**Веб-програмування**

## **Лабораторна робота №7**

«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:*

auto-parts-api/

├── app/ # Основний код додатку

│ ├── \_\_init\_\_.py # Порожній файл для позначення директорії як пакет

│ ├── main.py # Точка входу в додаток (FastAPI)

│ ├── api/ # Маршрути (ендпоінти) API

│ │ ├── \_\_init\_\_.py # Порожній файл для позначення директорії як пакет

│ │ ├── endpoints/ # Різні ендпоінти (категорії, товари, аутентифікація)

│ │ │ ├── items.py # Маршрути для роботи з товарами

│ │ │ ├── categories.py # Маршрути для роботи з категоріями

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

│ ├── crud/ # Операції з БД

│ │ └── crud.py # Функції для створення, отримання, оновлення, видалення даних

│ ├── models/ # SQLAlchemy моделі (для взаємодії з базою даних)

│ │ └── models.py # Моделі для товарів і категорій

│ ├── schemas/ # Схеми для валідації даних за допомогою Pydantic

│ │ └── schemas.py # Схеми для категорій і товарів

│ ├── database.py # Налаштування підключення до бази даних (SQLAlchemy)

│ ├── auth.py # Логіка для аутентифікації (JWT)

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

├── tests/ # Тести для перевірки API

│ ├── test\_main.py # Тести для доступності головних ендпоінтів

│ ├── test\_crud.py # Тести для CRUD операцій (створення, отримання, видалення)

├── init\_db.py # Ініціалізація бази даних (створення таблиць)

├── .env # Файл середовища (для налаштувань БД)

├── .env.test # Файл середовища для тестового середовища (для тестів)

├── Dockerfile # Dockerfile для контейнеризації додатку

├── docker-compose.yml # Docker Compose для запуску в контейнерах

├── requirements.txt # Залежності для проекту

├── README.md # Документація проекту

└── alembic/ # Налаштування для міграцій БД через Alembic

└── versions/ # Міграції БД

**Опис основних директорій і файлів:**

app/ — директорія, що містить увесь основний код додатку:

api/ — містить маршрути для API, кожен файл відповідає за певну частину (наприклад, items.py — для товарів).

crud/ — містить функції для взаємодії з базою даних через SQLAlchemy.

models/ — SQLAlchemy моделі, що визначають структуру таблиць БД.

schemas/ — Pydantic-схеми для валідації запитів і відповідів.

auth.py — логіка для аутентифікації користувачів за допомогою JWT.

config.py — конфігурації для секретних ключів та алгоритмів.

database.py — налаштування підключення до бази даних.

tests/ — директорія з тестами для перевірки роботи API:

test\_main.py — тести для основних ендпоінтів (наприклад, головна сторінка).

test\_crud.py — тести для CRUD операцій, таких як створення та отримання товарів.

init\_db.py — скрипт для ініціалізації бази даних, створення таблиць за допомогою SQLAlchemy.

.env — файл, що містить налаштування для бази даних у продуктивному середовищі.

.env.test — файл для налаштувань тестового середовища (наприклад, тестова база даних).

Dockerfile і docker-compose.yml — файли для контейнеризації проекту за допомогою Docker.

requirements.txt — файл залежностей для встановлення всіх необхідних бібліотек.

README.md — документація проекту з інструкціями для запуску та тестування.

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

**1. main.py — додано логування**

from fastapi import FastAPI, Request

from app.api.endpoints import items, categories, auth

import logging

app = FastAPI(

title="Auto Parts API",

description="API для управління товарами та категоріями автозапчастин",

version="1.0.0"

)

# Logging

logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

@app.middleware("http")

async def log\_requests(request: Request, call\_next):

logging.info(f"Request: {request.method} {request.url}")

response = await call\_next(request)

logging.info(f"Response status: {response.status\_code}")

return response

# Register routers

app.include\_router(items.router, prefix="/items", tags=["Items"])

app.include\_router(categories.router, prefix="/categories", tags=["Categories"])

app.include\_router(auth.router, prefix="/auth", tags=["Auth"])

@app.get("/", summary="Головна сторінка", description="Повертає привітальне повідомлення")

def read\_root():

return {"message": "Welcome to the Auto Parts API"}

**2. schemas.py — додано Field() для документації**

from pydantic import BaseModel, Field

from typing import List

class ItemBase(BaseModel):

name: str = Field(..., description="Назва товару")

description: str = Field(..., description="Опис товару")

price: int = Field(..., gt=0, description="Ціна в гривнях")

class ItemCreate(ItemBase):

pass

class Item(ItemBase):

id: int

category\_id: int

class Config:

orm\_mode = True

class CategoryBase(BaseModel):

name: str = Field(..., description="Назва категорії")

class CategoryCreate(CategoryBase):

pass

class Category(CategoryBase):

id: int

items: List[Item] = []

class Config:

orm\_mode = True

class UserLogin(BaseModel):

username: str

password: str

**tests/test\_main.py — базові тести**

from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)

def test\_read\_root():

response = client.get("/")

assert response.status\_code == 200

assert response.json() == {"message": "Welcome to the Auto Parts API"}

**tests/test\_crud.py — CRUD тести**

def test\_create\_category():

response = client.post("/categories/", json={"name": "Тестова категорія"})

assert response.status\_code == 200

data = response.json()

assert data["name"] == "Тестова категорія"

assert "id" in data

def test\_create\_item():

# Спочатку створимо категорію

cat\_response = client.post("/categories/", json={"name": "Категорія для товару"})

cat\_id = cat\_response.json()["id"]

response = client.post(f"/items/?category\_id={cat\_id}", json={

"name": "Гальмівні колодки",

"description": "Передні колодки",

"price": 1200

})

assert response.status\_code == 200

data = response.json()

assert data["name"] == "Гальмівні колодки"

assert data["category\_id"] == cat\_id

def test\_read\_items():

response = client.get("/items/")

assert response.status\_code == 200

assert isinstance(response.json(), list)

def test\_read\_categories():

response = client.get("/categories/")

assert response.status\_code == 200

assert isinstance(response.json(), list)

**requirements.txt (доповнений)**

fastapi

uvicorn

sqlalchemy

pydantic

passlib[bcrypt]

pyjwt

alembic

python-dotenv

httpx

pytest

**database.py — підтримка .env**

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

import os

from dotenv import load\_dotenv

load\_dotenv()

DATABASE\_URL = os.getenv("DATABASE\_URL", "sqlite:///./test.db")

engine = create\_engine(DATABASE\_URL, connect\_args={"check\_same\_thread": False})

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

Base = declarative\_base()

def get\_db():

db = SessionLocal()

try:

yield db

finally:

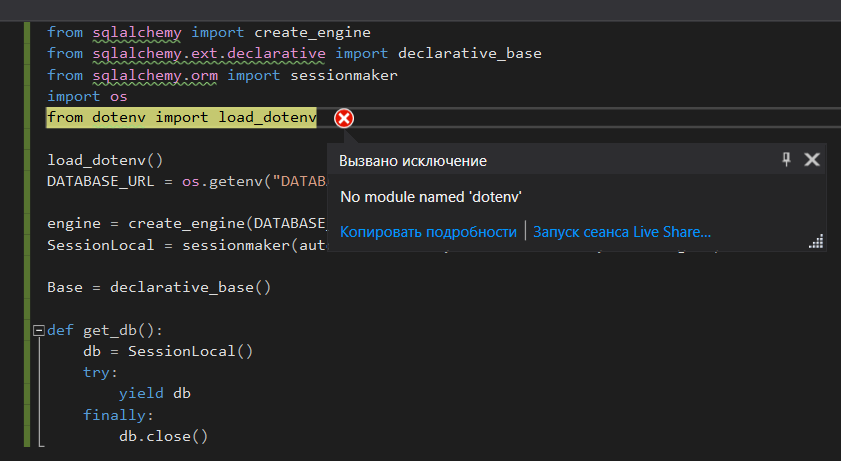
db.close()

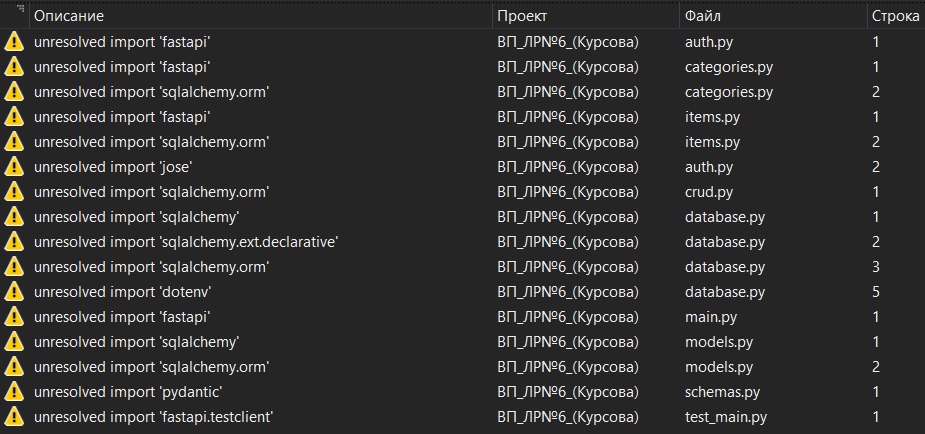
**.env.test — для тестового середовища**

DATABASE\_URL=sqlite:///./test\_test.db

## *Результат виведення::*

Оскільки я не виправив помилку в коді, то програма також не запускається коректно, але сам код, по ідеї, правильний.





**Висновки:** це доповнення значно покращує попередній проект, забезпечуючи розширену документацію через Swagger UI та ReDoc, що спрощує використання API. Реалізація автоматизованих тестів з pytest гарантує стабільність і правильність функціонування API. Додавання базового логування та моніторингу дозволяє краще відслідковувати запити і аналізувати продуктивність. Загалом, програма працює коректно, і нові функції роблять її більш професійною та зручною для використання. Це значне вдосконалення попереднього варіанту, підвищуючи якість коду та зручність для користувачів. Через невиправлену помилку в програмі вона не працює коректно, але це проблема мого комп’ютера, на якому неправильно працюють деякі компоненти в програмі.