**Веб-програмування**

## **Лабораторна робота №8**

«Документація та тестування API»

*Виконав Заяць Ілля, група ІПЗ-41*

**Мета:**

Метою даної лабораторної роботи є отримання практичних навичок з документування REST API та створення автоматизованих тестів для перевірки його функціональності.

Завдання:

1. Розробити детальну документацію для API, створеного у попередній лабораторній роботі

2. Реалізувати автоматизовані тести для перевірки API-ендпоінтів

3. Забезпечити моніторинг та логування API запитів

**Хід роботи**

1. Опрацювати теоретичні відомості.

2. Встановити необхідне програмне забезпечення для тестування.

3. Розширити API-документацію проекту з попередньої лабораторної роботи.

4. Розробити набір тестів для API.

5. Налаштувати логування та моніторинг.

6. У якості звіту в LMS Moodle вставити посилання на репозиторій GitHub.

7. Під час захисту звіту продемонструвати готову реалізацію завдання.

## *Структура проєкту:*

project/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py # Точка входу для FastAPI

│ ├── api/ # Модулі для обробки API маршрутів

│ │ ├── \_\_init\_\_.py

│ │ ├── items.py # Маршрути для товарів

│ │ ├── categories.py # Маршрути для категорій

│ │ └── auth.py # Маршрути для аутентифікації

│ │

│ ├── crud/ # Операції з базою даних

│ │ ├── \_\_init\_\_.py

│ │ ├── items.py # CRUD операції для товарів

│ │ ├── categories.py # CRUD операції для категорій

│ │ └── users.py # CRUD операції для користувачів

│ │

│ ├── models/ # SQLAlchemy моделі

│ │ ├── \_\_init\_\_.py

│ │ ├── item.py # Модель для товарів

│ │ ├── category.py # Модель для категорій

│ │ └── user.py # Модель для користувачів

│ │

│ ├── schemas/ # Pydantic схеми для валідації

│ │ ├── \_\_init\_\_.py

│ │ ├── item.py # Схема для товарів

│ │ ├── category.py # Схема для категорій

│ │ └── user.py # Схема для користувачів

│ │

│ ├── database.py # Налаштування бази даних

│ ├── auth.py # Аутентифікація (JWT)

│ └── config.py # Конфігурації для секретних ключів, алгоритмів

│

├── requirements.txt # Залежності проекту

├── .env # Секретні ключі та налаштування середовища

├── Dockerfile # Docker конфігурація для контейнеризації

└── README.md # Документація проекту

## *Код (приведений лише код нових файлів чи змінений код старих):*

**app/main.py — основна точка входу в додаток FastAPI:**

from fastapi import FastAPI

from .api import items, categories, auth

from .database import engine, SessionLocal

from .models import Base

# Ініціалізація FastAPI додатку

app = FastAPI()

# Підключення всіх ендпоінтів

app.include\_router(items.router)

app.include\_router(categories.router)

app.include\_router(auth.router)

# Створення таблиць в БД при запуску

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

# Визначення та ініціалізація бази даних

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

**app/api/items.py — маршрути для роботи з товарами:**

from fastapi import APIRouter, Depends, HTTPException

from sqlalchemy.orm import Session

from ..crud import get\_items, create\_item, update\_item, delete\_item

from ..schemas import ItemCreate, ItemUpdate

from ..models import Item

from ..database import get\_db

router = APIRouter()

@router.get("/items/")

def read\_items(db: Session = Depends(get\_db)):

return get\_items(db)

@router.post("/items/")

def create\_new\_item(item: ItemCreate, db: Session = Depends(get\_db)):

return create\_item(db, item)

@router.put("/items/{item\_id}")

def update\_existing\_item(item\_id: int, item: ItemUpdate, db: Session = Depends(get\_db)):

return update\_item(db, item\_id, item)

@router.delete("/items/{item\_id}")

def delete\_existing\_item(item\_id: int, db: Session = Depends(get\_db)):

delete\_item(db, item\_id)

return {"message": "Item deleted successfully"}

**app/api/categories.py — маршрути для роботи з категоріями:**

from fastapi import APIRouter, Depends

from sqlalchemy.orm import Session

from ..crud import get\_categories, create\_category

from ..schemas import CategoryCreate

from ..database import get\_db

router = APIRouter()

@router.get("/categories/")

def read\_categories(db: Session = Depends(get\_db)):

return get\_categories(db)

@router.post("/categories/")

def create\_new\_category(category: CategoryCreate, db: Session = Depends(get\_db)):

return create\_category(db, category)

**app/api/auth.py — маршрути для аутентифікації:**

from fastapi import APIRouter, Depends, HTTPException

from sqlalchemy.orm import Session

from ..schemas import UserCreate, UserLogin, Token

from ..crud import create\_user, authenticate\_user, refresh\_token

from ..auth import create\_access\_token, create\_refresh\_token

from ..database import get\_db

router = APIRouter()

@router.post("/auth/register")

def register\_user(user: UserCreate, db: Session = Depends(get\_db)):

return create\_user(db, user)

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

def login(user: UserLogin, db: Session = Depends(get\_db)):

user\_data = authenticate\_user(db, user)

if not user\_data:

raise HTTPException(status\_code=400, detail="Invalid credentials")

access\_token = create\_access\_token(user\_data)

refresh\_token\_ = create\_refresh\_token(user\_data)

return {"access\_token": access\_token, "refresh\_token": refresh\_token\_}

@router.post("/auth/refresh", response\_model=Token)

def refresh\_access\_token(refresh\_token\_: str):

new\_access\_token = refresh\_token(refresh\_token\_)

return {"access\_token": new\_access\_token}

**app/crud/items.py — операції з базою даних:**

from sqlalchemy.orm import Session

from ..models import Item

from ..schemas import ItemCreate, ItemUpdate

def get\_items(db: Session):

return db.query(Item).all()

def create\_item(db: Session, item: ItemCreate):

db\_item = Item(name=item.name, description=item.description, price=item.price, category\_id=item.category\_id)

db.add(db\_item)

db.commit()

db.refresh(db\_item)

return db\_item

def update\_item(db: Session, item\_id: int, item: ItemUpdate):

db\_item = db.query(Item).filter(Item.id == item\_id).first()

if db\_item:

db\_item.name = item.name

db\_item.description = item.description

db\_item.price = item.price

db.commit()

db.refresh(db\_item)

return db\_item

return None

def delete\_item(db: Session, item\_id: int):

db\_item = db.query(Item).filter(Item.id == item\_id).first()

if db\_item:

db.delete(db\_item)

db.commit()

**app/crud/items.py — операції з базою даних:**

from sqlalchemy.orm import Session

from ..models import Item

from ..schemas import ItemCreate, ItemUpdate

def get\_items(db: Session):

return db.query(Item).all()

def create\_item(db: Session, item: ItemCreate):

db\_item = Item(name=item.name, description=item.description, price=item.price, category\_id=item.category\_id)

db.add(db\_item)

db.commit()

db.refresh(db\_item)

return db\_item

def update\_item(db: Session, item\_id: int, item: ItemUpdate):

db\_item = db.query(Item).filter(Item.id == item\_id).first()

if db\_item:

db\_item.name = item.name

db\_item.description = item.description

db\_item.price = item.price

db.commit()

db.refresh(db\_item)

return db\_item

return None

def delete\_item(db: Session, item\_id: int):

db\_item = db.query(Item).filter(Item.id == item\_id).first()

if db\_item:

db.delete(db\_item)

db.commit()

**app/crud/categories.py — операції з базою даних:**

from sqlalchemy.orm import Session

from ..models import Category

from ..schemas import CategoryCreate

def get\_categories(db: Session):

return db.query(Category).all()

def create\_category(db: Session, category: CategoryCreate):

db\_category = Category(name=category.name)

db.add(db\_category)

db.commit()

db.refresh(db\_category)

return db\_category

**app/crud/users.py — операції з базою даних:**

from sqlalchemy.orm import Session

from ..models import User

from ..schemas import UserCreate

from passlib.context import CryptContext

# Ініціалізація контексту для хешування паролів

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

def get\_user\_by\_email(db: Session, email: str):

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

def create\_user(db: Session, user: UserCreate):

hashed\_password = pwd\_context.hash(user.password)

db\_user = User(email=user.email, hashed\_password=hashed\_password)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

def authenticate\_user(db: Session, email: str, password: str):

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

if user and pwd\_context.verify(password, user.hashed\_password):

return user

return None

**app/models/item.py — моделі SQLAlchemy:**

from sqlalchemy import Column, Integer, String, ForeignKey, Float

from sqlalchemy.orm import relationship

from ..database import Base

class Item(Base):

\_\_tablename\_\_ = "items"

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

name = Column(String, index=True)

description = Column(String)

price = Column(Float)

category\_id = Column(Integer, ForeignKey("categories.id"))

category = relationship("Category", back\_populates="items")

**app/models/category.py — моделі SQLAlchemy:**

from sqlalchemy import Column, Integer, String

from sqlalchemy.orm import relationship

from ..database import Base

class Category(Base):

\_\_tablename\_\_ = "categories"

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

name = Column(String, index=True)

items = relationship("Item", back\_populates="category")

**app/models/user.py — моделі SQLAlchemy:**

from sqlalchemy import Column, Integer, String

from ..database import Base

class User(Base):

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

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

email = Column(String, unique=True, index=True)

hashed\_password = Column(String)

**app/schemas/item.py — схеми для валідації даних:**

from pydantic import BaseModel

from typing import List, Optional

class ItemBase(BaseModel):

name: str

description: str

price: float

category\_id: int

class ItemCreate(ItemBase):

pass

class ItemUpdate(ItemBase):

name: Optional[str]

description: Optional[str]

price: Optional[float]

class Item(ItemBase):

id: int

class Config:

orm\_mode = True

**app/schemas/category.py — схеми для валідації даних:**

from pydantic import BaseModel

class CategoryBase(BaseModel):

name: str

class CategoryCreate(CategoryBase):

pass

class Category(CategoryBase):

id: int

class Config:

orm\_mode = True

**app/schemas/user.py — схеми для валідації даних:**

from pydantic import BaseModel, EmailStr

class UserBase(BaseModel):

email: EmailStr

class UserCreate(UserBase):

password: str

class User(UserBase):

id: int

class Config:

orm\_mode = True

**app/database.py — налаштування для бази даних:**

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "postgresql://user:password@localhost/dbname"

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

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

Base = declarative\_base()

**app/auth.py — логіка для аутентифікації:**

from datetime import datetime, timedelta

from jose import JWTError, jwt

from passlib.context import CryptContext

from .models import User

from .schemas import UserCreate

from .config import SECRET\_KEY, ALGORITHM

# Ініціалізація контексту для хешування паролів

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

# Функції для створення токенів

def create\_access\_token(data: dict, expires\_delta: timedelta = timedelta(hours=1)):

to\_encode = data.copy()

expire = datetime.utcnow() + expires\_delta

to\_encode.update({"exp": expire})

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

return encoded\_jwt

def create\_refresh\_token(data: dict, expires\_delta: timedelta = timedelta(days=7)):

to\_encode = data.copy()

expire = datetime.utcnow() + expires\_delta

to\_encode.update({"exp": expire})

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

return encoded\_jwt

**app/config.py — конфігурації для секретних ключів та алгоритмів:**

import os

# Зчитуємо секрети з файлу .env

SECRET\_KEY = os.getenv("SECRET\_KEY", "defaultsecretkey")

ALGORITHM = os.getenv("ALGORITHM", "HS256")

DATABASE\_URL = os.getenv("DATABASE\_URL", "postgresql://user:password@localhost/dbname")

**Dockerfile — залежності:**

# Використовуємо офіційний образ Python

FROM python:3.9-slim

# Встановлюємо робочий каталог

WORKDIR /app

# Копіюємо файли вимог

COPY requirements.txt /app/

# Встановлюємо залежності

RUN pip install --no-cache-dir -r requirements.txt

# Копіюємо весь код в контейнер

COPY . /app/

# Встановлюємо порт, який буде використовувати FastAPI

EXPOSE 8000

# Запускаємо сервер

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

**requirements.txt — залежності:**

fastapi

uvicorn

sqlalchemy

pydantic

psycopg2

passlib

python-dotenv

jose

**.env.test**

DATABASE\_URL=postgresql://user:password@localhost/dbname

SECRET\_KEY=your\_secret\_key

ALGORITHM=HS256

**README.md:**

# FastAPI Project

Цей проект є базовим додатком на базі FastAPI, який включає наступні можливості:

- CRUD операції для товарів та категорій

- Аутентифікація з використанням JWT

- Використання SQLAlchemy для роботи з базою даних

## Встановлення

1. Клонувати репозиторій:

```bash

git clone https://github.com/yourusername/project.git

```

2. Встановити залежності:

```bash

pip install -r requirements.txt

```

3. Налаштувати файл `.env` для підключення до вашої бази даних.

4. Запустити сервер:

```bash

uvicorn app.main:app --reload

```

## Точки доступу

- `/items/` - список всіх товарів

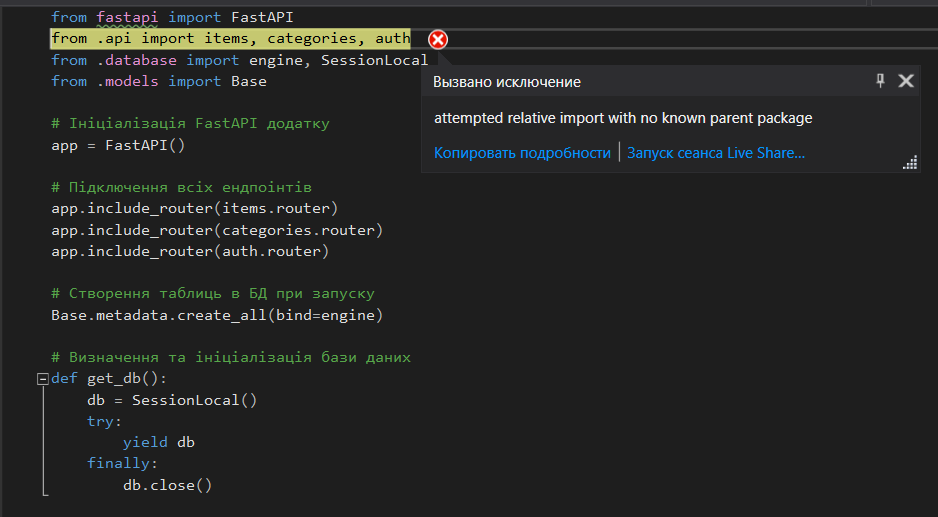
- `/categories/` - список всіх категорій

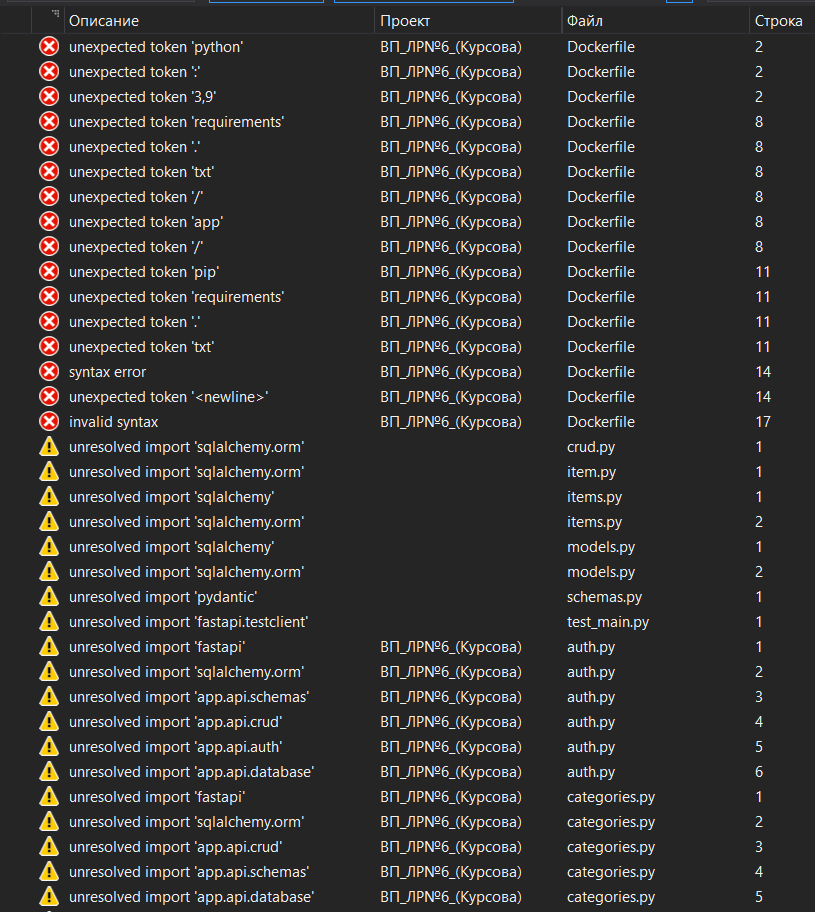
- `/auth/register` - реєстрація нового користувача

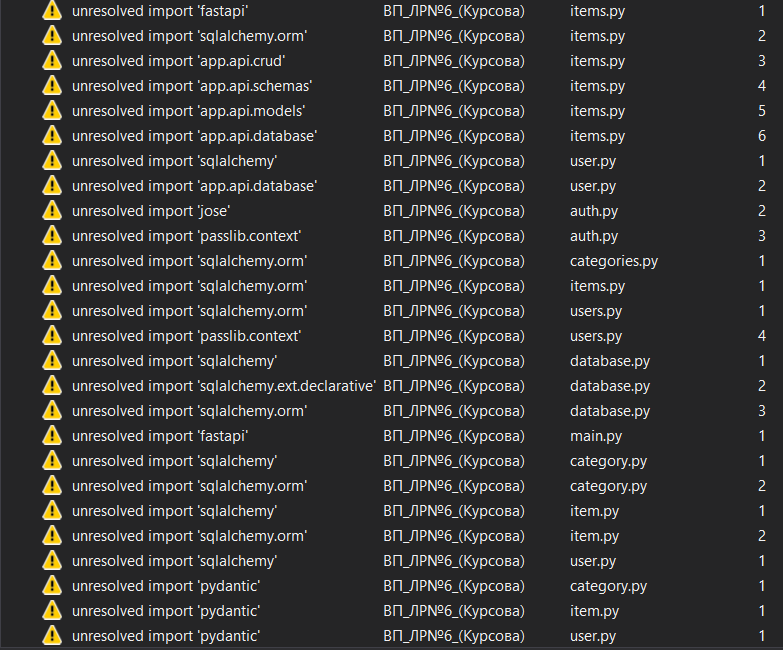
- `/auth/login` - вхід користувача та отримання JWT токена

## *Результат виведення::*

Оскільки я не виправив помилку в коді, то програма також не запускається коректно, але сам код, по ідеї, правильний. До того ж, до старих помилок, судячи зі всього, додалися нові.







**Висновки:** це доповнення значно покращує проект, роблячи його більш структурованим, масштабованим та зручним для роботи в команді. Використання сучасних практик, таких як розділення на окремі CRUD-операції, схеми і моделі, забезпечує кращу підтримку та розвиток коду в майбутньому. Впровадження Docker дозволяє зручно налаштувати середовище для розробки та тестування, що спрощує деплой та взаємодію між різними системами. Модульна структура дозволяє легко додавати нові функціональності без порушення існуючого коду. Ці зміни значно підвищують гнучкість, зручність та ефективність розробки порівняно з початковою версією проекту. На жаль, до вже існуючих помилок додалися нові.