

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

Кафедра Обчислювальної техніки Факультет Інформатики та обчислювальної
техніки

Звіт

до лабораторної роботи №4

з дисципліни

«Інтелектуальні вбудовані системи»

Виконав:

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

Перевірив:

Нікольський С.С.

Оцінка: _____ Дата: _____

Київ 2024

Завдання: Потрібно реалізувати Edge Data Logic. А саме, дана частина займається аналізом даних з датчиків на машині. Потрібно реалізувати логіку збору даних з MQTT, аналізу стану дорожнього покриття, та відправки проаналізованих даних на Hub. Наприклад за умови, якщо дані по осі Y більше за певне значення, то тоді в нас є яма.

Виконання

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

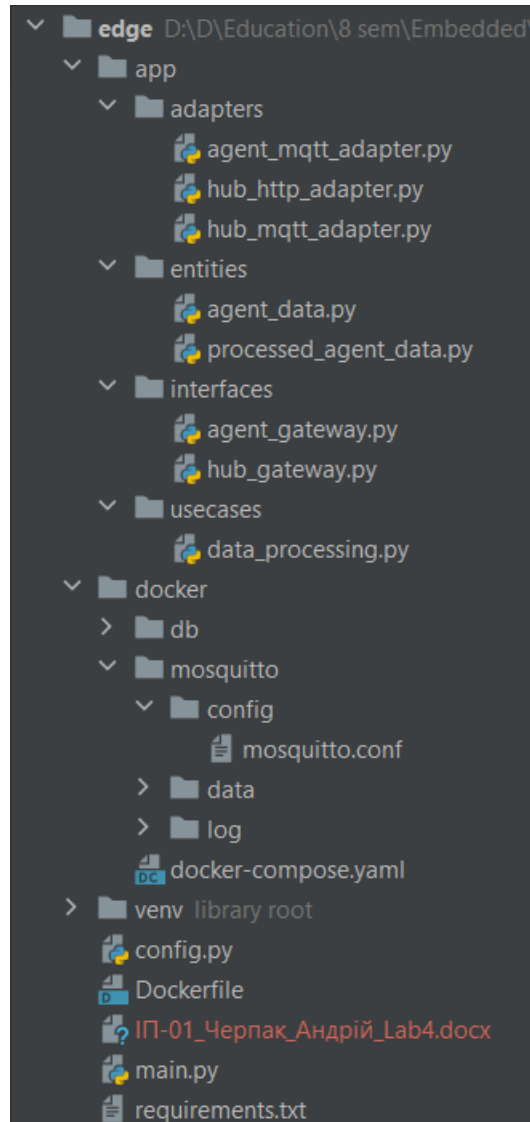


Рисунок 1 – Структура новоствореного проекту

Початкове наповнення файлів повністю відповідатиме коду, наведеному у методичці.

main.py

```
import logging
from app.adapters.agent_mqtt_adapter import AgentMQTTAdapter
```

```

from app.adapters.hub_http_adapter import HubHttpAdapter
from app.adapters.hub_mqtt_adapter import HubMqttAdapter
from config import (
    MQTT_BROKER_HOST,
    MQTT_BROKER_PORT,
    MQTT_TOPIC,
    HUB_URL,
    HUB_MQTT_BROKER_HOST,
    HUB_MQTT_BROKER_PORT,
    HUB_MQTT_TOPIC,
)

if __name__ == "__main__":
    # Configure logging settings
    logging.basicConfig(
        level=logging.INFO, # Set the log level to INFO
        # (you can use logging.DEBUG for more detailed logs)
        format="[% (asctime)s] [% (levelname)s]
[% (module)s] [% (message)s]",
        handlers=[
            logging.StreamHandler(), # Output log
messages to the console
            logging.FileHandler("app.log"), # Save log
messages to a file
        ],
    )
    # Create an instance of the StoreApiAdapter using the
configuration
    # hub_adapter = HubHttpAdapter(
    #     api_base_url=HUB_URL,
    # )
    hub_adapter = HubMqttAdapter(
        broker=HUB_MQTT_BROKER_HOST,
        port=HUB_MQTT_BROKER_PORT,
        topic=HUB_MQTT_TOPIC,
    )
    # Create an instance of the AgentMQTTAdapter using
the configuration
    agent_adapter = AgentMQTTAdapter(
        broker_host=MQTT_BROKER_HOST,
        broker_port=MQTT_BROKER_PORT,
        topic=MQTT_TOPIC,
        hub_gateway=hub_adapter,
    )
    try:
        # Connect to the MQTT broker and start listening
for messages

```

```

        agent_adapter.connect()
        agent_adapter.start()
        # Keep the system running indefinitely (you can
add other logic as needed)
        while True:
            pass
    except KeyboardInterrupt:
        # Stop the MQTT adapter and exit gracefully if
interrupted by the user
        agent_adapter.stop()
        logging.info("System stopped.")

```

config.py

```

import os

def try_parse_int(value: str):
    try:
        return int(value)
    except Exception:
        return None

# Configuration for agent MQTT
MQTT_BROKER_HOST = os.environ.get("MQTT_BROKER_HOST") or
"localhost"
MQTT_BROKER_PORT =
try_parse_int(os.environ.get("MQTT_BROKER_PORT")) or 1883
MQTT_TOPIC = os.environ.get("MQTT_TOPIC") or
"agent_data_topic"

# Configuration for hub MQTT
HUB_MQTT_BROKER_HOST =
os.environ.get("HUB_MQTT_BROKER_HOST") or "localhost"
HUB_MQTT_BROKER_PORT =
try_parse_int(os.environ.get("HUB_MQTT_BROKER_PORT")) or
1883
HUB_MQTT_TOPIC = os.environ.get("HUB_MQTT_TOPIC") or
"processed_agent_data_topic"

# Configuration for the Hub
HUB_HOST = os.environ.get("HUB_HOST") or "localhost"
HUB_PORT = try_parse_int(os.environ.get("HUB_PORT")) or
12000
HUB_URL = f"http://{HUB_HOST}:{HUB_PORT}"

```

agent_gateway.py

```

@abstractmethod
def on_message(self, client, userdata, msg):
    """
    Method to handle incoming messages from the agent.
    Parameters:
        client: MQTT client instance.
        userdata: Any additional user data passed to the
MQTT client.
        msg: The MQTT message received from the agent.
    """
    pass

@abstractmethod
def connect(self):
    """
    Method to establish a connection to the agent.
    """
    pass

@abstractmethod
def start(self):
    """
    Method to start listening for messages from the
agent.
    """
    pass

@abstractmethod
def stop(self):
    """
    Method to stop the agent gateway and clean up
resources.
    """
    pass

```

hub_gateway.py

```

from abc import ABC, abstractmethod
from app.entities.processed_agent_data import
ProcessedAgentData

class HubGateway(ABC):
    """
    Abstract class representing the Store Gateway
interface.
    All store gateway adapters must implement these

```

```

methods.
    """

    @abstractmethod
    def save_data(self, processed_data:
ProcessedAgentData) -> bool:
        """
        Method to save the processed agent data in the
        database.
        Parameters:
            processed_data (ProcessedAgentData): The
processed agent data to be saved.
        Returns:
            bool: True if the data is successfully saved,
False otherwise.
        """
        pass

```

data_processing.py

```

from app.entities.agent_data import AgentData
from app.entities.processed_agent_data import
ProcessedAgentData

def process_agent_data(
    agent_data: AgentData,
) -> ProcessedAgentData:
    """
    Process agent data and classify the state of the road
    surface.
    Parameters:
        agent_data (AgentData): Agent data that
containing accelerometer, GPS, and timestamp.
    Returns:
        processed_data_batch (ProcessedAgentData):
Processed data containing the classified state of the
road surface and agent data.
    """

```

Dockerfile

```

# Use the official Python image as the base image
FROM python:3.9-slim
# 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 ["python", "main.py"]
```

docker-compose.yaml

```
version: "3.9"
name: "road_vision"
services:
  mqtt:
    image: eclipse-mosquitto
    container_name: mqtt
    volumes:
      - ./mosquitto:/mosquitto
      - ./mosquitto/data:/mosquitto/data
      - ./mosquitto/log:/mosquitto/log
    ports:
      - 1883:1883
      - 19001:9001
    networks:
      mqtt_network:

  edge:
    container_name: edge
    build: ../
    depends_on:
      - mqtt
    environment:
      MQTT_BROKER_HOST: "mqtt"
      MQTT_BROKER_PORT: 1883
      MQTT_TOPIC: " "
      HUB_HOST: "store"
      HUB_PORT: 8000
      HUB_MQTT_BROKER_HOST: "mqtt"
      HUB_MQTT_BROKER_PORT: 1883
      HUB_MQTT_TOPIC: "processed_data_topic"
    networks:
      mqtt_network:
      edge_hub:

networks:
  mqtt_network:
```

```
db_network:
edge_hub:
hub:
hub_store:
hub_redis:

volumes:
  postgres_data:
  pgadmin-data:

requirements.txt
```

```
annotated-types==0.6.0
certifi==2024.2.2
charset-normalizer==3.3.2
idna==3.6
paho-mqtt==1.6.1
pydantic==2.6.1
pydantic_core==2.16.2
requests==2.31.0
typing_extensions==4.9.0
urllib3==2.2.0
```

Також бачимо, що не вистачає кількох сутностей з даними, а також файлу конфігурації mosquitto, які ми використовували у минулих лабораторних.

agent_data.py

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

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 parse_timestamp(cls, value):
        # Convert the timestamp to a datetime object
        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)."
            )

```

processed_agent_data.py

```

from pydantic import BaseModel
from app.entities.agent_data import AgentData

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

```

mosquitto.conf

```

persistence true
persistence_location /mosquitto/data/
listener 1883
## Authentication ##
allow_anonymous true
# allow_anonymous false
# password_file /mosquitto/config/password.txt
## Log ##
log_dest file /mosquitto/log/mosquitto.log
log_dest stdout
# listener 1883

```

Після цього необхідно було імплементувати адаптери, а також функцію відслідковування ям.

agent_mqtt_adapter.py

```

import logging
import paho.mqtt.client as mqtt
from app.interfaces.agent_gateway import AgentGateway
from app.entities.agent_data import AgentData, GpsData
from app.usecases.data_processing import
process_agent_data
from app.interfaces.hub_gateway import HubGateway

```

```

class AgentMQTTAdapter(AgentGateway):
    def __init__(
        self,
        broker_host,
        broker_port,
        topic,
        hub_gateway: HubGateway,
        batch_size=10,
    ):
        self.batch_size = batch_size
        # MQTT
        self.broker_host = broker_host
        self.broker_port = broker_port
        self.topic = topic
        self.client = mqtt.Client()
        # Hub
        self.hub_gateway = hub_gateway

    def on_connect(self, client, userdata, flags, rc):
        if rc == 0:
            logging.info(f"Connected to MQTT broker.
Topic: {self.topic}")
            self.client.subscribe(self.topic)
        else:
            logging.info(f"Failed to connect to MQTT
broker with code: {rc}")

    def on_message(self, client, userdata, msg):
        """Processing agent data and sent it to hub
gateway"""
        try:
            print("Received raw agent data from MQTT")
            payload: str = msg.payload.decode("utf-8")
            # Create AgentData instance with the received
data
            agent_data =
AgentData.model_validate_json(payload, strict=True)
            # Process the received data (you can call a
use case here if needed)
            processed_data =
process_agent_data(agent_data)
            # Store the agent_data in the database (you
can send it to the data processing module)
            if not
self.hub_gateway.save_data(processed_data):
                logging.error("Hub is not available")

```

```

        except Exception as e:
            logging.info(f"Error processing MQTT message:
{e}")

    def connect(self):
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.connect(self.broker_host,
self.broker_port, 60)

    def start(self):
        self.client.loop_start()

    def stop(self):
        self.client.loop_stop()

# Usage example:
if __name__ == "__main__":
    broker_host = "localhost"
    broker_port = 1883
    topic = "agent_data_topic"
    # Assuming you have implemented the StoreGateway and
passed it to the adapter
    store_gateway = HubGateway()
    adapter = AgentMQTTAdapter(broker_host, broker_port,
topic, store_gateway)
    adapter.connect()
    adapter.start()
    try:
        # Keep the adapter running in the background
        while True:
            pass
    except KeyboardInterrupt:
        adapter.stop()
        logging.info("Adapter stopped.")

```

hub_http_adapter.py

```

import logging

import requests as requests

from app.entities.processed_agent_data import
ProcessedAgentData
from app.interfaces.hub_gateway import HubGateway

```

```

class HubHttpAdapter(HubGateway):
    def __init__(self, api_base_url):
        self.api_base_url = api_base_url

    def save_data(self, processed_data:
ProcessedAgentData):
        """
        Save the processed road data to the Hub.
        Parameters:
            processed_data (ProcessedAgentData):
Processed road data to be saved.
        Returns:
            bool: True if the data is successfully saved,
False otherwise.
        """
        url =
f"{self.api_base_url}/processed_agent_data/"

        response = requests.post(url,
data=processed_data.model_dump_json())
        if response.status_code != 200:
            logging.info(
                f"Invalid Hub response\nData:
{processed_data.model_dump_json()}\nResponse: {response}"
            )
            return False
        return True

```

hub_mqtt_adapter.py

```

import logging

import requests as requests
from paho.mqtt import client as mqtt_client

from app.entities.processed_agent_data import
ProcessedAgentData
from app.interfaces.hub_gateway import HubGateway

class HubMqttAdapter(HubGateway):
    def __init__(self, broker, port, topic):
        self.broker = broker
        self.port = port
        self.topic = topic
        self.mqtt_client = self._connect_mqtt(broker,
port)

```

```

    def save_data(self, processed_data:
ProcessedAgentData):
        """
        Save the processed road data to the Hub.
        Parameters:
            processed_data (ProcessedAgentData):
Processed road data to be saved.
        Returns:
            bool: True if the data is successfully saved,
False otherwise.
        """
        msg = processed_data.model_dump_json()
        result = self.mqtt_client.publish(self.topic,
msg)

        status = result[0]
        if status == 0:
            return True
        else:
            print(f"Failed to send message to topic
{self.topic}")
            return False

    @staticmethod
    def _connect_mqtt(broker, port):
        """Create MQTT client"""
        print(f"CONNECT TO {broker}:{port}")

        def on_connect(client, userdata, flags, rc):
            if rc == 0:
                print(f"Connected to MQTT Broker
({broker}:{port})!")
            else:
                print("Failed to connect {broker}:{port},
return code %d\n", rc)
                exit(rc) # Stop execution

        client = mqtt_client.Client()
        client.on_connect = on_connect
        client.connect(broker, port)
        client.loop_start()
        return client

```

data_processing.py

```

from app.entities.agent_data import AgentData
from app.entities.processed_agent_data import
ProcessedAgentData

```

```

def process_agent_data(
    agent_data: AgentData,
) -> ProcessedAgentData:
    """
        Process agent data and classify the state of the road
        surface.
        Parameters:
            agent_data (AgentData): Agent data that
            containing accelerometer, GPS, and timestamp.
        Returns:
            processed_data_batch (ProcessedAgentData):
            Processed data containing the classified state of the
            road surface and agent data.
    """
    road_state = "good" if 15000 >
agent_data.accelerometer.z > 0 \
        else "bad" if agent_data.accelerometer.z < 17000
or agent_data.accelerometer.z > -1000 \
        else "very bad"
    return ProcessedAgentData(road_state=road_state,
agent_data=agent_data)

```

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

<https://github.com/CherpakAndrii/IntellectualEmbeddedSystems/tree/lab-4/edge>

На цьому етапі лабораторну роботу можна вважати виконаною. Піднімемо докер-контекнер та перевіримо результати. Команди для розгортання контейнеру зображено на рисунках 3-6.

```

Terminal: Local x + v
(venv) PS D:\Education\8 sem\Embedded\edge\docker> docker-compose up --build --detach
[+] Building 0.0s (0/0)
[+] Building 4.9s (19/19)
[+] Building 10.1s (26/26)
[+] Building 20.4s (36/36) FINISHED
=> [fake_agent internal] load build definition from Dockerfile
=> => transferring dockerfile: 443B
=> [fake_agent internal] load .dockerignore
=> => transferring context: 2B
=> [hub internal] load metadata for docker.io/library/python:latest
=> [store internal] load .dockerignore
=> => transferring context: 2B
=> [store internal] load build definition from Dockerfile
=> => transferring dockerfile: 482B
=> [fake_agent auth] library/python:pull token for registry-1.docker.io
=> [hub 1/5] FROM docker.io/library/python:latest@sha256:336461f63f4eb1100e178d5acbfea3d1a5b2a53dea88aa0f9b8482d4d02e981c
=> [fake_agent internal] load build context
=> => transferring context: 1.26kB
=> [store internal] load build context
=> => transferring context: 290.25kB
=> CACHED [fake_agent 2/5] WORKDIR /usr/agent
=> CACHED [fake_agent 3/5] COPY requirements.txt .
=> CACHED [fake_agent 4/5] RUN pip install -r requirements.txt
=> CACHED [fake_agent 5/5] COPY src/ .
=> [fake_agent] exporting to image
=> => exporting layers
=> => writing image sha256:02b4f7a606c4bb036e7a6a2b5af22983ec8f0a718882b5a332706a15c7fb5e9d
=> => naming to docker.io/library/road_vision-fake_agent
=> CACHED [hub 2/5] WORKDIR /app

```

```
=> => exporting layers
=> => writing image sha256:cfa2d8a278d61cee98093aef47080d549f86ba770364619d05b6f55b6279fc4d
=> => naming to docker.io/library/road_vision-edge
[+] Running 14/14
✔ Network road_vision_hub Created 0.0s
✔ Network road_vision_hub_store Created 0.1s
✔ Network road_vision_mqtt_network Created 0.1s
✔ Network road_vision_edge_hub Created 0.5s
✔ Network road_vision_hub_redis Created 0.2s
✔ Network road_vision_db_network Created 0.2s
✔ Container redis_edge Started 0.2s
✔ Container postgres_db Started 7.7s
✔ Container mqtt_edge Started 7.1s
✔ Container pgadmin4 Started 7.7s
✔ Container store_edge Started 7.7s
✔ Container agent_edge Started 3.7s
✔ Container hub_edge Started 2.8s
✔ Container edge_edge Started 2.1s
(venv) PS D:\Education\8 sem\Embedded\edge\docker>
```

```
2024-03-25 22:59:53 store_edge | INFO: Started server process [1]
2024-03-25 22:59:54 hub_edge | INFO: Started server process [1]
2024-03-25 22:59:54 hub_edge | [2024-03-25 20:59:54,207] [INFO] [main] Connected to MQTT broker
2024-03-25 22:59:53 store_edge | INFO: Waiting for application startup.
2024-03-25 22:59:48 pgadmin4 | postfix/postlog: starting the Postfix mail system
2024-03-25 22:59:53 store_edge | INFO: Application startup complete.
2024-03-25 22:59:53 store_edge | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
2024-03-25 22:59:48 redis_edge | 1:C 25 Mar 2024 20:59:48.179 # WARNING Memory overcommit must be enabled! Without it,
a background save or replication may fail under low memory condition. Being disabled, it can also cause failures without
low memory condition, see https://github.com/jemalloc/jemalloc/issues/1328. To fix this issue add 'vm.overcommit_memory =
1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
2024-03-25 22:59:47 postgres_db |
2024-03-25 22:59:48 mqtt_edge | 1711400388: mosquitto version 2.0.18 starting
2024-03-25 22:59:47 postgres_db | PostgreSQL Database directory appears to contain a database; Skipping initialization
2024-03-25 22:59:48 mqtt_edge | 1711400388: Config loaded from /mosquitto/config/mosquitto.conf.
2024-03-25 22:59:54 hub_edge | INFO: Waiting for application startup.
2024-03-25 22:59:48 mqtt_edge | 1711400388: Opening ipv4 listen socket on port 1883.
2024-03-25 22:59:54 store_edge | INFO: 172.27.0.3:38342 - "POST /processed_agent_data/ HTTP/1.1" 200 OK
2024-03-25 22:59:54 hub_edge | INFO: Application startup complete.
2024-03-25 22:59:54 store_edge | INFO: 172.27.0.3:38358 - "POST /processed_agent_data/ HTTP/1.1" 200 OK
2024-03-25 23:02:29 edge_edge | [2024-03-25 21:02:29,388] [INFO] [agent_mqtt_adapter] Connected to MQTT broker. Topic:
agent_data_topic
2024-03-25 22:59:48 redis_edge | 1:C 25 Mar 2024 20:59:48.179 * o000o000o000o Redis is starting o000o000o000o
2024-03-25 22:59:48 redis_edge | 1:C 25 Mar 2024 20:59:48.179 * Redis version=7.2.4, bits=64, commit=00000000, modified
=0, pid=1, just started
2024-03-25 22:59:48 redis_edge | 1:C 25 Mar 2024 20:59:48.179 # Warning: no config file specified, using the default co
nfig. In order to specify a config file use redis-server /path/to/redis.conf
2024-03-25 22:59:48 redis_edge | 1:M 25 Mar 2024 20:59:48.179 * monotonic clock: POSIX clock_gettime
2024-03-25 22:59:48 redis_edge | 1:M 25 Mar 2024 20:59:48.180 * Running mode=standalone, port=6379.
2024-03-25 22:59:48 redis_edge | 1:M 25 Mar 2024 20:59:48.181 * Server initialized
2024-03-25 22:59:47 postgres_db |
2024-03-25 22:59:54 store_edge | INFO: 172.27.0.3:38366 - "POST /processed_agent_data/ HTTP/1.1" 200 OK
2024-03-25 22:59:47 postgres_db | 2024-03-25 20:59:47.663 UTC [1] LOG: starting PostgreSQL 16.2 (Debian 16.2-1.pgdg120+
2) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
```

Рисунки 2-4 – Підняття docker-контейнерів з консолі

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

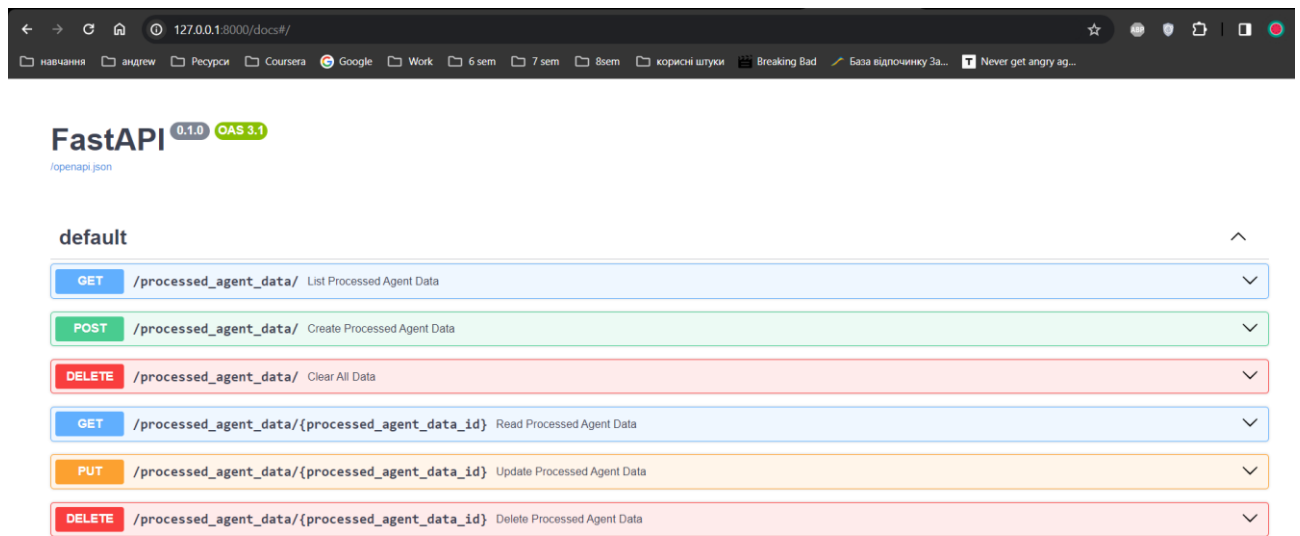


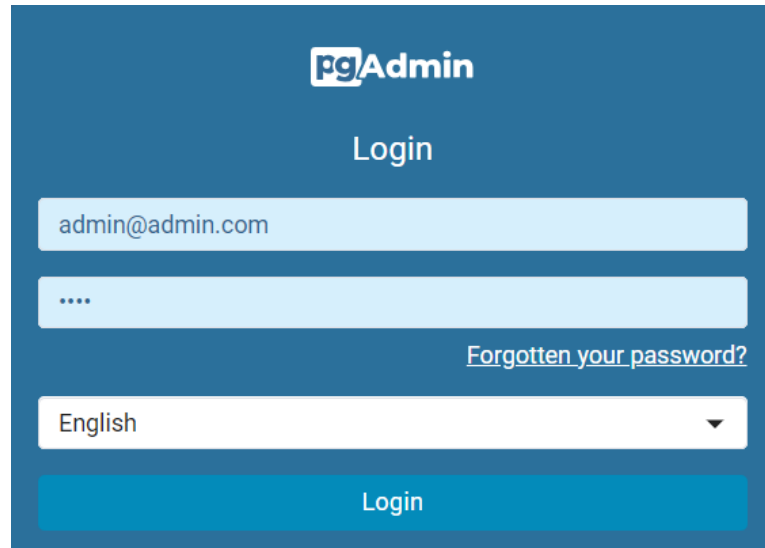
Рисунок 5 – SwaggerUI документація service store

Також одразу перевіримо, що на базу даних було надіслано зібрану інформацію. Для цього виконаємо запит «GET /processed_agent_data/» із запропонованих.

Curl	
<pre>curl -X 'GET' \ 'http://127.0.0.1:8000/processed_agent_data/' \ -H 'accept: application/json'</pre>	
Request URL	
<pre>http://127.0.0.1:8000/processed_agent_data/</pre>	
Server response	
Code	Details
200	<div>Response body</div> <pre>[{ "y": 4, "x": -17, "road_state": "very bad", "longitude": 50.450386085935094, "user_id": 0, "id": 1, "z": 16516, "latitude": 30.524547100067142, "timestamp": "2024-03-25T18:34:45.255130" }, { "y": 51, "x": -55, "road_state": "very bad", "longitude": 50.45333844051969, "user_id": 0, "id": 2, "z": 16757, "latitude": 30.521866627712697, "timestamp": "2024-03-25T18:34:47.281285" }, { "y": 20, "x": -48, "road_state": "very bad", "longitude": 50.45533526269925, "user_id": 0, "id": 3 }]</pre> <div>Response headers</div> <pre>content-length: 13053 content-type: application/json</pre>

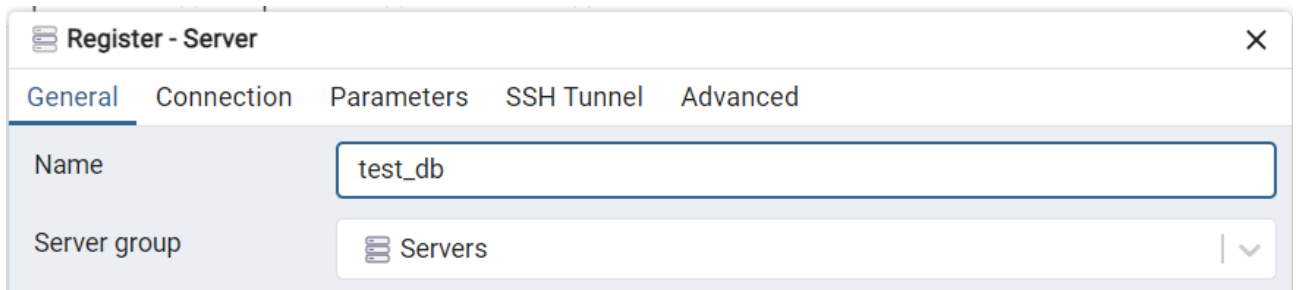
Рисунок 6 – Дані, уже надіслані агентом до БД

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:



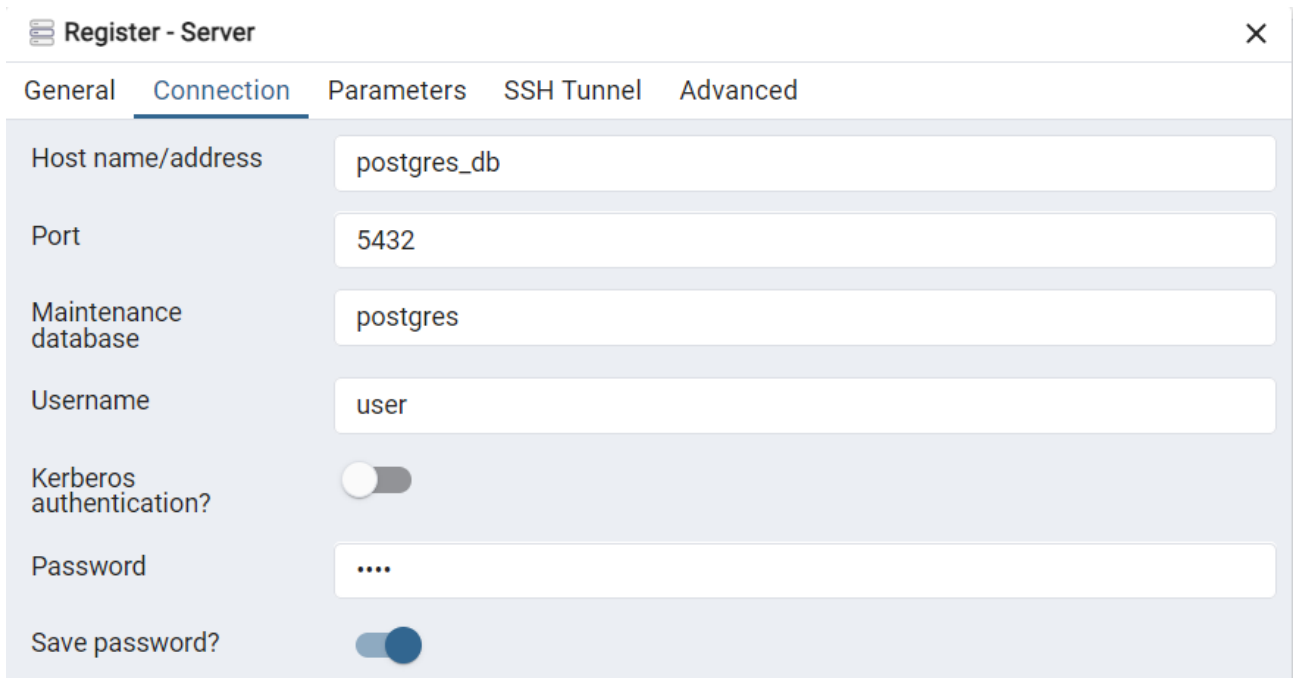
The image shows the PgAdmin login interface. At the top, the 'PgAdmin' logo is displayed. Below it is the 'Login' heading. There are two input fields: the first contains the email 'admin@admin.com', and the second contains masked characters '....'. To the right of the password field is a link that says 'Forgotten your password?'. Below these fields is a language dropdown menu currently set to 'English'. At the bottom is a large blue 'Login' button.

Рисунок 7 – Вхід у PgAdmin



This is a screenshot of the 'Register - Server' dialog box in PgAdmin, with the 'General' tab selected. The 'Name' field contains 'test_db'. The 'Server group' dropdown menu is set to 'Servers'.

Рисунок 8 – Під'єднання до бази даних у PgAdmin



This is a screenshot of the 'Register - Server' dialog box in PgAdmin, with the 'Connection' tab selected. The fields are filled with the following information: 'Host name/address' is 'postgres_db', 'Port' is '5432', 'Maintenance database' is 'postgres', and 'Username' is 'user'. The 'Kerberos authentication?' toggle is turned off, and the 'Save password?' toggle is turned on. The password field contains masked characters '....'.

Рисунок 9 – Під'єднання до бази даних у PgAdmin - продовження

Query

Query History

Scratch Pad

1 SELECT * FROM public.processed_agent_data

2 ORDER BY id DESC LIMIT 100;

Data Output

Messages

Notifications

	id [PK] integer	road_state character varying	user_id integer	x double precision	y double precision	z double precision	latitude double precision	longitude double precision	timestamp timestamp without time zone
1	73	good	0	17243	1243	13860	30.515411884765854	50.45089226323053	2024-03-25 21:02:32.498165
2	72	bad	0	568	12	18266	30.516685162980206	50.4541350972792	2024-03-25 21:02:30.468041
3	71	bad	0	-102	-1922	16382	30.51682186381075	50.45397769437711	2024-03-25 21:00:59.15368
4	70	bad	0	-1461	3799	15454	30.520474423051503	50.455093555427055	2024-03-25 21:00:57.129718
5	69	good	0	-172	-10976	12414	30.522413170763514	50.451757508676074	2024-03-25 21:00:55.103121
6	68	good	0	2318	-15980	2938	30.52285809923405	50.44907610254229	2024-03-25 21:00:53.078768
7	67	good	0	-1055	-14411	8280	30.51871360071455	50.44829852498907	2024-03-25 21:00:51.055073
8	66	bad	0	-2210	4881	15448	30.514976724501317	50.44984925455253	2024-03-25 21:00:49.030523
9	65	good	0	-3047	12914	9150	30.51666209617084	50.4532423325371	2024-03-25 21:00:47.007752
10	64	good	0	-2995	11821	10492	30.519141362963893	50.455275631429274	2024-03-25 21:00:44.983397
11	63	bad	0	-1254	4217	15738	30.522109535735282	50.45263580414475	2024-03-25 21:00:42.960592
12	62	bad	0	-4423	285	17197	30.523698667491768	50.44979880515514	2024-03-25 21:00:40.937208
13	61	good	0	-16079	-673	6805	30.52008010397022	50.44807133976736	2024-03-25 21:00:38.911431
14	60	bad	0	-16569	-1653	-3967	30.514614090947536	50.4489800806542	2024-03-25 21:00:36.885128
15	59	good	0	-14790	-1406	6013	30.51606462516266	50.45245677624752	2024-03-25 21:00:34.861958
16	58	bad	0	724	-188	16323	30.517733705776866	50.455350170516745	2024-03-25 21:00:32.835221
17	57	good	0	15106	-829	10100	30.521748100680377	50.4535017075372	2024-03-25 21:00:30.806653

Рисунок 10 – Відслідковування початкового стану бази даних у PgAdmin

Бачимо, що усі дані, надані рейковим агентом, проходять обробку та потрапляють у базу даних. Отже, лабораторну роботу виконано.

Висновок: під час виконання комп'ютерного практикуму я розібрався з наданою кодовою базою, реалізував логіку отримання даних від агентів, їх обробки, тобто визначення якості дороги, а також надсилення на hub, а потім розгорнув необхідні сервіси у Docker та протестував коректність роботи. Весь функціонал відпрацьовував рівно як і очікувалося.