

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ  
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

Кафедра Обчислювальної техніки Факультет Інформатики та обчислювальної  
техніки

**Звіт**

до лабораторної роботи №2

з дисципліни

«Інтелектуальні вбудовані системи»

Виконав:

Студент IV курсу групи ІІ-01 Черпак А.В.

Перевірив:

Нікольський С.С.

Оцінка: \_\_\_\_\_ Дата: \_\_\_\_\_

Київ 2024

**Завдання:** Для зберігання та доступу до даних потрібно реалізувати Store арі, який буде зберігати проаналізовані дані в базу даних. Також потрібно реалізувати спосіб отримання нових даних для UI клієнтів..

## Виконання

Для початку створимо проект згідно зі вказівками, наведеними у методичці. Отриманий проект матиме структуру, наведену на рисунку 1.

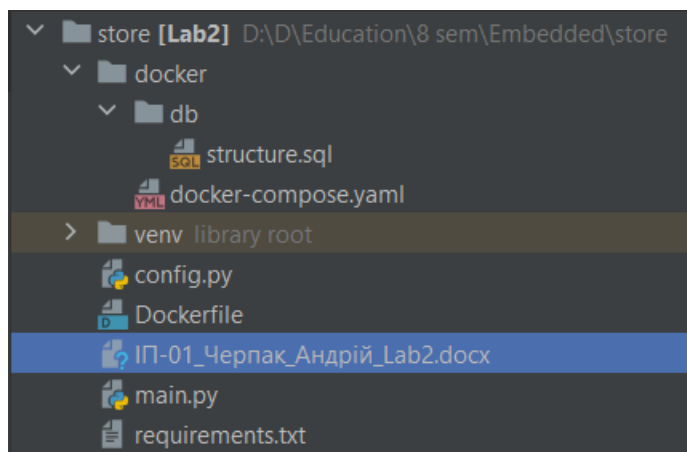


Рисунок 1 – Структура новоствореного проекту

Початкове наповнення файлів повністю відповідатиме коду, наведеному у методичці.

structure.sql

```
CREATE TABLE processed_agent_data (
    id SERIAL PRIMARY KEY,
    road_state VARCHAR(255) NOT NULL,
    user_id INTEGER NOT NULL,
    x FLOAT,
    y FLOAT,
    z FLOAT,
    latitude FLOAT,
    longitude FLOAT,
    timestamp TIMESTAMP
);
```

docker-compose.yaml

```
version: "3.9"
name: "road_vision__database"
services:
  postgres_db:
    image: postgres:latest
    container_name: postgres_db
    restart: always
    environment:
      POSTGRES_USER: user
```

```
    POSTGRES_PASSWORD: pass
    POSTGRES_DB: test_db
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./db/structure.sql:/docker-entrypoint-
initdb.d/structure.sql
  ports:
    - "5432:5432"
  networks:
    db_network:
```

```
pgadmin:
  container_name: pgadmin4
  image: dpage/pgadmin4
  restart: always
  environment:
    PGADMIN_DEFAULT_EMAIL: admin@admin.com
    PGADMIN_DEFAULT_PASSWORD: root
  volumes:
    - pgadmin-data:/var/lib/pgadmin
  ports:
    - "5050:80"
  networks:
    db_network:
```

```
store:
  container_name: store
  build: ..
  depends_on:
    - postgres_db
  restart: always
  environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: pass
    POSTGRES_DB: test_db
    POSTGRES_HOST: postgres_db
    POSTGRES_PORT: 5432
  ports:
    - "8000:8000"
  networks:
    db_network:
```

```
networks:
  db_network:
```

```
volumes:
  postgres_data:
  pgadmin-data:
```

config.py

```
import os

def try_parse(expected_type: type, value: str):
    try:
        return expected_type(value)
    except Exception:
        return None

# Configuration for POSTGRES
POSTGRES_HOST = os.environ.get("POSTGRES_HOST") or
"localhost"
POSTGRES_PORT = try_parse(int,
os.environ.get("POSTGRES_PORT")) or 5432
POSTGRES_USER = os.environ.get("POSTGRES_USER") or "user"
POSTGRES_PASSWORD = os.environ.get("POSTGRES_PASS") or
"pass"
POSTGRES_DB = os.environ.get("POSTGRES_DB") or "test_db"
```

main.py

```
import asyncio
import json
from typing import Set, Dict, List, Any
from fastapi import FastAPI, HTTPException, WebSocket,
WebSocketDisconnect, Body
from sqlalchemy import (
    create_engine,
    MetaData,
    Table,
    Column,
    Integer,
    String,
    Float,
    DateTime,
)
from sqlalchemy.orm import sessionmaker
from sqlalchemy.sql import select
from datetime import datetime
from pydantic import BaseModel, field_validator
```

```

from config import (
    POSTGRES_HOST,
    POSTGRES_PORT,
    POSTGRES_DB,
    POSTGRES_USER,
    POSTGRES_PASSWORD,
)

# FastAPI app setup
app = FastAPI()
# SQLAlchemy setup
DATABASE_URL =
f"postgresql+psycopg2://{POSTGRES_USER}:{POSTGRES_PASSWORD}@{POSTGRES_HOST}:{POSTGRES_PORT}/{POSTGRES_DB}"
engine = create_engine(DATABASE_URL)
metadata = MetaData()
# Define the ProcessedAgentData table
processed_agent_data = Table(
    "processed_agent_data",
    metadata,
    Column("id", Integer, primary_key=True, index=True),
    Column("road_state", String),
    Column("user_id", Integer),
    Column("x", Float),
    Column("y", Float),
    Column("z", Float),
    Column("latitude", Float),
    Column("longitude", Float),
    Column("timestamp", DateTime),
)
SessionLocal = sessionmaker(bind=engine)

# SQLAlchemy model
class ProcessedAgentDataInDB(BaseModel):
    id: int
    road_state: str
    user_id: int
    x: float
    y: float
    z: float
    latitude: float
    longitude: float
    timestamp: datetime

# FastAPI models

```

```

class AccelerometerData(BaseModel):
    x: float
    y: float
    z: float

class GpsData(BaseModel):
    latitude: float
    longitude: float

class AgentData(BaseModel):
    user_id: int
    accelerometer: AccelerometerData
    gps: GpsData
    timestamp: datetime

    @classmethod
    @field_validator("timestamp", mode="before")
    def check_timestamp(cls, value):
        if isinstance(value, datetime):
            return value
        try:
            return datetime.fromisoformat(value)
        except (TypeError, ValueError):
            raise ValueError(
                "Invalid timestamp format. Expected ISO
8601 format (YYYY-MM-DDTHH:MM:SSZ)."
            )

class ProcessedAgentData(BaseModel):
    road_state: str
    agent_data: AgentData

# WebSocket subscriptions
subscriptions: Dict[int, Set[WebSocket]] = {}

# FastAPI WebSocket endpoint
@app.websocket("/ws/{user_id}")
async def websocket_endpoint(websocket: WebSocket,
user_id: int):
    await websocket.accept()
    if user_id not in subscriptions:
        subscriptions[user_id] = set()

```

```

        subscriptions[user_id].add(websocket)
    try:
        while True:
            await websocket.receive_text()
    except WebSocketDisconnect:
        subscriptions[user_id].remove(websocket)

# Function to send data to subscribed users
async def send_data_to_subscribers(user_id: int, data):
    if user_id in subscriptions:
        for websocket in subscriptions[user_id]:
            await websocket.send_json(json.dumps(data))

# FastAPI CRUDL endpoints

@app.post("/processed_agent_data/")
async def create_processed_agent_data(data:
List[ProcessedAgentData]):
    # Insert data to database
    # Send data to subscribers
    pass

@app.get(
    "/processed_agent_data/{processed_agent_data_id}",
    response_model=ProcessedAgentDataInDB,
)
def read_processed_agent_data(processed_agent_data_id:
int):
    # Get data by id
    pass

@app.get("/processed_agent_data/",
response_model=list[ProcessedAgentDataInDB])
def list_processed_agent_data():
    # Get list of data
    pass

@app.put(
    "/processed_agent_data/{processed_agent_data_id}",
    response_model=ProcessedAgentDataInDB,
)

```

```

def update_processed_agent_data(processed_agent_data_id:
int, data: ProcessedAgentData):
    # Update data
    pass

@app.delete(
    "/processed_agent_data/{processed_agent_data_id}",
    response_model=ProcessedAgentDataInDB,
)
def delete_processed_agent_data(processed_agent_data_id:
int):
    # Delete by id
    pass

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="127.0.0.1", port=8000)

```

#### Dockerfile

```

# Use the official Python image as the base image
FROM python:latest
# Set the working directory inside the container
WORKDIR /app
# Copy the requirements.txt file and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Copy the entire application into the container
COPY . .
# Run the main.py script inside the container when it
starts
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]

```

#### requirements.txt

```

annotated-types==0.6.0
anyio==4.3.0
click==8.1.7
colorama==0.4.6
fastapi==0.110.0
greenlet==3.0.3
h11==0.14.0
httpptools==0.6.1
idna==3.6
psycpg2==2.9.9
pydantic==2.6.2

```



```
pydantic_core==2.16.3
python-dotenv==1.0.1
PyYAML==6.0.1
sniffio==1.3.1
SQLAlchemy==2.0.27
starlette==0.36.3
typing_extensions==4.10.0
uvicorn==0.27.1
watchfiles==0.21.0
websockets==12.0
```

Після цього необхідно було реалізувати основну логіку читання та запису даних у БД. Для цього спершу розділимо файл `main.py` на декілька з більш вузькою спеціалізацією, потім спростимо створення таблиць (зараз це робиться трішки складно), а тоді реалізуємо саму логіку заповнення БД та отримання даних з неї.

У першу чергу, створимо файл `models.py` та винесемо туди визначення моделей FastAPI.

`models.py`:

```
from datetime import datetime
from pydantic import field_validator, BaseModel

# FastAPI models
class AccelerometerData(BaseModel):
    x: float
    y: float
    z: float

class GpsData(BaseModel):
    latitude: float
    longitude: float

class AgentData(BaseModel):
    user_id: int
    accelerometer: AccelerometerData
    gps: GpsData
    timestamp: datetime

    @classmethod
    @field_validator("timestamp", mode="before")
    def check_timestamp(cls, value):
        if isinstance(value, datetime):
```

```

        return value
    try:
        return datetime.fromisoformat(value)
    except (TypeError, ValueError):
        raise ValueError(
            "Invalid timestamp format. Expected ISO
8601 format (YYYY-MM-DDTHH:MM:SSZ)."
        )

class ProcessedAgentData(BaseModel):
    road_state: str
    agent_data: AgentData

```

Наступним кроком буде створення файлу database.py та винесення туди ініціалізації БД та таблиць, а також визначення сутностей.

database.py:

```

from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String, Float,
DateTime
from sqlalchemy.orm import sessionmaker, declarative_base
from pydantic import BaseModel

from config import DATABASE_URL

Base = declarative_base()
# SQLAlchemy setup
engine = create_engine(DATABASE_URL)

SessionLocal = sessionmaker(bind=engine)
session = SessionLocal()

# SQLAlchemy model
class ProcessedAgentDataInDB(Base):
    __tablename__ = "processed_agent_data"

    id = Column(Integer, primary_key=True, index=True,
autoincrement=True)
    road_state = Column(String)
    user_id = Column(Integer)
    x = Column(Float)
    y = Column(Float)
    z = Column(Float)
    latitude = Column(Float)

```

```
longitude = Column(Float)
timestamp = Column(DateTime)
```

```
Base.metadata.create_all(engine)
```

Тепер створимо окремий роутер для обробки запитів на `/processed_agent_data/` та винесемо його у файл `processed_agent_data_routes.py`. Відповідно, у файлі `main.py` залишиться лише логіка створення додатку, підключення роутера та обробка веб-сокетних з'єднань. Наведемо код описаних файлів:

`main.py`:

```
import json
from typing import Set, Dict
from fastapi import FastAPI, HTTPException, WebSocket,
WebSocketDisconnect, Body
from processed_agent_data_routes import
processed_agent_data_router

# FastAPI app setup
app = FastAPI()
app.include_router(processed_agent_data_router,
prefix='/processed_agent_data')
# WebSocket subscriptions
subscriptions: Dict[int, Set[WebSocket]] = {}

# FastAPI WebSocket endpoint
@app.websocket("/ws/{user_id}")
async def websocket_endpoint(websocket: WebSocket,
user_id: int):
    await websocket.accept()
    if user_id not in subscriptions:
        subscriptions[user_id] = set()
    subscriptions[user_id].add(websocket)
    try:
        while True:
            await websocket.receive_text()
    except WebSocketDisconnect:
        subscriptions[user_id].remove(websocket)

# Function to send data to subscribed users
async def send_data_to_subscribers(user_id: int, data):
    if user_id in subscriptions:
```

```

        for websocket in subscriptions[user_id]:
            await websocket.send_json(json.dumps(data))

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="127.0.0.1", port=8000)

```

processed\_agent\_data\_routes.py

```

from typing import List

from fastapi import APIRouter

from models import ProcessedAgentData
from database import ProcessedAgentDataInDB, session

processed_agent_data_router = APIRouter()

@processed_agent_data_router.post("/")
async def create_processed_agent_data(data:
List[ProcessedAgentData]):
    session.bulk_save_objects(
        [ProcessedAgentDataInDB(
            road_state=a_data.road_state,
            user_id=a_data.agent_data.user_id,
            x=a_data.agent_data.accelerometer.x,
            y=a_data.agent_data.accelerometer.y,
            z=a_data.agent_data.accelerometer.z,
            latitude=a_data.agent_data.gps.latitude,
            longitude=a_data.agent_data.gps.longitude,
            timestamp=a_data.agent_data.timestamp
        ) for a_data in data]
    )
    session.commit()
    return

@processed_agent_data_router.get("/{processed_agent_data_
id}")
def read_processed_agent_data(processed_agent_data_id:
int):
    return
session.query(ProcessedAgentDataInDB).get(processed_agent
_data_id)

```

```

@processed_agent_data_router.get("/")
def list_processed_agent_data():
    return
list(session.query(ProcessedAgentDataInDB).all())

@processed_agent_data_router.put("/{processed_agent_data_
id}")
def update_processed_agent_data(processed_agent_data_id:
int, data: ProcessedAgentData):
    updated_instance = ProcessedAgentDataInDB(
        road_state=data.road_state,
        user_id=data.agent_data.user_id,
        x=data.agent_data.accelerometer.x,
        y=data.agent_data.accelerometer.y,
        z=data.agent_data.accelerometer.z,
        latitude=data.agent_data.gps.latitude,
        longitude=data.agent_data.gps.longitude,
        timestamp=data.agent_data.timestamp,
        id=processed_agent_data_id
    )

    session.merge(updated_instance)
    session.commit()
    return updated_instance

@processed_agent_data_router.delete("/{processed_agent_da
ta_id}")
def delete_processed_agent_data(processed_agent_data_id:
int):
    obj =
session.query(ProcessedAgentDataInDB).get(processed_agent
_data_id)
    session.delete(obj)
    session.commit()
    return obj

@processed_agent_data_router.delete("/")
def clear_all_data():
    data = session.query(ProcessedAgentDataInDB)
    lines_count = data.count()
    data.delete()

```

```
session.commit()
return {"lines_deleted": lines_count}
```

Форматування коду у word просто жахливе, тому краще відслідковувати зміни по комітам у репозиторії:

<https://github.com/CherpakAndrii/IntellectualEmbeddedSystems/tree/master/store>

На рисунку 2 зображено оновлену структуру проекту:

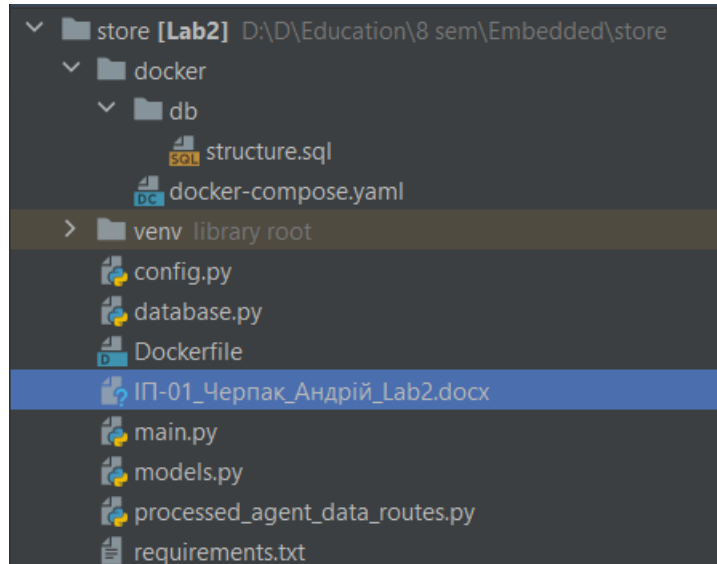


Рисунок 2 – Структура оновленого проекту

На цьому етапі лабораторну роботу можна вважати виконаною. Підніmemo докер-контекнер та перевіримо результати. Команди для розгортання контейнеру зображено на рисунку 3.

```
(venv) PS D:\D\Education\8 sem\Embedded\store> cd docker
(venv) PS D:\D\Education\8 sem\Embedded\store\docker> docker-compose up --build
[+] Building 6.4s (10/10) FINISHED
=> [store internal] load .dockerignore
=> => transferring context: 2B
=> [store internal] load build definition from Dockerfile
=> => transferring dockerfile: 482B
=> [store internal] load metadata for docker.io/library/python:latest
=> [store 1/5] FROM docker.io/library/python:latest@sha256:e83d1f4d0c735c7a54fc9dae3cca8c58473e3b3de08fcb7ba3d342ee75cfc09d
=> [store internal] load build context
=> => transferring context: 902.44kB
=> CACHED [store 2/5] WORKDIR /app
=> => exporting layers
=> => writing image sha256:97010f2353d0085f095893d433da680edf8910bd12b5e6bb75c0b71723ec307a
=> => naming to docker.io/library/road_vision__database-store
[+] Running 4/4
✓ Network road_vision__database_db_network Created
✓ Container pgadmin4 Created
✓ Container postgres_db Created
✓ Container store Created
Attaching to pgadmin4, postgres_db, store
postgres_db |
postgres_db | PostgreSQL Database directory appears to contain a database; Skipping initialization
postgres_db |
postgres_db | 2024-02-28 20:59:02.559 UTC [1] LOG: starting PostgreSQL 16.2 (Debian 16.2-1.pgdg120+2) on x86_64-pc-linux-gnu,
postgres_db | 2024-02-28 20:59:02.560 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
postgres_db | 2024-02-28 20:59:02.560 UTC [1] LOG: listening on IPv6 address ":::", port 5432
postgres_db | 2024-02-28 20:59:02.579 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
postgres_db | 2024-02-28 20:59:02.619 UTC [29] LOG: database system was shut down at 2024-02-28 20:57:45 UTC
```

```

postgres_db | 2024-02-28 20:59:02.649 UTC [1] LOG:  database system is ready to accept connections
pgadmin4    | postfix/postlog: starting the Postfix mail system
store       | INFO:      Started server process [1]
store       | INFO:      Waiting for application startup.
store       | INFO:      Application startup complete.
store       | INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
pgadmin4    | [2024-02-28 20:59:14 +0000] [1] [INFO] Starting gunicorn 20.1.0
pgadmin4    | [2024-02-28 20:59:14 +0000] [1] [INFO] Listening at: http://[::]:80 (1)
pgadmin4    | [2024-02-28 20:59:14 +0000] [1] [INFO] Using worker: gthread
pgadmin4    | [2024-02-28 20:59:14 +0000] [87] [INFO] Booting worker with pid: 87

```

Рисунок 3 – Підняття docker-контейнера з консолі

Тепер перейдемо за посиланням <http://127.0.0.1:8000/docs> та переглянемо автоматично згенеровану з допомогою SwaggerUI документацію проекту:

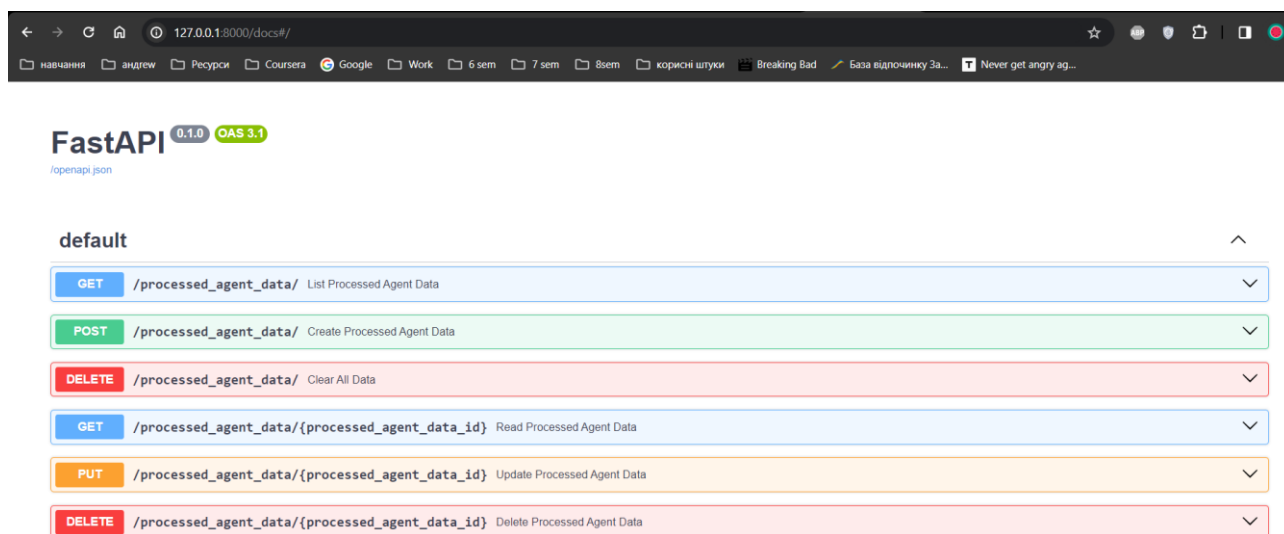


Рисунок 4 – SwaggerUI документація проекту

Так само перейдемо за посиланням <http://127.0.0.1:5050/browser/> та переглянемо стан нашої БД у PgAdmin, попередньо увійшовши в акаунт та під'єднавшись до БД:

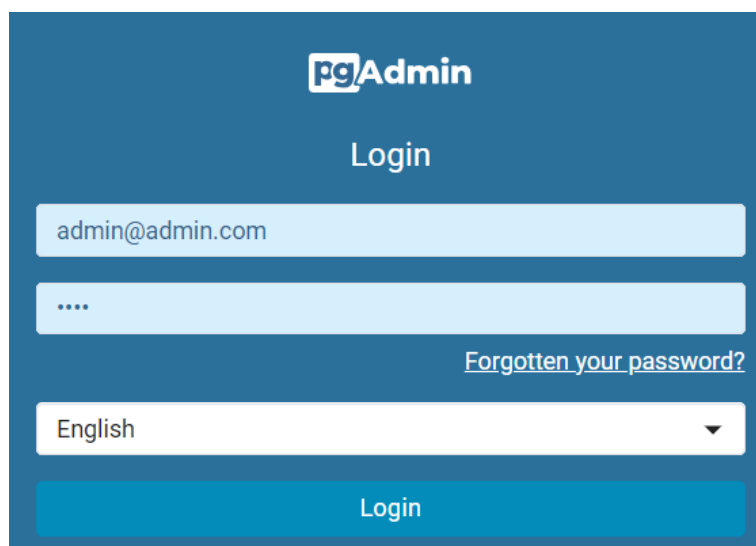


Рисунок 5 – Вхід у PgAdmin

**Register - Server**

General Connection Parameters SSH Tunnel Advanced

Name: test\_db

Server group: Servers

Рисунок 6 – Під'єднання до бази даних у PgAdmin

**Register - Server**

General **Connection** Parameters SSH Tunnel Advanced

Host name/address: postgres\_db

Port: 5432

Maintenance database: postgres

Username: user

Kerberos authentication? ☐

Password: ....

Save password? ☒

Рисунок 7 – Під'єднання до бази даних у PgAdmin - продовження

**PgAdmin** File Object Tools Help

Object Explorer

- FTS Configurations
- FTS Dictionaries
- FTS Parsers
- FTS Templates
- Foreign Tables
- Functions
- Materialized Views
- Operators
- Procedures
- Sequences
- Tables (1)
  - processed\_agent\_data
    - Columns
    - Constraints
    - Indexes
    - RLS Policies
    - Rules
    - Triggers
    - Trigger Functions
    - Types

Query Editor: public.processed\_agent\_data/test\_db/user@test\_db

```

1 SELECT * FROM public.processed_agent_data
2 ORDER BY id ASC LIMIT 100
3

```

Data Output

id	road_state	user_id	x	y
[PK] integer	character varying (255)	integer	double precision	double p

Рисунок 8 – Відслідковування стану бази даних у PgAdmin



Тепер протестуємо систему за алгоритмом, наведеним у методичці, та покроково відобразимо результати.

Для початку, додамо запис, заповнений одиницями в усіх полях:

POST /processed\_agent\_data/ Create Processed Agent Data

Parameters: No parameters

Request body required: application/json

```
{
  "road_state": "string",
  "agent_data": {
    "user_id": 1,
    "accelerometer": {
      "x": 1,
      "y": 1,
      "z": 1
    },
    "gps": {
      "latitude": 1,
      "longitude": 1
    },
    "timestamp": "2024-02-28T19:10:58.337Z"
  }
}
```

Execute Clear

Рисунок 9 – Додавання нового запису у БД

Пересвідчимося, що запис додано:

public.processed\_agent\_data/test\_db/user@test\_db x

public.processed\_agent\_data/test\_db/user@test\_db

Query: 1 SELECT \* FROM public.processed\_agent\_data; 2 ORDER BY id ASC LIMIT 100; 3

Data Output: Messages: Notifications:

id	road_state	user_id	x	y	z	latitude	longitude	timestamp
[PK] integer	character varying (255)	integer	double precision	double precision	double precision	double precision	double precision	timestamp without time zone
1	1	string	1	1	1	1	1	2024-02-28 19:10:58.337

Рисунок 10 – Перегляд оновленого стану бази даних у PgAdmin

Додамо ще два записи, заповнивши поля двійками та трійками відповідно. Після чого знову переглянемо стан таблиці БД з допомогою PgAdmin.

Data Output: Messages: Notifications:

id	road_state	user_id	x	y	z	latitude	longitude	timestamp
[PK] integer	character varying (255)	integer	double precision	double precision	double precision	double precision	double precision	timestamp without time zone
1	1	string	1	1	1	1	1	2024-02-28 19:10:58.337
2	2	Perfect	2	2	2	2	2	2024-02-28 19:10:58.337
3	3	Awesome	3	3	3	3	3	2024-02-28 19:10:58.337

Рисунок 11 – Стан бази даних після додавання трьох записів

Спробуємо отримати список усіх ProcessedAgentData:

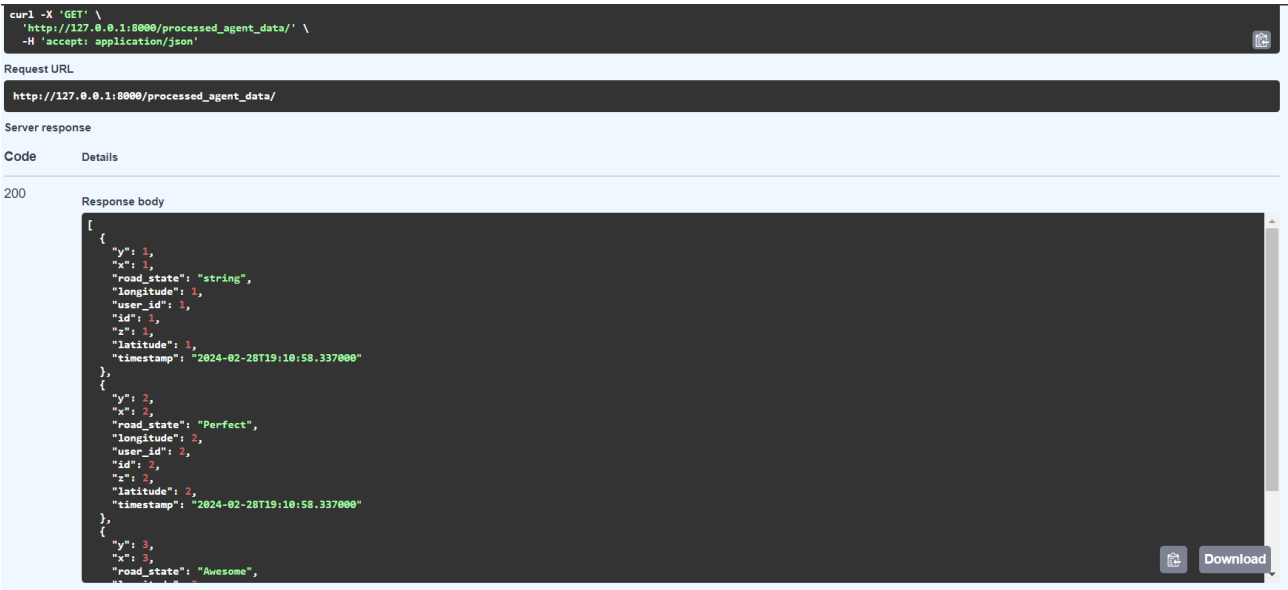


Рисунок 12 – Результат отримання усіх записів

Тепер видалимо другий запис та ще раз переглянемо вміст БД:

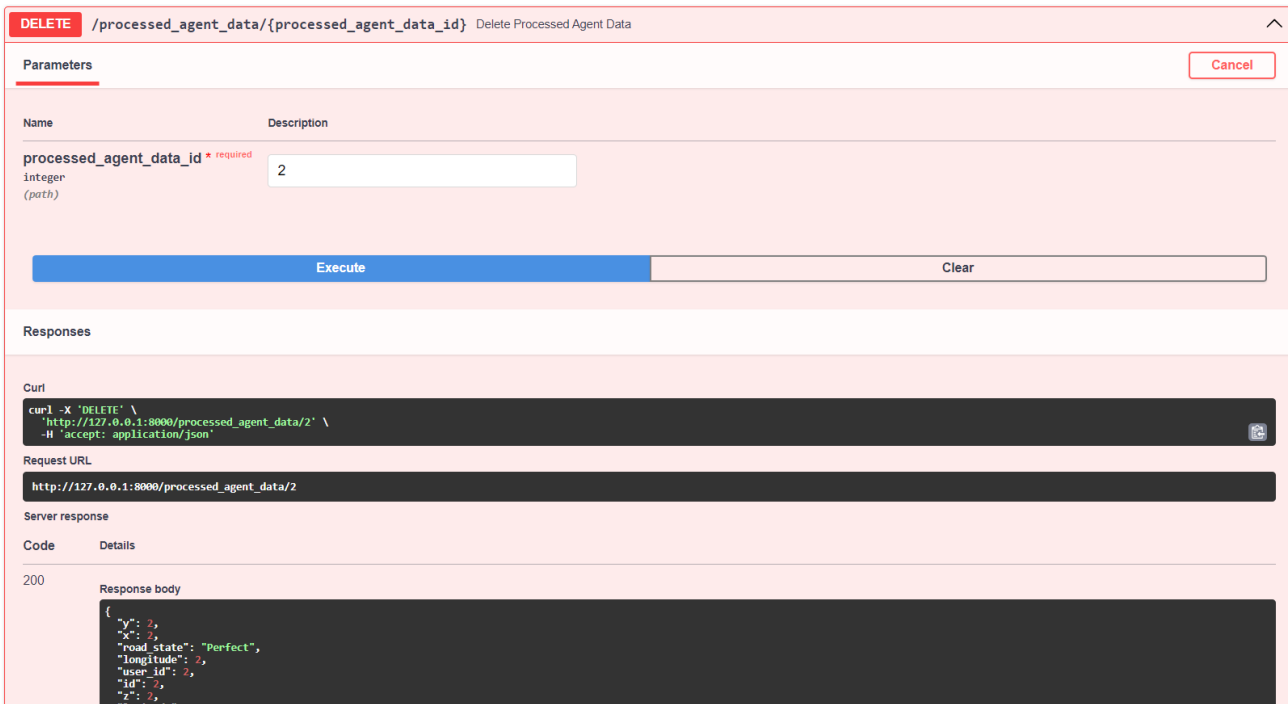


Рисунок 13 – Результат видалення запису з id 2

	id [PK] integer	road_state character varying (255)	user_id integer	x double precision	y double precision	z double precision	latitude double precision	longitude double precision	timestamp timestamp without time zone
1	1	string	1	1	1	1	1	1	2024-02-28 19:10:58.337
2	3	Awesome	3	3	3	3	3	3	2024-02-28 19:10:58.337

Рисунок 14 – Стан таблиці після видалення запису з id 2

Знову отримаємо список усіх записів з допомогою SwaggerUI:



Рисунок 15 – Результат отримання усіх поточних записів

Лишилося лиш оновити один із записів у БД і пересвідчитися, що і цей функціонал відпрацьовує коректно:

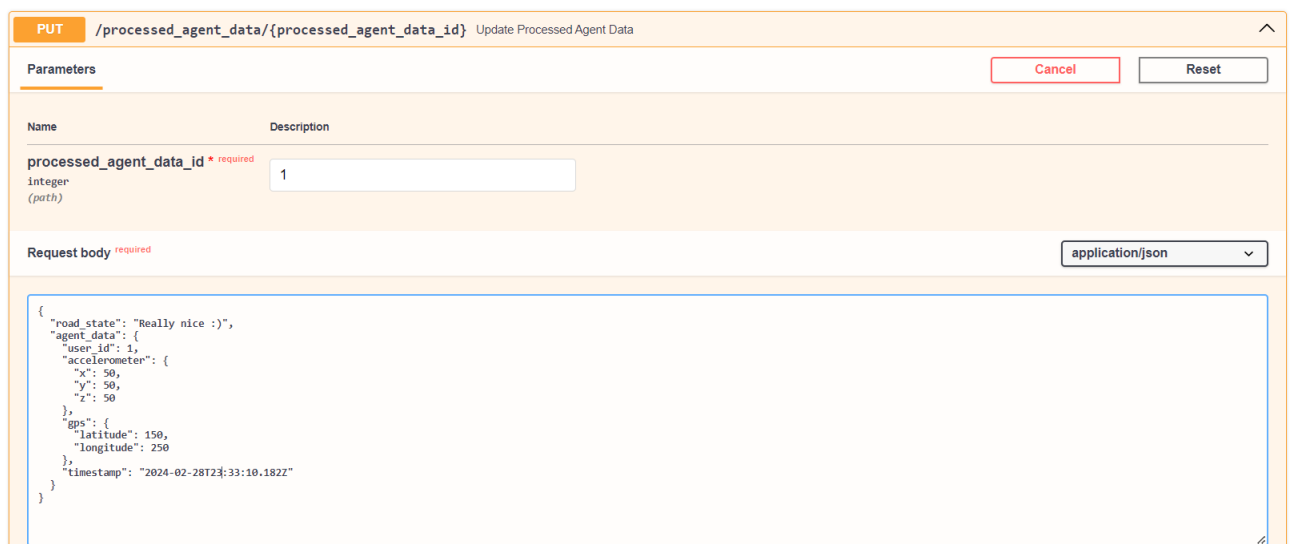
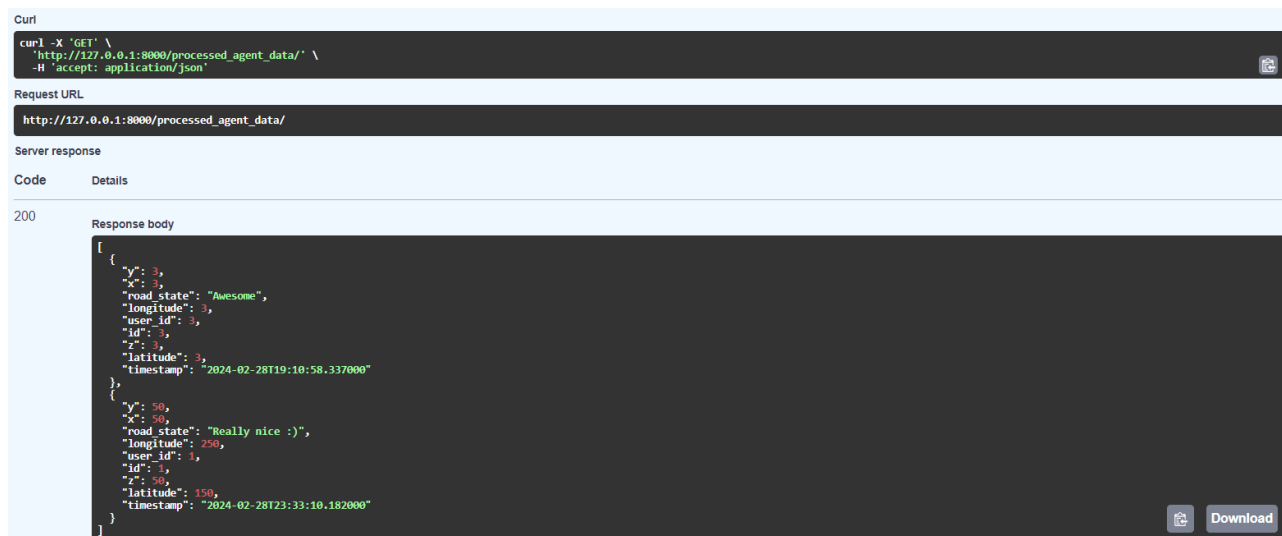


Рисунок 16 – Оновлення запису з використанням SwaggerUI



Рисунок 17 – Результат оновлення запису

Ну і, звісно ж, пересвідчимося, що дані дійсно змінено:



```
Curl
curl -X 'GET' \
'http://127.0.0.1:8000/processed_agent_data/' \
-H 'accept: application/json'

Request URL
http://127.0.0.1:8000/processed_agent_data/

Server response
Code    Details
200
Response body
[
  {
    "y": 3,
    "x": 3,
    "road_state": "Awesome",
    "longitude": 3,
    "user_id": 3,
    "id": 3,
    "z": 3,
    "latitude": 3,
    "timestamp": "2024-02-28T19:10:58.337000"
  },
  {
    "y": 50,
    "x": 50,
    "road_state": "Really nice :)",
    "longitude": 250,
    "user_id": 1,
    "id": 1,
    "z": 50,
    "latitude": 150,
    "timestamp": "2024-02-28T23:33:10.182000"
  }
]
```

Рисунок 18 – Результат отримання усіх записів після оновлення.

Як бачимо, дані дійсно оновилися. Все, тепер лабораторна робота остаточно виконана :)

**Висновок:** під час виконання комп'ютерного практикуму я розібрався з наданою кодовою базою, реалізував логіку роботи з базою даних, а саме її наповнення, отримання даних, їх модифікацію та видалення, а потім розгорнув необхідні сервіси у Docker та протестував коректність роботи. Весь функціонал відпрацьовував рівно як і очікувалося.