net/2/) (https://oauth.net/2/). Цей потік аутентифікації часто використовується для входу користувачів з адресою електронної пошти та паролем. Коли ви визначаєте клас OAuth2PasswordBearer, вам потрібно передати URL точки входу, яка оброблятиме видачу токена, зазвичай, у документації /token, але ми перейменували його на адекватніший /login. Це той маршрут, де наш застосунок буде отримувати електронну пошту та пароль користувача та видавати токен доступу клієнту. OAuth2PasswordBearer автоматично обробляє процес парсингу токена із заголовка Authorization запиту та передачі його у функцію обробки маршруту, який захищений цим класом. Якщо токен недійсний або його термін дії минув, клас поверне HTTPException зі status\_code рівним 401. Ми, наприклад, використовуємо декоратор Depends(oauth2\_scheme) у функції get\_current\_user, щоб передати токен у функцію і перевірити, чи є токен дійсним.

INFO

Коротко, клас OAuth2PasswordBearer використовується для захисту маршрутів вашого застосунку перевіркою дійсності токена, переданого в заголовку Authorization запиту.

Клас Hash надає методи для хешування та перевірки паролів з використанням алгоритму bcrypt. Алгоритм bcrypt є функцією хешування паролів, яка вважається безпечною та ефективною.

class Hash:

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

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)

Атрибут pwd\_context класу є екземпляром класу CryptContext з бібліотеки passlib. Цей клас надає зручний спосіб роботи з кількома схемами хешування паролів, у тому числі bcrypt. Метод verify\_password класу приймає як аргументи звичайний пароль plain\_password та хешований пароль hashed\_password, і повертає булеве значення, що вказує на те, чи збігається звичайний пароль з хешованим паролем. Метод використовує метод verify об'єкта pwd\_context для порівняння звичайного пароля та хешованого пароля. Метод get\_password\_hash класу приймає як аргумент звичайний пароль та повертає хешовану версію пароля з використанням методу hash об'єкта pwd\_context

Цей клас надає безпечний спосіб зберігання паролів з використанням алгоритму bcrypt. Алгоритм bcrypt розроблений таким чином, щоб бути повільним та високо витратним в обчисленнях, це робить його стійкішим до атак перебором. Коли пароль користувача хешується, він може бути збережений в базі даних і оригінальний пароль не може бути відновлений з хешованого пароля. Це ускладнює зловмиснику, якщо він отримав доступ до бази даних, вгадати оригінальний пароль, пробуючи різні вхідні дані.

async def create\_access\_token(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({"exp": expire})

encoded\_jwt = jwt.encode(to\_encode, SECRET\_KEY, algorithm=ALGORITHM)

return encoded\_jwt

Функція create\_access\_token генерує новий access\_token з кінцевим часом життя, кодуючи словник даних, так званий to\_encode в JWT. Функція спочатку робить копію словника даних, щоб вихідні дані не змінювалися. Потім вона перевіряє, чи надано аргумент expires\_delta, якщо так, то розраховує час закінчення терміну дії, додаючи кількість секунд, наданих в expires\_delta, до поточного часу UTC. Якщо expires\_delta не надано, то встановлює час закінчення терміну дії на 15 хвилин від поточного часу UTC. Функція потім додає час закінчення терміну дії до словника to\_encode з використанням методу update. Врешті, вона використовує функцію jwt.encode для кодування даних у JWT, використовуючи раніше визначений SECRET\_KEY та ALGORITHM, і повертає закодований access\_token.

Надалі цей токен буде використаний як токен доступу для авторизації.

async def get\_current\_user(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, SECRET\_KEY, algorithms=[ALGORITHM])

email = payload["sub"]

if email is None:

raise credentials\_exception

except JWTError as e:

raise credentials\_exception

user: User = db.query(User).filter(User.email == email).first()

if user is None:

raise credentials\_exception

return user

Функція get\_current\_user використовується для аутентифікації користувача на основі його токена доступу: access\_token. Функція використовує клас OAuth2PasswordBearer для витягування токена із запиту, потім декодує JWT з використанням SECRET\_KEY та алгоритму HS256. Після цього функція витягує адресу електронної пошти користувача з навантаження JWT та використовує її для запиту інформації про користувача з бази даних. Якщо користувач не знайдений або токен недійсний, функція викликає виключення HTTP з кодом статусу 401. Функція get\_db є залежністю, що дозволяє коду використовувати сесію бази даних. Функція get\_db створює нову сесію під час її виклику та повертає сесію, а потім закриває сесію після закінчення роботи.

Після того, як ми детально розібрали код цього модуля, переходимо до основного застосунку.

**Основний застосунок**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D0%B8%D0%B9-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA)

**app.py**

from fastapi import FastAPI, Depends, HTTPException, status

from fastapi.security import OAuth2PasswordRequestForm

from sqlalchemy.orm import Session

from pydantic import BaseModel

from auth import create\_access\_token, get\_current\_user, Hash

from db import User, get\_db

app = FastAPI()

hash\_handler = Hash()

class UserModel(BaseModel):

username: str

password: str

@app.post("/signup")

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

exist\_user = db.query(User).filter(User.email == body.username).first()

if exist\_user:

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

new\_user = User(email=body.username, password=hash\_handler.get\_password\_hash(body.password))

db.add(new\_user)

db.commit()

db.refresh(new\_user)

return {"new\_user": new\_user.email}

@app.post("/login")

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

user = db.query(User).filter(User.email == body.username).first()

if user is None:

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

if not hash\_handler.verify\_password(body.password, user.password):

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

# Generate JWT

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

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

@app.get("/")

async def root():

return {"message": "Hello World"}

@app.get("/secret")

async def read\_item(current\_user: User = Depends(get\_current\_user)):

return {"message": 'secret router', "owner": current\_user.email}

Це простий вебзастосунок з функціями аутентифікації та авторизації. Як ми вже говорили, у ньому визначено кілька маршрутів:

* /signup - для реєстрації нового користувача
* /login - для аутентифікації користувача
* / - для тестування застосунку
* /secret - захищений маршрут, який вимагає авторизації

**Реєстрація користувача**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%D1%80%D0%B5%D1%94%D1%81%D1%82%D1%80%D0%B0%D1%86%D1%96%D1%8F-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0)Маршрут /signup приймає дані нового користувача та додає його в базу даних. Маршрут /login приймає ім'я користувача та пароль, та повертає токен JWT для авторизації користувача. Маршрут /secret вимагає авторизації та повертає захищену інформацію.

Функція signup відповідає за дію маршруту для реєстрації нового користувача у застосунку. Коли користувач надсилає запит POST на цей маршрут, у тілі запиту повинні бути представлені дані нового користувача у форматі JSON, і вони будуть перетворені на об'єкт UserModel.

class UserModel(BaseModel):

username: str

password: str

Стандарт **OAuth2** вимагає, щоб при використанні "потоку паролів" (який ми використовуємо) клієнт/користувач відправив поля у вигляді даних форми: username та password. У нашому застосунку ми збираємося використовувати email та password. Тут не варто хвилюватися, оскільки кінцевим користувачам на клієнті ми можемо завжди показати, що чекаємо від них поля email та password. А наша модель баз даних вже використовує поле email, замість username. Але для операції шляху входу нам потрібно використовувати ці імена username та password, щоб бути сумісними зі специфікацією та мати можливість використовувати **Swagger** документацію та клас OAuth2PasswordBearer для витягування токена із запиту.

Функція перевіряє, чи існує користувач з таким самим ім'ям у базі даних. Якщо так, то функція повертає помилку HTTP 409 Conflict, повідомляючи про те, що обліковий запис вже існує. Якщо ні, то створюється новий об'єкт User з email користувача та хешованим паролем.

new\_user = User(email=body.username, password=hash\_handler.get\_password\_hash(body.password))

Цей об'єкт додається в базу даних і повертається відповідь з інформацією про нового користувача.

**Аутентифікація**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%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)

Функція login діє як обробник маршруту для аутентифікації користувача у застосунку. Коли користувач надсилає запит POST на цей маршрут /login зі своїм ім'ям користувача (у нашому випадку нагадуємо email) та паролем у тілі запиту, вони перетворюються на об'єкт OAuth2PasswordRequestForm.

INFO

OAuth2PasswordRequestForm - це клас фреймворку FastAPI для обробки запитів з ім'ям користувача та паролем у форматі OAuth 2.0. Він містить властивості username та password відповідно до формату OAuth 2.0. Ми використовуємо цей клас для обробки запиту з ім'ям користувача та паролем для валідації.

Функція шукає користувача у базі даних із зазначеним ім'ям користувача. Якщо користувача не знайдено, функція повертає помилку HTTP 401 Unauthorized, повідомляючи про недійсне ім'я користувача. Якщо користувач знайдений, то функція перевіряє пароль користувача за допомогою класу Hash, якщо пароль неправильний, то функція повертає помилку HTTP 401 Unauthorized, повідомляючи про недійсний пароль. Якщо ім'я користувача та пароль валідні, то функція генерує новий токен доступу за допомогою функції create\_access\_token і повертає цей токен як частину відповіді. Цей токен може бути використаний для ідентифікації користувача для подальших запитів до захищених маршрутів.

**Авторизація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/basic#%D0%B0%D0%B2%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D1%96%D1%8F)Останній наш маршрут вимагає авторизації, щоб повернути "секретну" інформацію:

@app.get("/secret")

async def read\_item(current\_user: User = Depends(get\_current\_user)):

return {"message": 'secret router', "owner": current\_user.email}

Тут current\_user: User = Depends(get\_current\_user) - це аргумент функції, який отримує інформацію про поточного користувача з токена доступу access\_token, який ми повинні надати разом із запитом. Отримуємо ми access\_token під час роботи з маршрутом /login. Сам процес отримання інформації досягається за допомогою залежності Depends(get\_current\_user), яка викликає безпосередньо функцію get\_current\_user для отримання інформації про користувача з токена.

Запустимо наш застосунок і створимо тестового користувача username=test@test.com, password=123456 за маршрутом /signup

**Створення нового користувача**

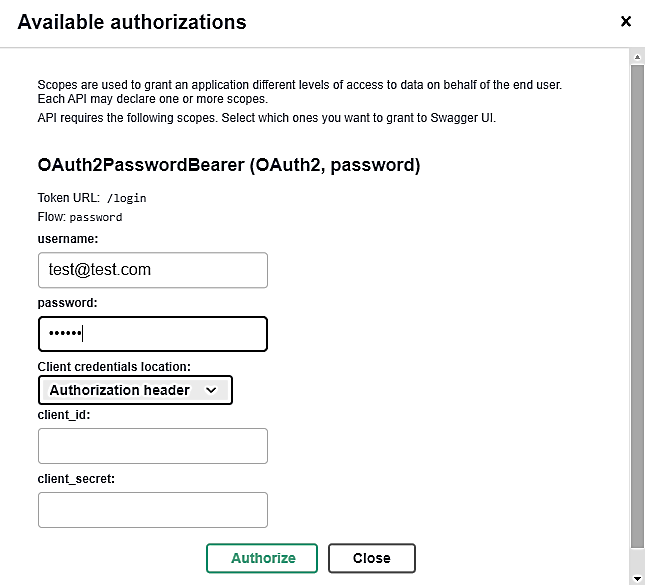
Документація повинна повернути відповідь:

{

"new\_user": "test@test.com"

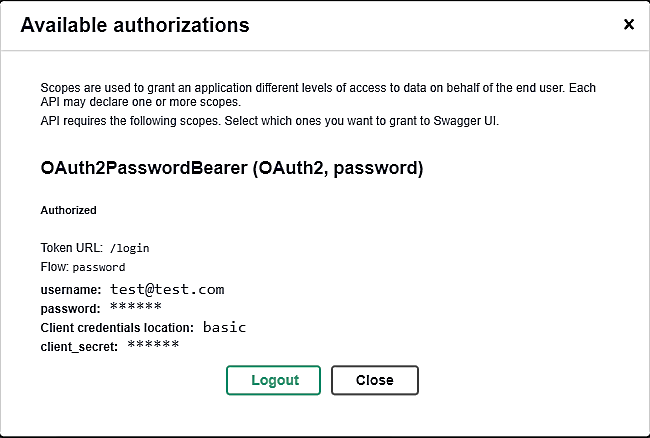
}

Тепер у базі даних у нас є користувач і ми можемо виконати процес аутентифікації за маршрутом /login або ж у верхньому кутку нашої документації вибрати та натиснути кнопку Authorize. З'явиться вікно з формою для аутентифікації. Введемо в ньому дані тільки-но створеного користувача.



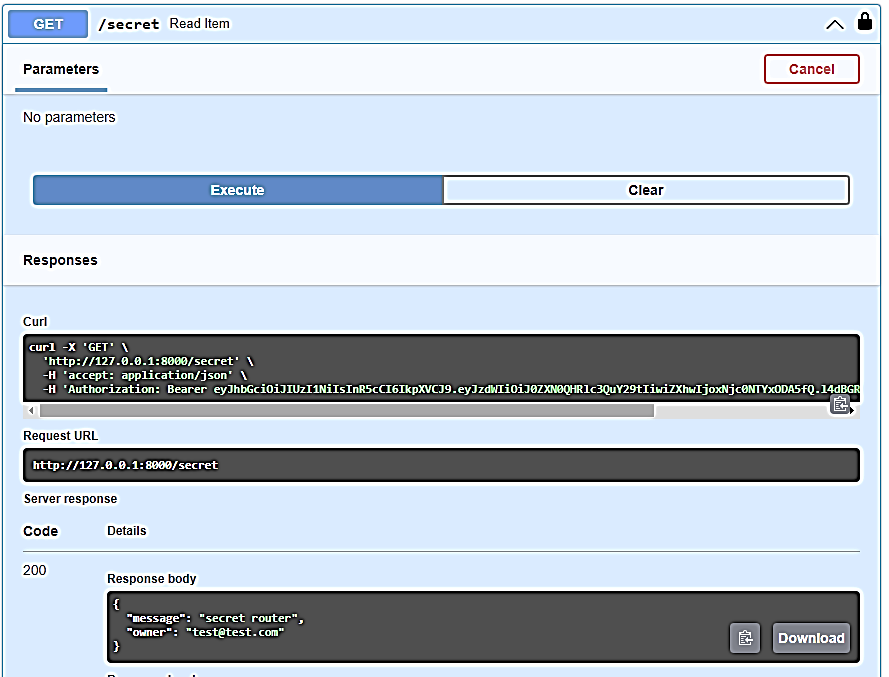
**Введення email і пароля**

Якщо при введенні ми не помилилися, то Swagger документація аутентифікує нас і збереже отриманий access\_token для подальшої роботи.



**Успішна аутентифікація**

Тепер ми готові отримати відповідь від секретного маршруту. Зверніть увагу, що значок замочка праворуч змінив колір на чорний, це означає, що ми вже маємо до нього доступ:



**Успішна авторизація**

Готовий результат повинен бути як у цьому живому прикладі, запустіть та подивіться код:

Щоб побачити Swagger документацію нашого застосунку, після запуску прикладу натисніть [посилання](https://fastapi-jwt.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-Jwt#main.py)

Ми завершили базовий приклад, настав час розібрати складніший варіант із двома токенами.

**Використання пари JWT токенів**

У попередньому прикладі ми використали один токен, але це не зовсім безпечно. Якщо термін JWT токена великий, то після крадіжки токена, зловмисник міг би досить довго продовжувати використовувати його для отримання доступу до ресурсів.

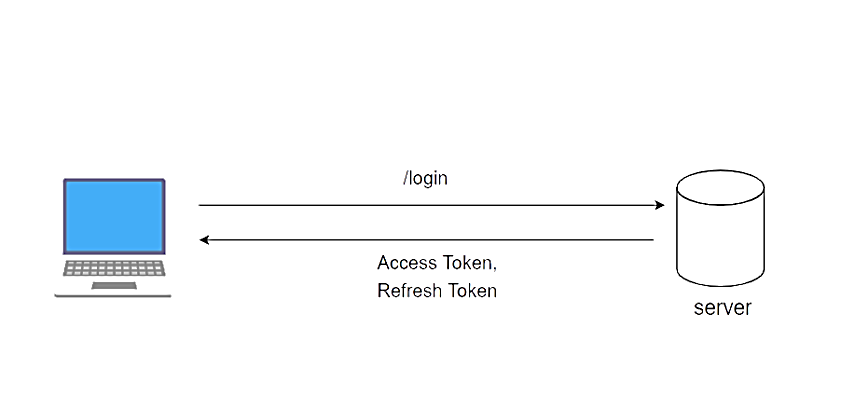
Тому використовують пару токенів: access token та refresh token.

* access token - використовується для авторизації запитів та зберігання додаткової інформації про користувача.
* refresh token - видається сервером за результатами успішної аутентифікації та використовується для отримання нового access/refresh токена.

Таким чином refresh token використовується для отримання нового токена доступу access token після закінчення терміну дії старого токена. Це необхідно для того, щоб забезпечити безпеку застосунку та захист від крадіжки токена.

Схема роботи авторизації буде такою:

1. Клієнт відправляє запит на авторизацію з логіном та паролем на сервер.
2. Сервер перевіряє дані та, у разі успіху, генерує пару JWT токенів і відправляє їх клієнту.

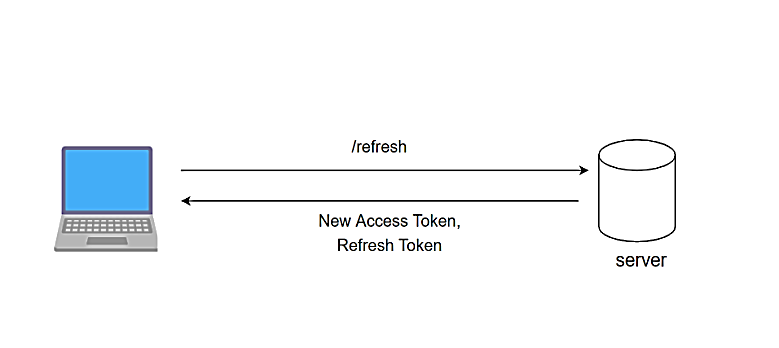


**Отримання пари 'access token' і 'refresh token'**

3.Клієнт використовує access token для авторизації та надсилання запитів на сервер.

4.Коли час життя access token закінчується, клієнт надсилає запит на оновлення з використанням refresh token.

5.Сервер перевіряє refresh token та, у разі успіху, генерує новий доступний access token і відправляє його клієнту.



**Запит нового 'access token' за допомогою 'refresh token'**

В алгоритмі авторизації з використанням пари JWT токенів, клієнт ініціює запит на аутентифікацію з наданням логіну та паролю. Якщо логін і пароль вірні, сервер повертає два токена: access token та refresh token.

**Access token** використовується для доступу до захищених ресурсів, оскільки він має обмежений термін життя та буде завершуватися. Коли термін життя токена завершується, клієнт повинен запросити новий access token, використовуючи свій refresh token.

**Refresh token**використовується для отримання нового access token без необхідності повторної аутентифікації. Він має більший термін життя і повинен зберігатися в безпеці.

**Реалізація**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt_refresh#%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F)Структура файлів та склад пакетів залишається тим самим. Зосередимося на відмінностях цієї реалізації.

**Підключення до бази даних**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt_refresh#%D0%BF%D1%96%D0%B4%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D0%B1%D0%B0%D0%B7%D0%B8-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

**db.py**

from sqlalchemy import create\_engine, Column, String, Integer

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./app.db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}

)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

class User(Base):

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

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

email = Column(String(150), nullable=False, unique=True)

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

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

Base.metadata.create\_all(bind=engine)

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

У моделі User у нас з'явилося поле refresh\_token для зберігання рефреш токена. Зберігання рефреш токенів у базі даних необхідно для того, щоб система могла перевіряти їх достовірність та дійсність. Коли клієнт запитує новий доступний токен за допомогою рефреш токена, система може перевірити його наявність та валідність у базі даних, перш ніж видати новий токен. Це також дозволяє відкликати рефреш токени та заборонити їх використання у разі компрометації або інших проблем, пов'язаних з безпекою.

INFO

Варто зауважити, що для використання можливості аутентифікації на більш ніж одному девайсі, необхідно зберігати всі рефреш токени для кожного клієнта. А це означає, що для реалізації нам потрібна окрема таблиця зберігання рефреш токенів для кожного пристрою користувача. У нашій реалізації ми вважаємо, що у користувача лише один пристрій.

**Аутентифікація та створення токенів**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt_refresh#%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-%D1%82%D0%B0-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%BE%D0%BA%D0%B5%D0%BD%D1%96%D0%B2)

**auth.py**

from datetime import datetime, timedelta

from typing import Optional

from fastapi import Depends, HTTPException

from passlib.context import CryptContext

from fastapi.security import OAuth2PasswordBearer

from sqlalchemy.orm import Session

from jose import JWTError, jwt

from starlette import status

from db import get\_db, User

class Hash:

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

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)

SECRET\_KEY = "secret\_key"

ALGORITHM = "HS256"

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

# define a function to generate a new access token

async def create\_access\_token(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, SECRET\_KEY, algorithm=ALGORITHM)

return encoded\_access\_token

# define a function to generate a new refresh token

async def create\_refresh\_token(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, SECRET\_KEY, algorithm=ALGORITHM)

return encoded\_refresh\_token

async def get\_email\_form\_refresh\_token(refresh\_token: str):

try:

payload = jwt.decode(refresh\_token, SECRET\_KEY, algorithms=[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(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, SECRET\_KEY, algorithms=[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: User = db.query(User).filter(User.email == email).first()

if user is None:

raise credentials\_exception

return user

При створенні access\_token ми трохи змінили склад токена:

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

* iat (issued at) – це час, коли токен був виданий.
* exp (expire) - це час, коли термін дії токена спливає.
* scope використовується для зазначення типу токена: доступу або оновлення. Це використовується для обмеження доступу до певних функцій або ресурсів, залежно від типу токена.

async def create\_refresh\_token(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, SECRET\_KEY, algorithm=ALGORITHM)

return encoded\_refresh\_token

Метод create\_refresh\_token створює новий токен оновлення. Цей токен використовується для оновлення поточного токена доступу, коли він закінчується. Він приймає два аргументи: data - словник з даними, які потрібно закодувати в JWT та expires\_delta - час, протягом якого створюваний токен буде дійсний. Якщо expires\_delta не передано, токен буде дійсний протягом 7 днів. У словник to\_encode додається інформація про час створення токена iat, час закінчення токена exp і про те, що це рефреш токен в полі scope.

async def get\_email\_form\_refresh\_token(refresh\_token: str):

try:

payload = jwt.decode(refresh\_token, SECRET\_KEY, algorithms=[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')

Також для реалізації оновлення токена доступу використовується функція get\_email\_form\_refresh\_token, яка використовується для отримання з refresh\_token email користувача. Якщо токен успішно декодований і має правильний scope, то функція повертає email користувача. В іншому випадку викидається виняток HTTPException з кодом 401 та повідомленням `Could not validate credentials'.

Ми додали всередині функції get\_current\_user перевірку, що поле scope в заголовку токена дорівнює 'access\_token'. Якщо це так, то витягується значення поля "sub", яке містить email користувача, і повертається як результат. Якщо поле scope має інше значення, буде викликано виняток credentials\_exception. Якщо сталася помилка при декодуванні, викликається та сама credentials\_exception.

Інший код змін не зазнав.

**Основний застосунок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt_refresh#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D0%B8%D0%B9-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA)from fastapi import FastAPI, Depends, HTTPException, status, Security

from fastapi.security import OAuth2PasswordRequestForm, HTTPAuthorizationCredentials, HTTPBearer

from sqlalchemy.orm import Session

from pydantic import BaseModel

from auth import create\_access\_token, create\_refresh\_token, get\_email\_form\_refresh\_token, get\_current\_user, Hash

from db import User, get\_db

app = FastAPI()

hash\_handler = Hash()

security = HTTPBearer()

class UserModel(BaseModel):

username: str

password: str

@app.post("/signup")

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

exist\_user = db.query(User).filter(User.email == body.username).first()

if exist\_user:

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

new\_user = User(email=body.username, password=hash\_handler.get\_password\_hash(body.password))

db.add(new\_user)

db.commit()

db.refresh(new\_user)

return {"new\_user": new\_user.email}

@app.post("/login")

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

user = db.query(User).filter(User.email == body.username).first()

if user is None:

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

if not hash\_handler.verify\_password(body.password, user.password):

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

# Generate JWT

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

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

user.refresh\_token = refresh\_token

db.commit()

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

@app.get('/refresh\_token')

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

token = credentials.credentials

email = await get\_email\_form\_refresh\_token(token)

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

if user.refresh\_token != token:

user.refresh\_token = None

db.commit()

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

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

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

user.refresh\_token = refresh\_token

db.commit()

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

@app.get("/")

async def root():

return {"message": "Hello World"}

@app.get("/secret")

async def read\_item(current\_user: User = Depends(get\_current\_user)):

return {"message": 'secret router', "owner": current\_user.email}

Маршрут реєстрації залишився без змін. При аутентифікації, в маршруті /login, ми тепер створюємо та повертаємо два токени:

# Generate JWT

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

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

user.refresh\_token = refresh\_token

db.commit()

Також ми поміщаємо refresh\_token у базу даних.

У нас з'явився новий маршрут /refresh\_token та обробник для нього:

@app.get('/refresh\_token')

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

token = credentials.credentials

email = await get\_email\_form\_refresh\_token(token)

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

if user.refresh\_token != token:

user.refresh\_token = None

db.commit()

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

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

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

user.refresh\_token = refresh\_token

db.commit()

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

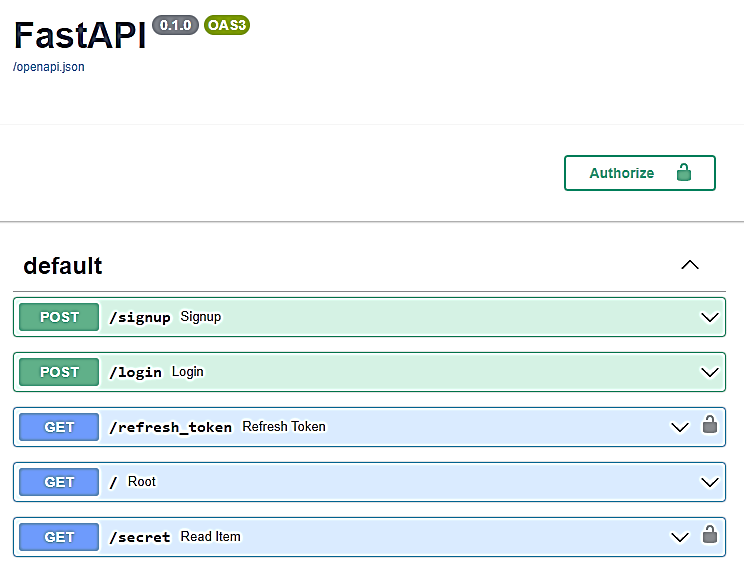
Функція приймає як параметр заголовок авторизації з refresh\_token, використовуючи механізм захисту Security з FastAPI.

Використовуючи HTTPAuthorizationCredentials з fastapi.security, ми отримуємо refresh\_token із заголовка запиту token=credentials.credentials. Потім, використовуючи функцію get\_email\_form\_refresh\_token, декодуємо токен і отримуємо email з нього. Якщо токен не відповідає токену, збереженому в базі даних, функція викликає виняток raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid refresh token") та видаляє поточний refresh\_token з бази даних. Можливо, він скомпрометований, і тепер клієнт повинен повторити операцію аутентифікації знову. Якщо токен є валідним, створюються нові токени доступу та оновлення, і старий токен оновлення refresh\_token оновлюємо в базі даних.

Тут необхідно пояснити один момент із оновленням refresh\_token у базі даних. В момент оновлення access\_token оновлюються обидва токени. Виникає питання - навіщо refresh\_token тоді термін життя, якщо ми оновлюємо його щоразу при оновленні access\_token? Це зроблено на випадок, якщо клієнт буде в офлайні більше 7 днів, тоді доведеться знову пройти процес аутентифікації. Якщо ж ви хочете, щоб рівно через 7 днів клієнт проходив процес аутентифікації, тоді ми оновлюємо тільки access\_token, а refresh\_token залишаємо той, що є в базі даних, і через 7 днів термін життя його закінчиться. Обидва підходи мають право на існування та все залежить від вашої реалізації.

**Перевірка працездатності**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-12/module-12-01/jwt_refresh#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%BF%D1%80%D0%B0%D1%86%D0%B5%D0%B7%D0%B4%D0%B0%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%96)

Зараз у нас реалізовані маршрути як на малюнку знизу:



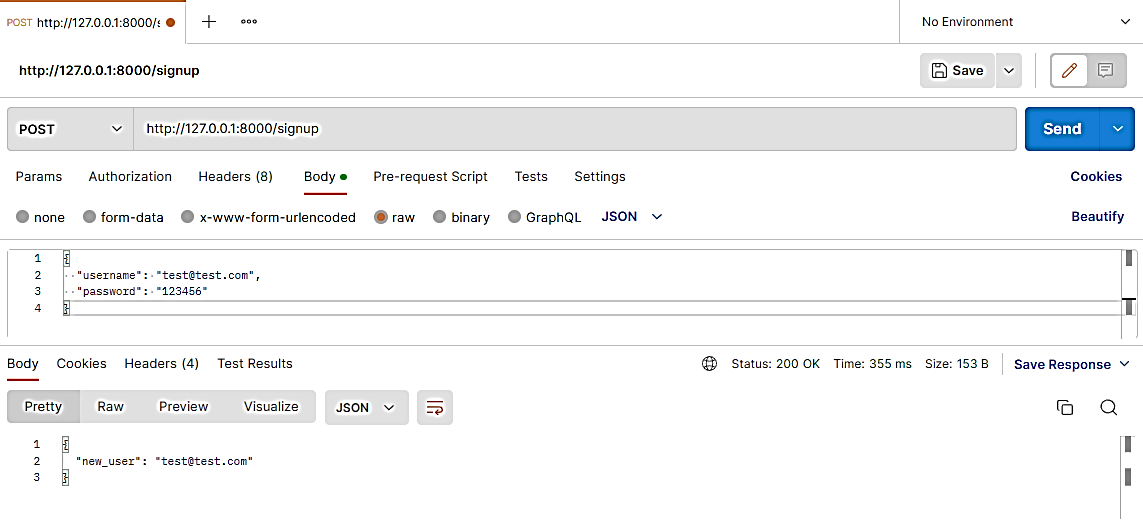
**Swagger документація прикладу**

Щоб перевірити працездатність нашого застосунку, ми можемо знову використовувати Swagger документацію. Але Swagger документація не вміє працювати з рефреш токенами, а використовує лише access\_token. Тому ми будемо використовувати інструмент розробника [Postman](https://www.postman.com/) (https://www.edu.goit.global/uk/learn/15249151/10926565/11865871/training?blockId=20111512). Для цього [завантажте](https://www.postman.com/downloads/) (https://www.postman.com/downloads/) - клієнт для роботи з API і приступимо.

INFO

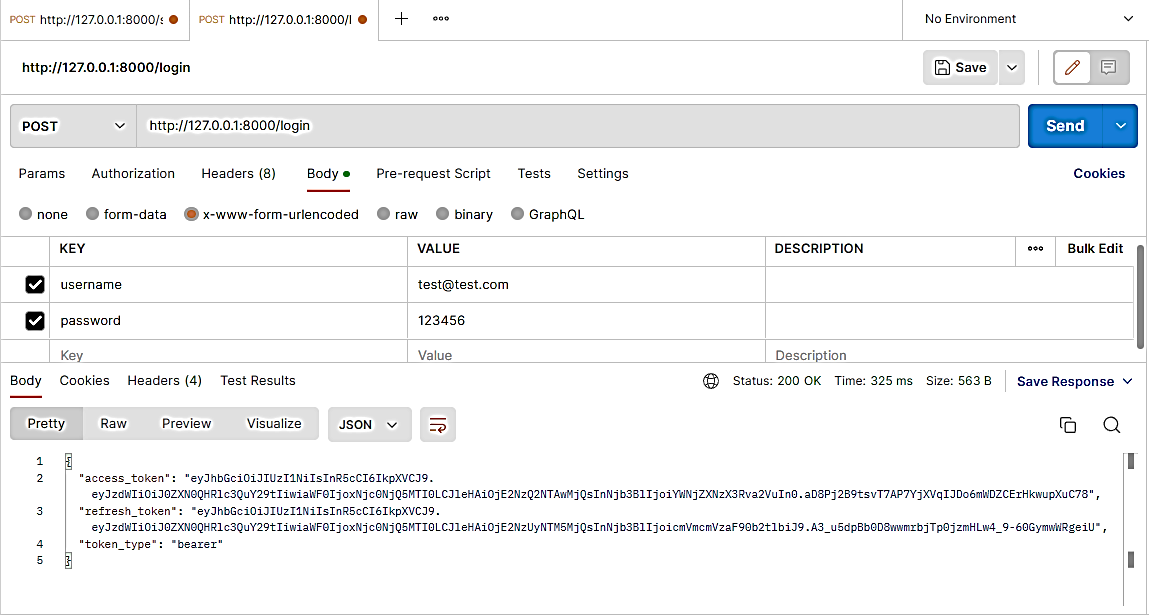
Postman – це інструмент, який допомагає розробникам протестувати та використовувати API. Він дозволяє надсилати запити HTTP, переглядати та аналізувати відповіді сервера, а також зберігати та організовувати запити в колекціях. Це корисно для розробки та відлагодження API, а також для документації та роботи з командою.

1. Створимо користувача. При виконанні операції необхідно буде передати тіло запиту у форматі application/json



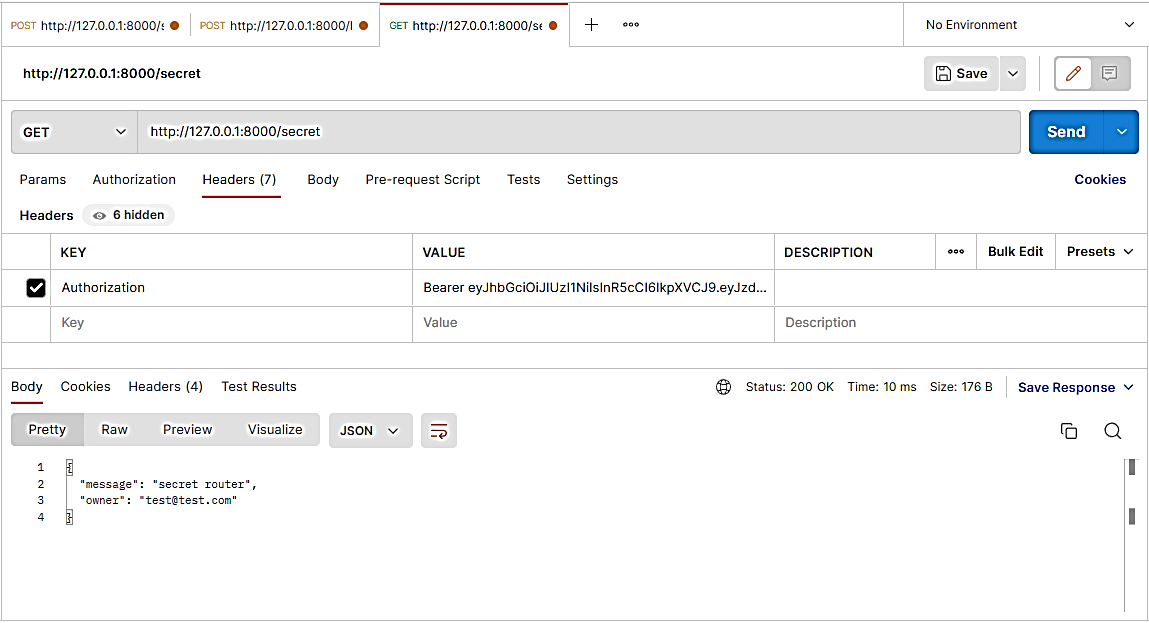
**POST - http://127.0.0.1:8000/signup**

2.Виконаємо аутентифікацію та отримаємо два токени. При виконанні операції необхідно буде передати тіло запиту у форматі application/x-www-form-urlencoded:



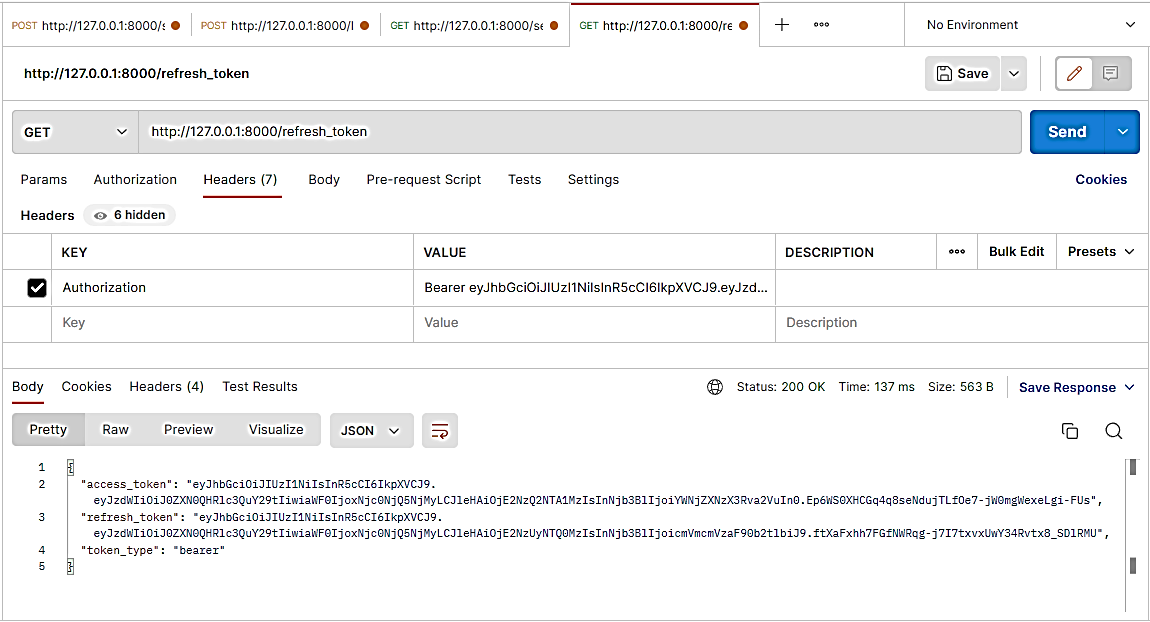
**POST - http://127.0.0.1:8000/login**

3.Перевіримо, що access\_token дає доступ до "секретного" маршруту. При виконанні операції необхідно буде передати заголовок Authorization зі значенням Bearer{access\_token}. Все працює.



**GET - http://127.0.0.1:8000/secret**

4.Перевіримо, що оновлення access\_token працює. При виконанні операції необхідно буде передати заголовок Authorization зі значенням Bearer {refresh\_token}



**GET - http://127.0.0.1:8000/refresh\_token**

Готовий результат повинен бути як у цьому живому прикладі, запустіть та подивіться код:

Щоб побачити Swagger документацію нашого застосунку, після запуску прикладу натисніть [**посилання**](https://fastapi-refresh-jwt.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-refresh-jwt#main.py)

Ми закінчили розбір JWT авторизації, настав час додати її до нашого REST API застосунку.