**Модуль 13: Розширені теми Beckend-розробки**

**Заняття 2: Основи безпеки вебзастосунків**

**Використання змінних середовища**

Змінні середовища можуть використовуватися для зберігання конфіденційної інформації, такої як паролі та ключі API сторонніх сервісів, які не повинні зберігатися в коді або комітитися в репозиторій.

У Python можна використовувати модуль os для роботи зі змінними середовища. Наприклад, щоб отримати значення змінної середовища API\_KEY, можна використовувати наступний код:

import os

api\_key = os.environ.get('API\_KEY')

Також для зручної роботи зі змінними середовища можна використовувати інші бібліотеки, такі як python-decouple та python-dotenv.

Важливо зазначити, що змінні середовища повинні бути задані на стороні сервера або в конфігураційних файлах. Це дає змогу уникнути можливості витоку конфіденційної інформації та забезпечує безпечнішу роботу з даними.

Крім того, під час використання змінних середовища в коді, необхідно враховувати ситуації, коли вони можуть бути недоступні й необхідно передбачити запасні значення або обробку винятків.

Бібліотека python-dotenv дозволяє використовувати змінні середовища з файлу .env у вашому проекті. Для використання python-dotenv необхідно встановити її за допомогою poetry:

poetry add python-dotenv

Після встановлення можна створити файл .env у кореневій директорії проекту та визначити змінні середовища у форматі NAME=VALUE. Наприклад:

API\_KEY=abcdefghijklmnopqrstuvwxyz

SECRET\_KEY=1234567890

DEBUG=True

DANGER

Файл .env завжди необхідно додавати до файлу .gitignore!

Зверніть увагу, що рядкові дані йдуть без лапок. Використання лапок у цьому випадку буде помилкою! Далі ви можете використовувати ці змінні у своєму коді, використовуючи функцію load\_dotenv з бібліотеки python-dotenv:

from dotenv import load\_dotenv

load\_dotenv()

Це завантажить змінні з файлу .env і додасть їх до словника os.environ, який можна використовувати для отримання значень змінних:

import os

api\_key = os.environ.get('API\_KEY')

secret\_key = os.environ.get('SECRET\_KEY')

debug = os.environ.get('DEBUG')

У цьому прикладі ми використовуємо функцію get зі словника os.environ для отримання значень змінних API\_KEY, SECRET\_KEY та DEBUG.

Важливо ще раз зазначити, що під час використання python-dotenv необхідно уникати коміту файлу .env в репозиторії і додати його в .gitignore, щоб уникнути розкриття секретної інформації.

Залежно від вашого сервера і середовища, вам може знадобитися використовувати інший спосіб встановлення змінних середовища, наприклад, через системні налаштування або використання конфігураційного файлу. У будь-якому випадку використання змінних середовища забезпечує безпечніше та гнучкіше розгортання застосунку.

**Змінні середовища REST API**

У нашого REST API достатньо багато налаштувань, які є конфіденційними та можуть змінюватися в різних середовищах під час запуску застосунку. Корисно помістити їх у файл .env, а потім прочитати їх з нього, ніби вони були змінними середовища.

Створимо файл .env у корені нашого проекту та помістимо туди наступні змінні:

**.env**

POSTGRES\_DB=rest\_app

POSTGRES\_USER=postgres

POSTGRES\_PASSWORD=567234

POSTGRES\_PORT=5432

SQLALCHEMY\_DATABASE\_URL=postgresql+psycopg2://${POSTGRES\_USER}:${POSTGRES\_PASSWORD}@localhost:${POSTGRES\_PORT}/${POSTGRES\_DB}

SECRET\_KEY=secret\_key

ALGORITHM=HS256

MAIL\_USERNAME=example@meta.ua

MAIL\_PASSWORD=secretPassword

MAIL\_FROM=example@meta.ua

MAIL\_PORT=465

MAIL\_SERVER=smtp.meta.ua

REDIS\_HOST=localhost

REDIS=6379

Тут – налаштування підключення до наших баз даних, секретний ключ для шифрування токена JWT і його алгоритм, а також дані сервера електронної пошти.

Pydantic надає чудову утиліту для обробки цих налаштувань, що надходять зі змінних середовища. Створимо файл src/conf/config.py і всередині помістимо наступний код:

**src/conf/config.py**

from pydantic import BaseSettings

class Settings(BaseSettings):

sqlalchemy\_database\_url: str

secret\_key: str

algorithm: str

mail\_username: str

mail\_password: str

mail\_from: str

mail\_port: int

mail\_server: str

redis\_host: str = 'localhost'

redis\_port: int = 6379

class Config:

env\_file = ".env"

env\_file\_encoding = "utf-8"

settings = Settings()

Імпортуємо з Pydantic модель BaseSettings і створимо клас Settings, наслідуючи його від класу BaseSettings. Клас Settings визначає кілька змінних класу як поля, включаючи інформацію про підключення до бази даних, секретний ключ, алгоритм, дані сервера електронної пошти та дані сервера Redis. Так само як і у випадку з моделями, атрибути класу можуть оголошуватися з анотаціями типів і, можливо, значеннями за замовчуванням. Ви можете використовувати ті самі функції та інструменти перевірки, які ви використовуєте для моделей pydantic, такі як різні типи даних і додаткові перевірки за допомогою Field.

Внутрішній клас Config задає розташування файлу середовища .env та його кодування utf-8. Це дасть змогу прочитати вміст файлу .env і присвоїти відповідній змінній своє значення. Як бачимо, регістр значення не має.

Нарешті, створюється об’єкт settings як екземпляр класу Settings. Цей об’єкт можна використовувати для доступу до інформації про конфігурацію, що зберігається у файлі середовища .env.

Поля, визначені в класі Settings, заповнені змінними середовища, і до них можна отримати доступ у коді за допомогою об’єкта settings, наприклад: settings.sqlalchemy\_database\_url або settings.mail\_server.

Пройдемося нашим проектом і замінимо всі важливі дані на змінні з об’єкта settings.

Спочатку замінимо рядок підключення в базі даних у файлі src/database/db.py. Тепер у коді ми не зберігаємо ім’я користувача і пароль доступу до бази даних, що набагато безпечніше.

**src/database/db.py**

SQLALCHEMY\_DATABASE\_URL = settings.sqlalchemy\_database\_url

Далі замінимо критичні дані в сервісі авторизації Auth:

**src/services/auth.py**

...

class Auth:

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

SECRET\_KEY = settings.secret\_key

ALGORITHM = settings.algorithm

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

r = redis.Redis(host=settings.redis\_host, port=settings.redis\_port, db=0)

...

Також поміняємо електронну пошту і пароль та інші параметри на значення з об’єкта settings у сервісі надсилання листів:

**src/services/email.py**

conf = ConnectionConfig(

MAIL\_USERNAME=settings.mail\_username,

MAIL\_PASSWORD=settings.mail\_password,

MAIL\_FROM=EmailStr(settings.mail\_from),

MAIL\_PORT=settings.mail\_port,

MAIL\_SERVER=settings.mail\_server,

MAIL\_FROM\_NAME="Desired Name",

MAIL\_STARTTLS=False,

MAIL\_SSL\_TLS=True,

USE\_CREDENTIALS=True,

VALIDATE\_CERTS=True,

TEMPLATE\_FOLDER=Path(\_\_file\_\_).parent / 'templates',

)

Останній штрих – це використання змінних середовища всередині файлу docker-compose.yml. Тут ми використовуємо плейсхолдери ${POSTGRES\_DB}.

Під час запуску Docker побачить плейсхолдер і автоматично підставить значення змінної POSTGRES\_DB з файлу .env на місце плейсхолдера.

**docker-compose.yml**

version: '3'

services:

redis:

image: redis:alpine

ports:

- "6379:6379"

postgres:

image: postgres:12

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

POSTGRES\_PASSWORD: ${POSTGRES\_PASSWORD}

ports:

- "5432:5432"

volumes:

- ./postgres-data:/var/lib/postgresql/data

Тепер ми забезпечили безпеку всіх наших конфіденційних даних у проекті.

**Змінні середовища Django**

У проекті Django всі конфіденційні дані захисту знаходяться у файлі notes/notes/settings.py.

Ми будемо використовувати пакет[**Django Environ**](https://django-environ.readthedocs.io/en/latest/install.html) (https://django-environ.readthedocs.io/en/latest/install.html)**.** Встановімо його:

poetry add django-environ

Після цього імпортуємо environ у settings.py та ініціалізуємо його:

import environ

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

env = environ.Env()

environ.Env.read\_env(BASE\_DIR / '.env')

Створимо файл .env у кореневому каталозі та не забудемо додати його до .gitignore. Оголосимо змінні середовища в .env і переконаємося, що ми не використовуємо лапки навколо рядків.

**.env**

SECRET\_KEY=django-insecure-6njcs!&=f6o)\_=tqieoj6+wzljxwg=71l9th&n\_m8w$o1c\*xe3

DATABASE\_NAME=django\_email

DATABASE\_USER=postgres

DATABASE\_PASSWORD=567234

DATABASE\_HOST=127.0.0.1

DATABASE\_PORT=5432

EMAIL\_HOST=smtp.meta.ua

EMAIL\_PORT=465

EMAIL\_HOST\_USER=example@meta.ua

EMAIL\_HOST\_PASSWORD=secretPassword

Замінимо всі посилання на змінні середовища у файлі settings.py

1. Секретний ключ:

SECRET\_KEY = env('SECRET\_KEY')

2.Підключення до бази даних

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': env('DATABASE\_NAME'),

'USER': env('DATABASE\_USER'),

'PASSWORD': env('DATABASE\_PASSWORD'),

'HOST': env('DATABASE\_HOST'),

'PORT': env('DATABASE\_PORT'),

}

}

3.Змінні для надсилання листів

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = env('EMAIL\_HOST')

EMAIL\_PORT = env('EMAIL\_PORT')

EMAIL\_STARTTLS = False

EMAIL\_USE\_SSL = True

EMAIL\_USE\_TLS = False

EMAIL\_HOST\_USER = env('EMAIL\_HOST\_USER')

EMAIL\_HOST\_PASSWORD = env('EMAIL\_HOST\_PASSWORD')

DEFAULT\_FROM\_EMAIL = EMAIL\_HOST\_USER

Все, ми забезпечили безпеку всіх наших конфіденційних даних у проекті Django.

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

**Обмеження кількості запитів для REST API**

Лімітування кількості запитів до маршруту може бути необхідним для низки причин, включаючи:

* Захист від DDoS-атак: Якщо сервер отримує велику кількість запитів за короткий час, він може перевантажитися і почати відповідати повільно або навіть не відповідати зовсім. Лімітування кількості запитів може допомогти захистити сервер від таких атак.
* Обмеження ресурсів: Якщо маршрут використовує дорогі ресурси (наприклад, доступ до бази даних або обчислювальні ресурси), обмеження кількості запитів допоможе уникнути вичерпання цих ресурсів.
* Обмеження пропускної спроможності: Якщо маршрут використовує обмежені пропускні спроможності (наприклад, хмарний сервіс, який має обмежену кількість запитів за одиницю часу), лімітування кількості запитів може допомогти уникнути перевищення цього ліміту.
* Обмеження кількості запитів від одного користувача: Лімітування кількості запитів може бути використане для обмеження кількості запитів, які може надіслати один користувач, наприклад, для запобігання автоматизованому збору даних.

Для реалізації механізму обмеження кількості запитів для маршруту у FastAPI можна використовувати кілька бібліотек:

* [**fastapi-ratelimiter**](https://fastapi-ratelimit.readthedocs.io/en/latest/) (https://fastapi-ratelimit.readthedocs.io/en/latest/)
* [**SlowApi**](https://github.com/laurentS/slowapi) (https://github.com/laurentS/slowapi)
* [**fastapi-limiter**](https://github.com/long2ice/fastapi-limiter) (https://github.com/long2ice/fastapi-limiter)

Ці пакети для обмеження швидкості запитів у FastAPI. Основна перевага пакетів полягає в тому, що вони забезпечують простий спосіб обмеження швидкості запитів API для різних форм зловживань. Це може бути як спам, так і інтенсивне використання користувачами доступу до вашого REST API.

Також потрібно пам’ятати про недолік цих пакетів. Вони можуть додавати накладні витрати в цикл запиту-відповіді, потенційно сповільнюючи продуктивність вашого REST API. Крім того, обмеження швидкості може негативно позначитися на користувацькому досвіді, якщо воно застосовується занадто агресивно. Важливо ретельно оцінити потреби REST API в обмеженні швидкості та вибрати відповідний алгоритм і налаштування обмеження швидкості, щоб збалансувати безпеку і продуктивність вашого застосунку.

INFO

Для промислових застосунків, для захисту від DDoS-атак і не тільки, хорошим рішенням буде використання сервісу [Cloudflare](https://www.cloudflare.com/) (https://www.cloudflare.com/). Cloudflare – це компанія, яка надає низку мережевих послуг, таких як захист від DDoS-атак, кешування, оптимізація продуктивності та інші. Вона використовує безліч дата-центрів по всьому світу, щоб забезпечити швидке завантаження сайтів і застосунків для користувачів. Вона також надає сервіси безпеки, такі як захист від злому і захист від шахрайства, які допомагають захистити сайти від шкідливих атак.

**Приклад**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ratelimit#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4)У нашому REST API ми будемо використовувати бібліотеку fastapi-limiter.

Розберемо приклад коду з базовою реалізацією:

import redis.asyncio as redis

import uvicorn

from fastapi import Depends, FastAPI

from fastapi\_limiter import FastAPILimiter

from fastapi\_limiter.depends import RateLimiter

app = FastAPI()

@app.on\_event("startup")

async def startup():

r = await redis.Redis(host='localhost', port=6379, db=0, encoding="utf-8", decode\_responses=True)

await FastAPILimiter.init(r)

@app.get("/", dependencies=[Depends(RateLimiter(times=2, seconds=5))])

async def index():

return {"msg": "Hello World"}

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

uvicorn.run("app:app", reload=True)

На старті застосунку @app.on\_event("startup") метод FastAPILimiter.init використовується для ініціалізації з’єднання з Redis. Це дає змогу використовувати Redis для зберігання інформації про обмеження швидкості.

Залежність RateLimiter використовується для встановлення обмеження швидкості для маршруту "/". Обмеження швидкості встановлено на 2 запити за 5 секунд. Це означає, що користувач може зробити лише 2 запити до цього маршруту протягом 5 секунд. Якщо користувач перевищить цей ліміт, він отримає відповідь 429 Too Many Requests.

Цей приклад демонструє, як використовувати пакет fastapi-limiter для обробки обмеження швидкості в застосунку FastAPI.

Готовий результат повинен бути як у цьому живому прикладі нижче, запустіть і подивіться код.

Щоб побачити документацію Swagger застосунку, після запуску прикладу натисніть [посилання](https://fastapi-ratelimit.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-ratelimit) та виконайте кілька запитів, щоб отримати 429 Too Many Requests.

**Додаємо обмеження до застосунку**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ratelimit#%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-%D0%BE%D0%B1%D0%BC%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BA%D1%83)Додамо, для прикладу, що користувач може робити не більше 10 запитів на хвилину за маршрутом api/notes. Для цього нам потрібно змінити файл main.py застосунку та встановити пакет fastapi-limiter у проект.

**main.py**

import redis.asyncio as redis

from fastapi import FastAPI

from fastapi\_limiter import FastAPILimiter

from src.routes import notes, tags, auth

from src.conf.config import settings

app = FastAPI()

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

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

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

@app.on\_event("startup")

async def startup():

r = await redis.Redis(host=settings.redis\_host, port=settings.redis\_port, db=0, encoding="utf-8",

decode\_responses=True)

await FastAPILimiter.init(r)

@app.get("/")

def read\_root():

return {"message": "Hello World"}

А для маршруту встановити обмеження:

**src/routes/notes.py**

@router.get("/", response\_model=List[NoteResponse], description='No more than 10 requests per minute',

dependencies=[Depends(RateLimiter(times=10, seconds=60))])

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

Не забувши вказати через параметр description опис змінених умов для документації Swagger.

**Cross-Origin Resource Sharing**

Cross-Origin Resource Sharing (CORS) – це механізм, який використовується для дозволу або заборони запитів до ресурсів із різних джерел (доменів). Цей механізм створений для захисту від атак типу «кросдоменний скриптинг» (XSS) та «кросдоменний запит» (CSRF). Коли браузер надсилає запит на сервер, він додає заголовок «Origin» з ім’ям домену, з якого ініційовано запит. Сервер може дозволити або заборонити запит на основі цієї інформації.

З метою безпеки браузери обмежують CORS-запити, ініційовані скриптами. Наприклад, XMLHttpRequest і Fetch API дотримуються політики одного джерела (same-origin policy). Це означає, що вебзастосунки, які використовують такі API, можуть запитувати HTTP-ресурси тільки з того домену, з якого їх було завантажено, поки не буде використано CORS-заголовки. Простими словами — код JavaScript клієнта з одного домену не може отримати дані від іншого домену без особливого дозволу – **CORS.**

Припустимо, у нас є клієнтський застосунок та його JavaScript намагається взаємодіяти з нашим REST API, що працює за адресою http://localhost:8000. Сам клієнтський застосунок працює, наприклад, за адресою http://localhost:3000. Браузер відправить HTTP-запит на серверну частину (http://localhost:8000), і якщо серверна частина надішле відповідні заголовки **CORS**, які дозволяють зв’язок з цього іншого джерела (http://localhost:3000), то браузер дозволить JavaScript у клієнтській частині (Frontend) відправити свій запит до нашого застосунку REST API (Backend). Для цього наш Backend повинен мати список дозволених джерел, і щоб Frontend працював коректно, нам доведеться включити до цього списку http://localhost:3000

ЗНАКИ ПІДСТАВЛЕННЯ

Також можна оголосити список як знак підставлення, щоб сказати, що всі адреси дозволені."\*"

Для використання CORS у FastAPI ми імпортуємо CORSMiddleware з пакета fastapi:

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

Створюємо екземпляр застосунку:

app = FastAPI()

Визначаємо список доменів, які можуть надсилати запити до нашого API:

origins = [

"http://localhost:3000"

]

Додаємо CORSMiddleware у наш застосунок:

app.add\_middleware(

CORSMiddleware,

allow\_origins=origins,

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

Коли клієнт надсилає запит до API, FastAPI аналізує заголовок Origin у запиті.

Якщо домен в Origin міститься у списку origins, FastAPI додає у відповідь заголовок Access-Control-Allow-Origin зі значенням цього домену. Таким чином, браузер розуміє, що домен, який надіслав запит, має право отримати дані з API.

Розшифруємо параметри проміжного програмного забезпечення:

* allow\_origins – цей параметр визначає список джерел, яким дозволено доступ до застосунку.
* allow\_credentials – цей параметр встановлює значення True, що означає, що дозволені кросдоменні запити з урахуванням облікових даних.
* allow\_methods – цей параметр визначає список дозволених методів HTTP, які можуть використовуватися для кросдоменних запитів. Значення "\*" означає, що всі методи дозволені.
* allow\_headers – цей параметр визначає список дозволених заголовків HTTP, які можуть використовуватися в кросдоменних запитах. Значення "\*" означає, що всі заголовки дозволені.

Цього достатньо, щоб клієнтський застосунок з адреси http://localhost:3000 міг працювати з нашим REST API.

**Обмеження доступу за IP та User Agent**

**IP-бан**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ip#ip-%D0%B1%D0%B0%D0%BD)IP-бан – це механізм обмеження доступу до сайту або застосунку на основі IP-адреси. Цей механізм може використовуватися для блокування зловмисних або небажаних користувачів або для обмеження кількості запитів з однієї IP-адреси. Для реалізації IP-бану у FastAPI можна використовувати міддлвари (проміжне ПЗ).

У разі використання IP-бану можна використовувати модуль ip\_address зі стандартної бібліотеки Python для перевірки IP-адреси клієнта. Наприклад, можна створити список заблокованих IP-адрес і порівнювати їх з IP-адресою клієнта під час кожного запиту.

**Чорний список**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ip#%D1%87%D0%BE%D1%80%D0%BD%D0%B8%D0%B9-%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA)Варіант 1: бан IP-адреси або використання чорного списку адрес.

from ipaddress import ip\_address

from typing import Callable

from fastapi import FastAPI, Request, status

from fastapi.responses import JSONResponse

app = FastAPI()

banned\_ips = [ip\_address("192.168.1.1"), ip\_address("192.168.1.2"), ip\_address("127.0.0.1")]

@app.middleware("http")

async def ban\_ips(request: Request, call\_next: Callable):

ip = ip\_address(request.client.host)

if ip in banned\_ips:

return JSONResponse(status\_code=status.HTTP\_403\_FORBIDDEN, content={"detail": "You are banned"})

response = await call\_next(request)

return response

@app.get("/")

def read\_root():

return {"message": "Hello world"}

У цьому прикладі ми створюємо список banned\_ips із забороненими IP-адресами та визначаємо middleware функцію ban\_ips, яка перевірятиме IP-адресу клієнта на предмет відповідності списку заборонених адрес. Якщо IP-адреса клієнта знаходиться в списку заборонених, проміжне ПЗ повертає JSONResponse з кодом стану 403 (Forbidden) та повідомленням "You are banned". Якщо IP-адреса клієнта не перебуває у списку заборонених, запит передається наступному проміжному ПЗ або функції-обробнику та повертається його відповідь.

**Білий список**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ip#%D0%B1%D1%96%D0%BB%D0%B8%D0%B9-%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA)Варіант 2 – це використання білого списку IP-адрес. У цьому випадку ми будемо дозволяти доступ лише до списку авторизованих IP-адрес.

from ipaddress import ip\_address

from typing import Callable

from fastapi import FastAPI, Request, status

from fastapi.responses import JSONResponse

app = FastAPI()

ALLOWED\_IPS = [ip\_address('192.168.1.0'), ip\_address('172.16.0.0'), ip\_address("127.0.0.1")]

@app.middleware("http")

async def limit\_access\_by\_ip(request: Request, call\_next: Callable):

ip = ip\_address(request.client.host)

if ip not in ALLOWED\_IPS:

return JSONResponse(status\_code=status.HTTP\_403\_FORBIDDEN, content={"detail": "Not allowed IP address"})

response = await call\_next(request)

return response

@app.get("/")

def read\_root():

return {"message": "Hello world"}

Функція проміжного ПЗ limit\_access\_by\_ip застосовується до всіх HTTP-запитів за допомогою декоратора @app.middleware("http"). Функція приймає як параметри об’єкт запиту request і функцію call\_next, яка є наступною функцією в ланцюжку проміжного ПЗ.

Спочатку функція витягує IP-адресу клієнта з об’єкта запиту та перевіряє, чи є вона у списку ALLOWED\_IPS. Якщо IP-адреса відсутня у списку, функція повертає JSONResponse з кодом стану HTTP\_403\_FORBIDDEN та повідомленням про те, що доступ не дозволено.

Якщо IP знаходиться у списку дозволених, функція викликає await call\_next(request) для продовження обробки запиту в наступній функції проміжного ПЗ або функції-обробнику маршруту, і повертає відповідь, згенеровану цією функцією.

Функція read\_root – це проста кінцева точка (endpoint), яка повертає відповідь у форматі JSON з повідомленням " Hello world". Цей маршрут буде доступний лише у тому випадку, якщо IP-адреса клієнта знаходиться у списку ALLOWED\_IPS.

INFO

Кінцева точка (endpoint) – це адреса на конкретний ресурс у мережі, до якої можна звернутися для отримання інформації або виконання певних дій. В API – це певний маршрут, який веде до конкретного ресурсу. Наприклад, у нашому вебзастосунку URL-адреса /api/notes/ – кінцева точка для отримання списку всіх записів.

**Бан User Agent**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-02/ip#%D0%B1%D0%B0%D0%BD-user-agent)User-Agent (UA) – це рядок, який браузер надсилає на сервер у заголовку HTTP-запиту. Вона містить інформацію про браузер, операційну систему, версію та інші деталі. Сервер може використовувати цю інформацію для ідентифікації та аналізу клієнта, а також для сумісності зі старими браузерами або для визначення мобільних пристроїв.

Для бану на основі рядка User Agent можна використовувати регулярні вирази для перевірки відповідності рядка User Agent заблокованим значенням.

import re

from typing import Callable

from fastapi import FastAPI, Request, status

from fastapi.responses import JSONResponse

app = FastAPI()

user\_agent\_ban\_list = [r"Gecko", r"Python-urllib"]

@app.middleware("http")

async def user\_agent\_ban\_middleware(request: Request, call\_next: Callable):

user\_agent = request.headers.get("user-agent")

for ban\_pattern in user\_agent\_ban\_list:

if re.search(ban\_pattern, user\_agent):

return JSONResponse(status\_code=status.HTTP\_403\_FORBIDDEN, content={"detail": "You are banned"})

response = await call\_next(request)

return response

@app.get("/")

def read\_root():

return {"message": "Hello world"}

Наша функція проміжного ПЗ user\_agent\_ban\_middleware перевіряє заголовок "user-agent" вхідних запитів на відповідність списку заборонених рядків користувацьких агентів, що зберігаються в user\_agent\_ban\_list. Якщо рядок агента користувача відповідає будь-якому із заборонених шаблонів, проміжне ПЗ повертає JSON-відповідь з кодом статусу HTTP 403 (Forbidden) та повідомленням "You are banned".

Якщо рядок агента користувача не заборонений, проміжне ПЗ викликає наступну функцію проміжного ПЗ або обробник кінцевої точки (в цьому випадку read\_root) для продовження обробки запиту та повертає свою відповідь.

**Збереження файлів у хмарі**

Робота з файлами на своєму сервері пов’язана з низкою труднощів. Тому, наприклад, зображення користувачів, такі як аватари або інший контент, зберігають у якомусь хмарному сервісі. Які переваги зберігання файлів у хмарі?

1. Збільшення продуктивності: Зберігання зображень у хмарі може значно поліпшити продуктивність сайту, оскільки всі навантаження будуть винесені на сторонній сервер.
2. Збільшення доступності: Зберігання зображень у хмарі дає змогу збільшити їхню доступність і підвищити захист у разі втрати працездатності сервера.
3. Економія місця: Хмарне зберігання дає змогу заощадити місце на локальному сервері, що своєю чергою покращує його продуктивність.
4. Економія грошей: Зберігання зображень у хмарі, зазвичай, є дешевшим рішенням, ніж встановлення та обслуговування власного файлового сервера.

Також потрібно розуміти, що зберігання зображень у хмарі безпечніше, тому що в такому разі у вас є додатковий рівень захисту даних у вигляді резервного копіювання та масштабованості, що надаються сервісом зберігання даних. Це означає, що в разі виникнення будь-яких проблем, таких як вихід з ладу жорсткого диска або вірусна атака, ви зможете відновити дані з хмари.

Існує багато сервісів для зберігання зображень у хмарі, серед них:

* Amazon S3
* Google Cloud Storage
* Microsoft Azure Blob Storage

Цей список можна продовжувати довго, але ми виберемо сервіс [**Cloudinary**](https://cloudinary.com/) (https://cloudinary.com/)**.**

Одна з переваг якого — можливість використовувати безплатно невеликий обсяг інформації для зберігання файлів. Цього буде достатньо для виконання наших прикладів і домашнього завдання.

INFO

Cloudinary є провідним провайдером послуг хмарного зберігання та обробки мультимедіа. Він пропонує широкий спектр послуг, включно зі зберіганням, перетворенням, оптимізацією, управлінням і делегуванням зображень, відео та аудіо. Він використовує розширені алгоритми машинного навчання, щоб забезпечити високу якість зображень і мінімізувати час завантаження. Таким чином, Cloudinary може істотно спростити роботу із зображеннями та відео, а також поліпшити продуктивність вашого сайту.

Після реєстрації для того, щоб працювати із сервісом, на Dashboard ми отримаємо Product Environment Credentials. Помістимо їх у наш файл .env (прим. Замініть їх на свої значення).

**.env**

CLOUDINARY\_NAME=cloud\_name

CLOUDINARY\_API\_KEY=12345678

CLOUDINARY\_API\_SECRET=api\_secret

Встановимо необхідну [**бібліотеку**](https://cloudinary.com/documentation/python_quickstart) (<https://cloudinary.com/documentation/python> \_quickstart) для роботи:

poetry add cloudinary

Розширимо наш клас Settingns новими змінними:

**src/conf/config.py**

from pydantic import BaseSettings

class Settings(BaseSettings):

sqlalchemy\_database\_url: str

secret\_key: str

algorithm: str

mail\_username: str

mail\_password: str

mail\_from: str

mail\_port: int

mail\_server: str

redis\_host: str = 'localhost'

redis\_port: int = 6379

cloudinary\_name: str

cloudinary\_api\_key: str

cloudinary\_api\_secret: str

class Config:

env\_file = ".env"

env\_file\_encoding = "utf-8"

settings = Settings()

Зараз наш аватар – це посилання, згенероване сервісом Gravatar. Ми збираємося дозволити користувачеві завантажувати свої аватари в нашому REST API. Для цього визначимо новий роутинг /api/users та помістимо його у файл src/routes/users.py.

**src/routes/users.py**

from fastapi import APIRouter, Depends, status, UploadFile, File

from sqlalchemy.orm import Session

import cloudinary

import cloudinary.uploader

from src.database.db import get\_db

from src.database.models import User

from src.repository import users as repository\_users

from src.services.auth import auth\_service

from src.conf.config import settings

from src.schemas import UserDb

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

@router.get("/me/", response\_model=UserDb)

async def read\_users\_me(current\_user: User = Depends(auth\_service.get\_current\_user)):

return current\_user

@router.patch('/avatar', response\_model=UserDb)

async def update\_avatar\_user(file: UploadFile = File(), current\_user: User = Depends(auth\_service.get\_current\_user),

db: Session = Depends(get\_db)):

cloudinary.config(

cloud\_name=settings.cloudinary\_name,

api\_key=settings.cloudinary\_api\_key,

api\_secret=settings.cloudinary\_api\_secret,

secure=True

)

r = cloudinary.uploader.upload(file.file, public\_id=f'NotesApp/{current\_user.username}', overwrite=True)

src\_url = cloudinary.CloudinaryImage(f'NotesApp/{current\_user.username}')\

.build\_url(width=250, height=250, crop='fill', version=r.get('version'))

user = await repository\_users.update\_avatar(current\_user.email, src\_url, db)

return user

Для того щоб користувач оновив зображення свого аватара, ми використовуємо маршрут /users/avatar та операцію PATCH. Завантажений файл приймається через параметр file, який буде об’єктом UploadFile з модуля fastapi.UploadFile.

Ми використовуємо бібліотеку cloudinary для завантаження зображення в хмару та його подальше зберігання. Функція cloudinary.config використовується для налаштування з’єднання з обліковим записом cloudinary, використовуючи значення з нашого модуля налаштувань. Потім функція cloudinary.uploader.upload використовується для завантаження файлу зображення, при цьому параметр public\_id встановлюємо відповідно до імені поточного користувача та папки NotesApp.

Після того як зображення завантажено, ми отримуємо URL-адресу зображення за допомогою класу cloudinary.CloudinaryImage та методу build\_url. Ми передаємо параметр version у функцію build\_url, яка генерує URL зображення із заданими параметрами ширини, висоти та кадру. За допомогою параметра version, можна вказати, яку саме версію зображення необхідно відобразити на сторінці.

Наприклад, якщо на сторінці раніше відображалася одна версія зображення, але користувач вніс зміни до зображення та перезавантажив його на сервер, то за допомогою параметру version можна отримати нову версію зображення з урахуванням змін, які були зроблені користувачем.

Значення параметра crop='fill' означає, що зображення буде масштабовано до заданого розміру, а потім обрізано до того, щоб повністю заповнити область з цим розміром. Цей параметр зазвичай використовується для забезпечення того, що зображення має певний розмір та зберігає пропорції, які потрібні для конкретного дизайну або макету.

Отримана URL-адреса потім передається у функцію repository\_users.update\_avatar, яка оновлює аватар користувача в базі даних.

**src/repository/users.py**

async def update\_avatar(email, url: str, db: Session) -> User:

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

user.avatar = url

db.commit()

return user

Нарешті, оновлена модель користувача повертається як відповідь на виклик API.

Залишилося додати новий роутинг у файл main.py

**main.py**

...

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

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

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

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

...

Таким чином, досить безпечно і просто ми змогли реалізувати складну операцію зі зберігання аватарів користувачів.

Ви також можете[**завантажити повний приклад коду**](https://github.com/GoIT-Python-Web/rest-api-secure)**(https://github.com/GoIT-Python-Web/rest-api-secure)** цього модуля для запуску на локальній машині.