WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

STUDIA II°

Temat: **SPRAWOZDANIE Z REALIZACJI ĆWICZENIA LABORATORYJNEGO**

INFORMATYKA

(kierunek studiów)

INŻYNIERIA SYSTEMÓW

(specjalność)

Wykonał:

Radosław Relidzyński

Prowadzący:

mgr inż. Tomasz Gutowski

Warszawa 2023

Spis treści

Rozdział I. Treść zadań

I.1. Etap 1

Zadanie + sprawozdanie

I etap:

- Zaimplementować komunikację z wykorzystaniem protokołu MQTT
  - Co najmniej 3 topic'i, różne QoS – z uzasadnieniem
  - Co najmniej 2 subskrypcje i 2 publikacje z poziomu paho-mqtt

- Zaimplementować REST API z wykorzystaniem fast-api oraz komunikację z API z poziomu aplikacji klienckiej
  - Co najmniej 1 usługa pobierające dane
  - Co najmniej 1 usługa modyfikująca/dodająca dane

  *Możliwe przechowywanie obiektów w pamięci operacyjnej

I.2. Etap 2

Zadanie + sprawozdanie

II etap:

- I etap
- Usługi REST API wywoływane przez klienta MQTT, odpowiedź przetwarzana przez klienta
- Zapisywanie i odczytywanie danych (REST API) z pliku lub bazy danych
- Prosta walidacja danych

I.3. Etap 3

Zadanie + sprawozdanie

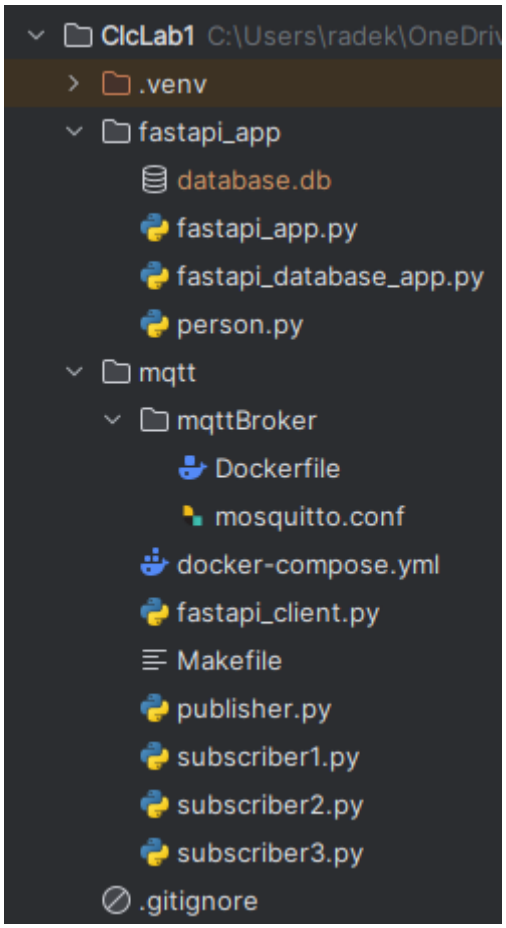
III etap:

- II etap
- Zapisywanie i odczytywanie danych (REST API) z bazy danych
- Zapis danych otrzymywanych przez MQTT do bazy danych

Rozdział II. Rozwiązanie zadania 1

II.1. Implementacja zadania 1

II.1.1. Struktura projektu



II.1.2. fastapi_app.py

```python
import random

from fastapi import FastAPI, HTTPException

import fastapi_database_app as db
from person import Person

app = FastAPI()


def validate_person_data(person_data):
    return set(Person.__fields__.keys()) == set(person_data.keys())


@app.get("/get_first_names")
def get_data1():
    result = db.get_data_from_db()
    return [row[1] for row in result]


@app.get("/get_emails")
def get_data2():
    result = db.get_data_from_db()
    return [row[3] for row in result]


@app.get("/get_all")
def get_data3():
```

```
    result = db.get_data_from_db()
    return result


@app.post("/add_person/")
def add_person(person: Person):
    if validate_person_data(person):
        return db.add_data_to_db(person)
    else:
        return HTTPException(
            status_code=400,
            detail="Invalid data. Ensure all required fields are provided."
        )


@app.post("/add_random_person")
def add_random_person():
    first_names = ["A", "B", "C"]
    last_names = ["AA", "BB", "CC"]

    first_name = random.choice(first_names)
    last_name = random.choice(last_names)
    age = random.randint(18, 60)
    email = f"{first_name}.{last_name}@gmail.com"

    person = Person()
    person.first_name = first_name
    person.last_name = last_name
    person.age = age
    person.email = email

    # db.add_data_to_db(person)
    return db.add_data_to_db(person)


@app.put("/update_person/")
def update_person(person_to_update: Person):
    if validate_person_data(person_to_update):
        return db.update_db_person(person_to_update)
    else:
        return HTTPException(
            status_code=400,
            detail="Invalid data. Ensure all required fields are provided."
        )

@app.delete("/delete_people")
def delete_people():
    return db.delete_data_from_db()


if __name__ == "__main__":
    db.init_database()
    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)
```

II.1.3. fastapi_database_app.py

```
import sqlite3
```

```
from person import Person


def run_query(query, params=None):
    conn = sqlite3.connect("database.db")
    cursor = conn.cursor()

    if params:
        cursor.execute(query, params)
    else:
        cursor.execute(query)

    conn.commit()
    result = cursor.fetchall()
    cursor.close()
    conn.close()
    return result


def init_database():
    create_query = """
    CREATE TABLE IF NOT EXISTS people (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        age INT,
        email VARCHAR(50)
    )
    """
    run_query(create_query)
    return "Database initialized"


def add_data_to_db(person: Person):
    query = "INSERT INTO people (first_name, last_name, age, email) VALUES (?,
?, ?, ?)"
    person_data = (person.first_name, person.last_name, person.age,
person.email)
    run_query(query, person_data)
    return "Person added successfully"


def get_data_from_db():
    result = run_query("SELECT * FROM people")
    return result


def delete_data_from_db():
    run_query("DELETE FROM people")
    return "People successfully deleted"


def update_db_person(person_to_update: Person):
    query_str = f"UPDATE people SET %s WHERE %s"
    update_str = f" age = ?, email = ? "
    person_to_update_str = f" first_name = ? AND last_name = ? "
    query = query_str % (update_str, person_to_update_str)
    person_data = (person_to_update.first_name, person_to_update.last_name,
person_to_update.age, person_to_update.email)
    run_query(query, person_data)
```

```
    return f"Person {person_to_update.first_name} {person_to_update.last_name}
updated successfully"
```

II.1.4. person.py

```
from pydantic import BaseModel


class Person(BaseModel):
    first_name: str
    last_name: str
    age: int
    email: str
```

II.1.5. Dockerfile

```
FROM eclipse-mosquitto:2.0

COPY mosquitto.conf /mosquitto/config/mosquitto.conf
```

II.1.6. mosquitto.conf

```
allow_anonymous true
listener 1883
```

II.1.7. docker-compose.yml

```
services:
  mqtt_broker:
    container_name: mqttBroker
    build:
      context: mqttBroker
      dockerfile: Dockerfile
    ports:
      - "1883:1883"
```

II.1.8. fastapi_client.py

```
import requests


class FastapiClient:

    def __init__(self):
        self.url = "http://127.0.0.1:8000/"

    def add_person(self, person_data):
        add_url = self.url + "add_person/"
        return requests.post(add_url, json=person_data)

    def update_person(self, person_to_update):
        add_url = self.url + "update_person/"
        return requests.put(add_url, json=person_to_update)

    def show_people(self):
```

```
        show_url = self.url + "get_all/"
        return requests.get(show_url)
```

II.1.9. Makefile

```
build:
    docker-compose up --build

terminate:
    docker-compose down

clear:
    docker system prune -a --volumes -f
```

II.1.10. publisher.py:

```
""" Publisher script that sends values to sensors """
import json
import time

import paho.mqtt.client as mqtt

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)

client.connect("localhost", 1883, 60)

# qos 2
print("Publishing add_person with qos=2")
people_data = [
    {"first_name": "John", "last_name": "Doe", "age": 30, "email":
"john.doe@example.com"},
    {"first_name": "Alice", "last_name": "Smith", "age": 25, "email":
"alice.smith@example.com"},
    {"first_name": "Bob", "last_name": "Brown", "age": 40, "email":
"bob.brown@example.com"},
]
for person_data in people_data:
    client.publish("people/add_person", payload=json.dumps(person_data), qos=2)

# qos 1
print("Publishing update_person with qos=1")
person_to_update = {"first_name": "John", "last_name": "Doe", "age": 30,
"email": "john.doe.thesecond@example.com"}
client.publish("people/update_person", payload=json.dumps(person_to_update),
qos=1)

# qos 0
print("Publishing get_all_people with qos=0")
client.publish("people/get_all_people", payload="get_all_people", qos=0)

client.loop_start()

time.sleep(2)

client.disconnect()
```

II.1.11. subscriber1.py:

```
"""
Subscriber that registers sensor values from qos=0
qos 0 (Zero Assurance) is not checking if the receiver got the message
"""
import json

import paho.mqtt.client as mqtt

from fastapi_client import FastapiClient

fastapi_client = FastapiClient()


def on_connect(client, userdata, flags, reason_code, properties):
    print("Subscriber1 connected with result code " + str(reason_code))


def on_message(client, userdata, msg):
    print("Subscriber1 " + msg.topic + ": " + str(msg.payload) + ", qos = " +
str(msg.qos))
    person_data = json.loads(msg.payload)

    response = fastapi_client.add_person(person_data)

    print(f"Repsonse: {response}")

    if response.status_code == 200:
        print(f"Response: {response.json()}")
    else:
        print(f"Error {response.status_code}: {response.text}")


client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

client.subscribe("people/add_person", qos=2)

try:
    print("Subscriber1 working...")
    client.loop_forever()
except KeyboardInterrupt:
    print("Subscriber1 shutdown")

client.disconnect()
```

II.1.12. subscriber2.py:

```
"""
Subscriber that registers sensor values from qos=1
qos 1 (At Least One) is sending messages till he gets the confirmation
"""
import json

import paho.mqtt.client as mqtt

from fastapi_client import FastapiClient
```

```
fastapi_client = FastapiClient()


def on_connect(client, userdata, flags, reason_code, properties):
    print("Subscriber2 connected with result code " + str(reason_code))


def on_message(client, userdata, msg):
    print("Subscriber2 " + msg.topic + ": " + str(msg.payload) + ", qos = " +
str(msg.qos))
    person_to_update = json.loads(msg.payload)

    response = fastapi_client.update_person(person_to_update)

    print(f"Repsonse: {response}")

    if response.status_code == 200:
        print(f"Response: {response.json()}")
    else:
        print(f"Error {response.status_code}: {response.text}")


client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

client.subscribe("people/update_person", qos=1)

try:
    print("Subscriber2 working...")
    client.loop_forever()
except KeyboardInterrupt:
    print("Subscriber2 shutdown")

client.disconnect()
```

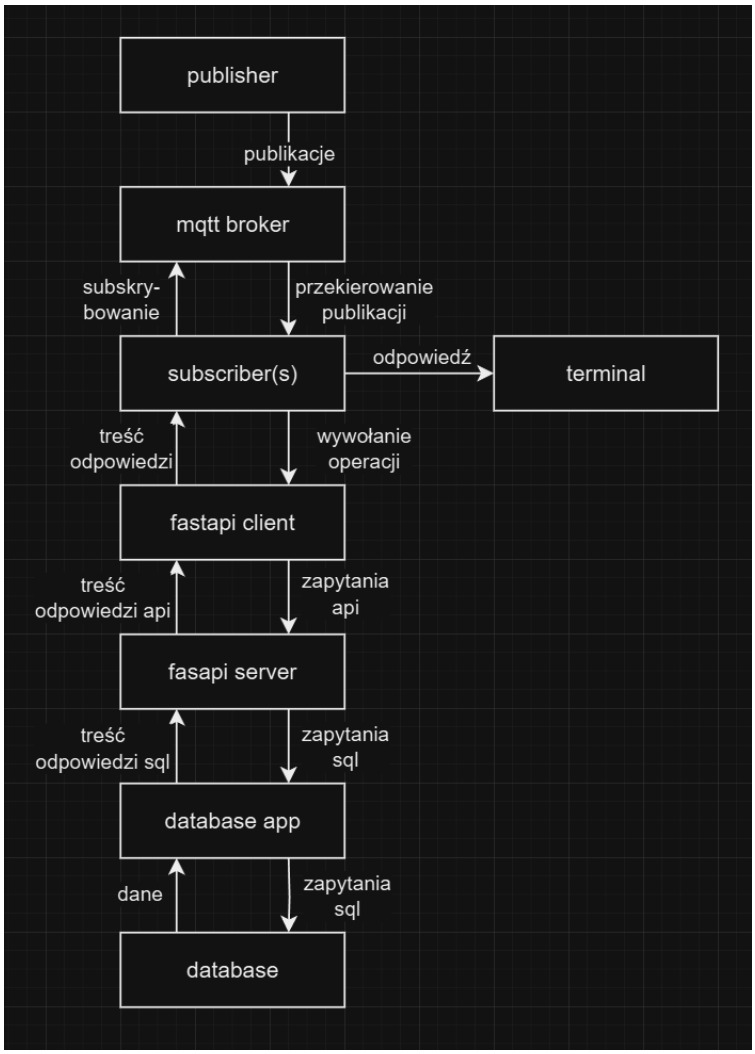
II.1.13. subscriber3.py:

```
"""
Subscriber that registers sensor values from qos=2
qos 2 (Exactly One) is sending one message and looks for confirmation
"""

import paho.mqtt.client as mqtt

from fastapi_client import FastapiClient

fastapi_client = FastapiClient()


def on_connect(client, userdata, flags, reason_code, properties):
    print("Subscriber3 connected with result code " + str(reason_code))


def on_message(client, userdata, msg):
    print("Subscriber3 " + msg.topic + ": " + str(msg.payload) + ", qos = " +
str(msg.qos))
```

```
    response = fastapi_client.show_people()

    print(f"Repsonse: {response}")

    if response.status_code == 200:
        print(f"Response: {response.json()}")
        print()
        print("All people:")
        for person in response.json():
            print(person)
    else:
        print(f"Error {response.status_code}: {response.text}")


client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

client.subscribe("people/get_all_people", qos=0)

try:
    print("Subscriber3 working...")
    client.loop_forever()
except KeyboardInterrupt:
    print("Subscriber3 shutdown")

client.disconnect()
```

II.2. Opis działania zadania

Struktura działania systemu:



Są 3 sensory:

- add_person – dodawanie ludzi, kluczowy element
- update_person – aktualizowanie informacji o człowieku, nadal istotny, ale nie kluczowy
- get_all_people – wyświetlenie wszystkich ludzi, poboczny element (np. tylko do logów)

Każdy subscriber podłącza się do danego sensora (każdy przy pomocy innego qos) i w nieskończonej pętli nasłuchuje nadchodzących wartości.
Publisher wysyła sygnały do wszystkich sensorów w taki sposób, żeby każdy subscriber otrzymał odpowiednią dla niego wartość.

Subscriber1 działa przy użyciu qos 2 (Exactly One), gwarantuje, że wiadomość zostanie dostarczona dokładnie raz, bez duplikacji. Nawiązuje trwałe połączenie, a wiadomość oznacza unikatowym identyfikatorem. Przy pomocy połączenia upewnia się, że wiadomość została dostarczona.
Mechanizm qos 2 sprawdza się przy krytycznych sensorach, przy których dostarczenie danych jest niezbędne oraz to w jaki sposób są przekazywane (pod względem chociażby ilości przesyłanych wiadomości).

Subscriber2 działa przy użyciu qos 1 (At Least One), gwarantuje, że wiadomość zostanie dostarczona co najmniej raz. Wysyła tą samą wiadomość do momentu potwierdzenia odbioru.

Mechanizm qos 1 sprawdza się przy średnio krytycznych sensorach, przy których dostarczenie danych jest niezbędne.

Subscriber1 działa przy użyciu qos 0 (Zero Assurance), wiadomość jest przesyłana bez upewniania się, że dotrze. Nie sprawdza on dostępności odbiorcy.
Mechanizm qos 0 sprawdza się przy mało krytycznych sensorach, przy których dostarczenie danych nie jest niezbędne.

Sensory po otrzymaniu wiadomości wywołuje odpowiednią dla niego instrukcję klienta fastapi

Klient fastapi konstruuje zapytanie http do serwera odpowiednie dla sensora.

Serwer fastapi posiada mechanizm przechwytywania zapytań http i na ich podstawie obsługuje system (w tym przypadku bazę danych) w odpowiedni dla danego zapytania sposób.

Serwer wystawiany jest pod adresem http://0.0.0.0:8000/
Przyjmuje on różne zapytania http (get, post, put, delete) i na ich podstawie wykonuje odpowiednią metodę.
Zapytania get zwracają odpowiednią zawartość z bazy danych
Zapytania post dodają elementy do bazy danych poprzez generowanie wartości lub przejmowanie przekazywanych danych w zapytaniu.
Zapytania put służą do aktualizacji danych w bazie.
Zapytanie delete usuwa zawartość bazy danych.

W ramach serwera przebiega walidacja danych sprawdzająca, czy dane wejściowe odpowiadają tym, które są potrzebne do realizacji wywołanej funkcji dla bazy danych:

```
@app.post("/add_person/")
def add_person(person: Person):
    if validate_person_data(person):
        return db.add_data_to_db(person)
    else:
        return HTTPException(
            status_code=400,
            detail="Invalid data. Ensure all required fields are provided."
        )
```

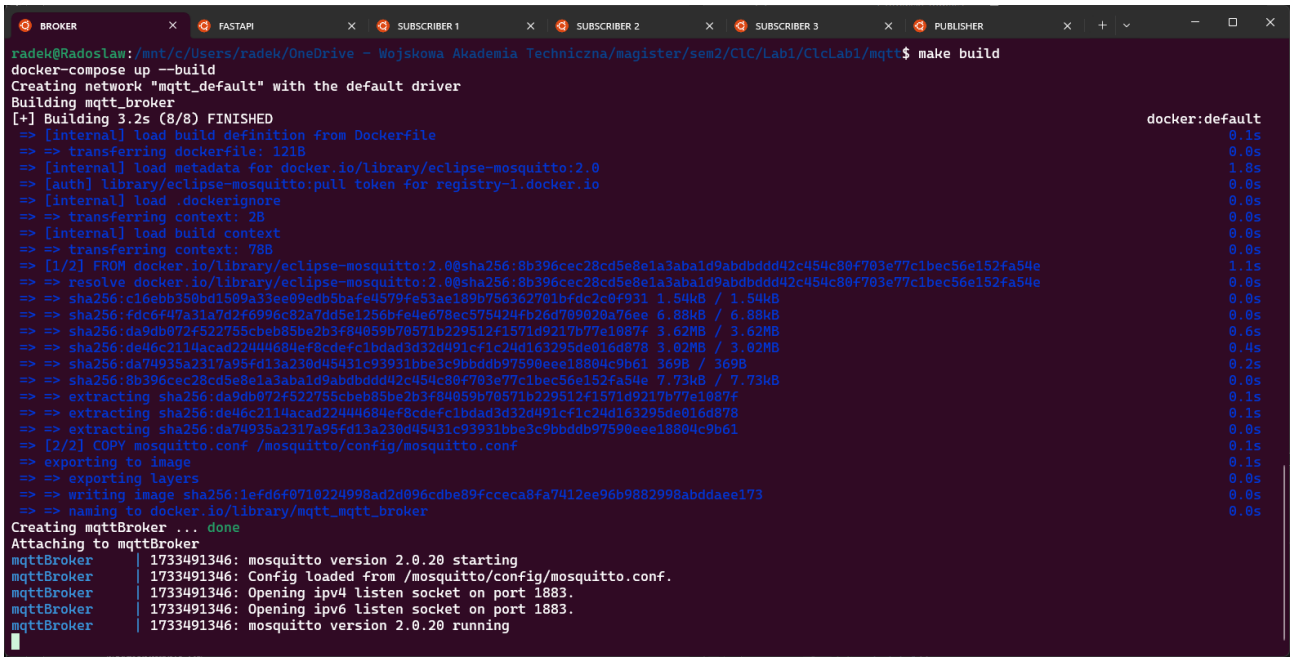
```
@app.put("/update_person/")
def update_person(person_to_update: Person):
    if validate_person_data(person_to_update):
        return db.update_db_person(person_to_update)
    else:
        return HTTPException(
            status_code=400,
            detail="Invalid data. Ensure all required fields are provided."
        )
```

Aplikacja bazy danych formuuje odpowiednie zapytanie sql i wywołuje je przy pomocy sqlite3.

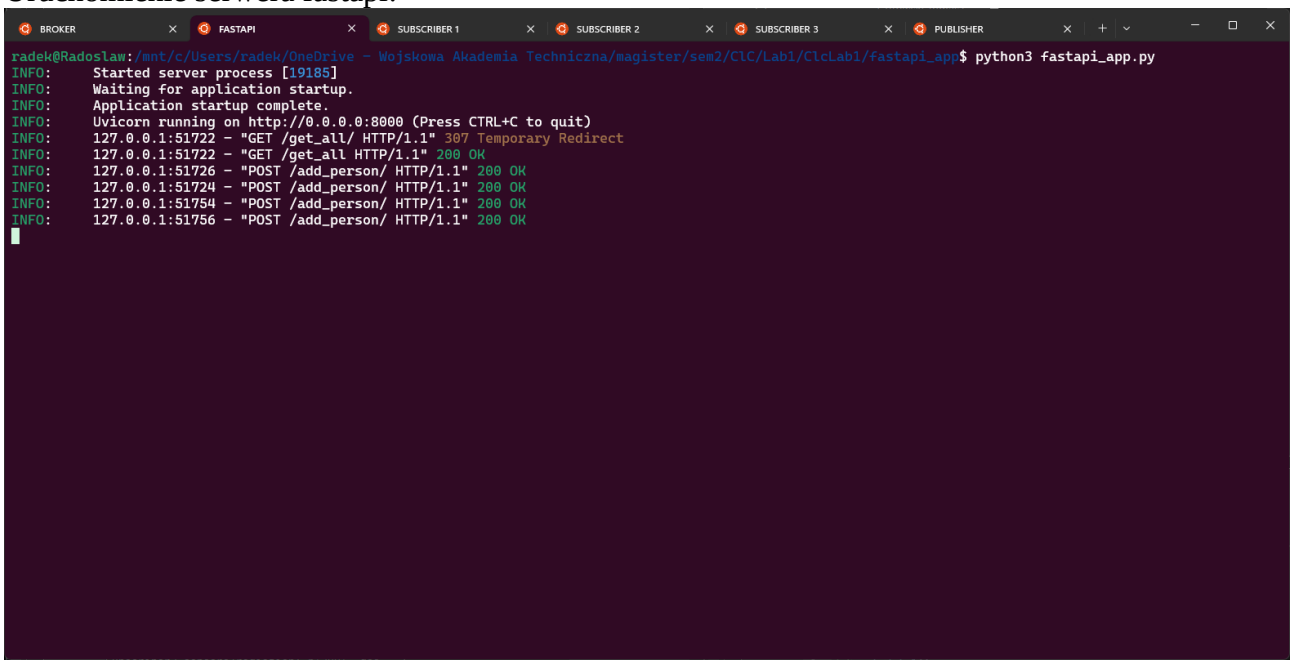
Baza danych przetwarza zapytanie i zwraca wynik aplikacji bazy danych.

II.3. Prezentacja działania zadania

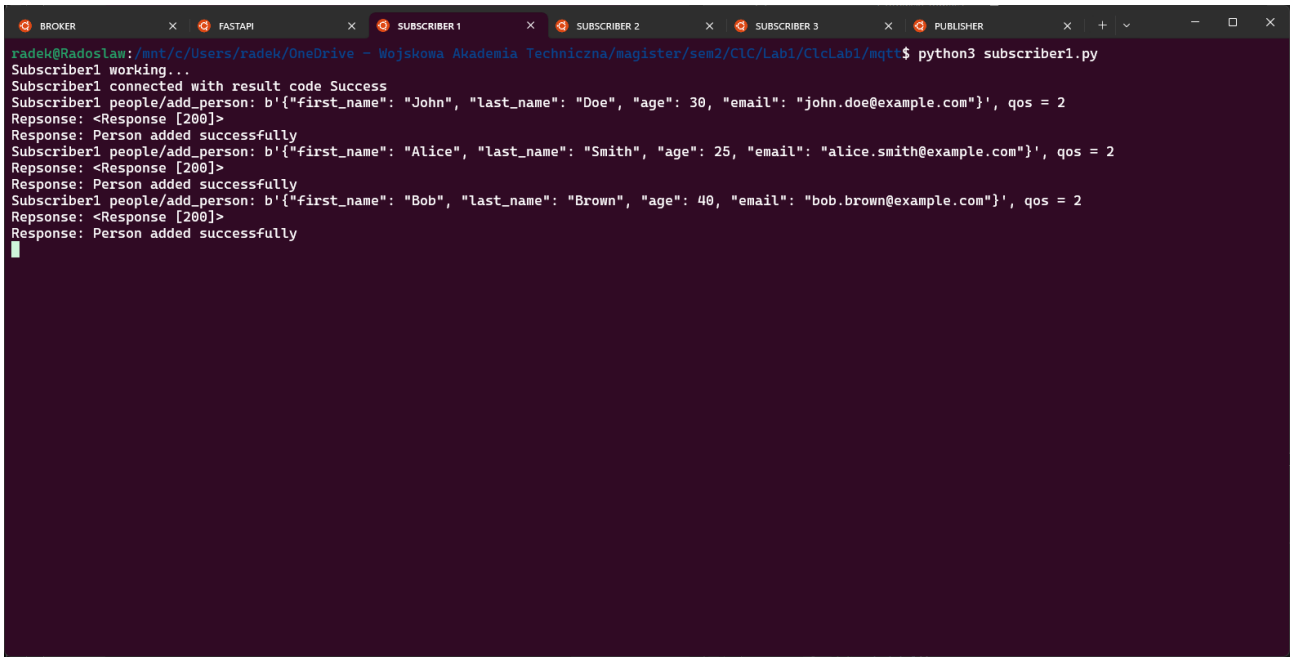
Uruchomienie brokera:



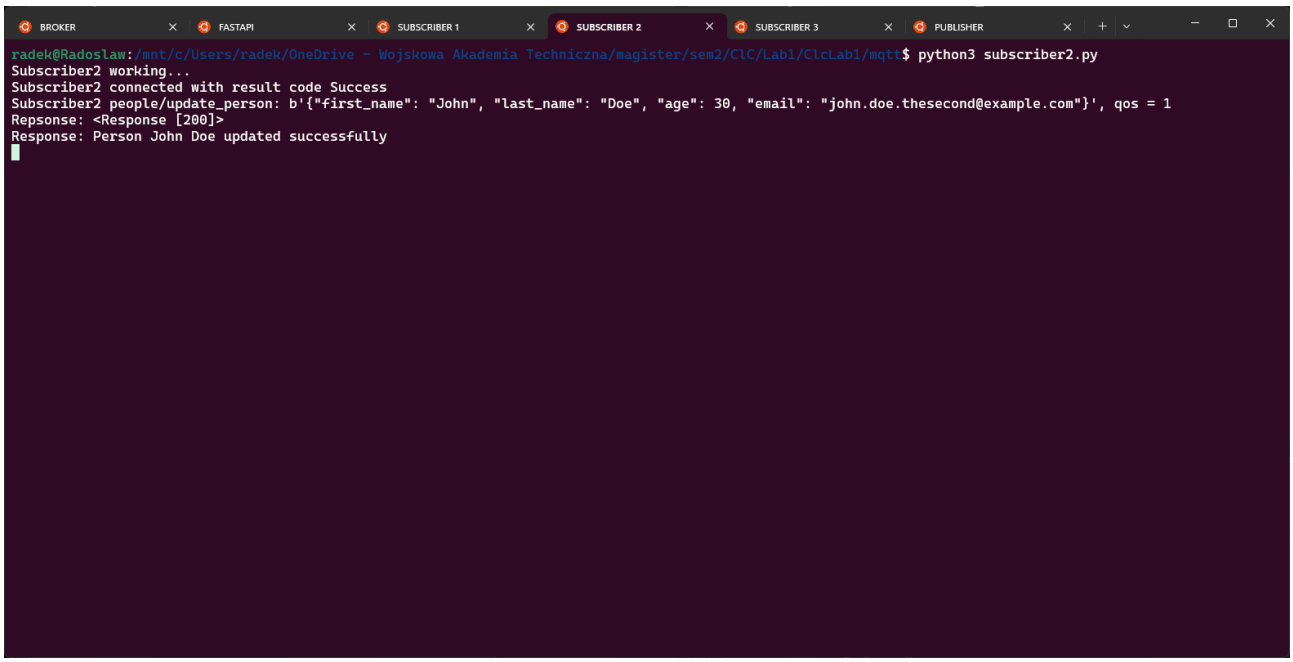
Uruchomienie serwera fastapi:



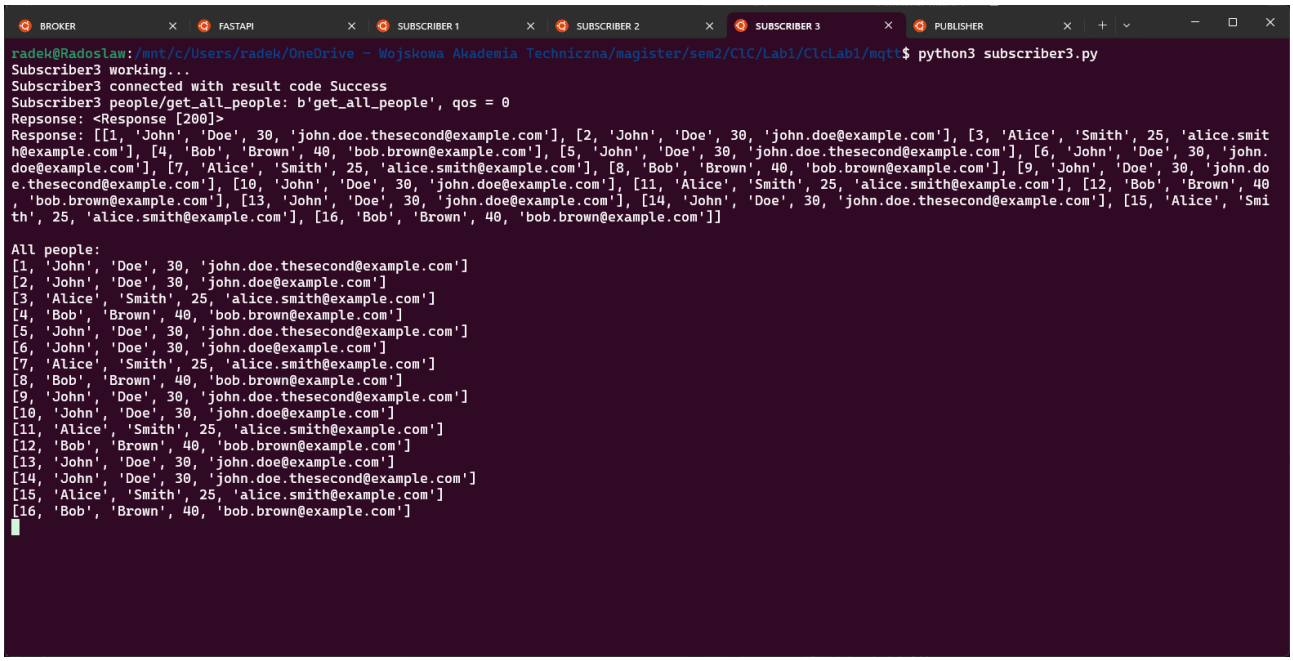
Uruchomienie subscriber1



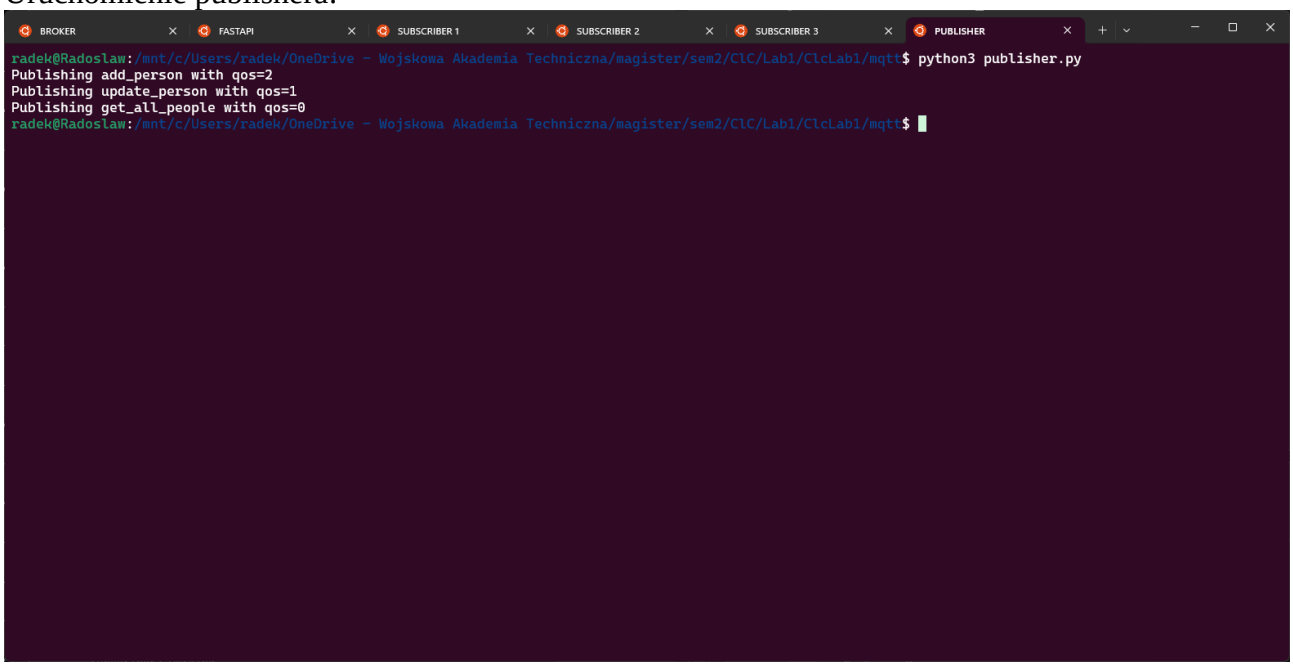
Uruchomienie subscriber2



Uruchomienie subscriber3



Uruchomienie publishera:



Rozdział III. Podsumowanie

Podczas ćwiczenia laboratoryjnego udało się zrealizować przy pomocy mqtt oraz fastapi system komunikacji. System został zaprojektowany w sposób modularny, składając się z różnych komponentów.

W ramach ćwiczenia udało się poznać jak działa protokół mqtt, jak działają różne qos, a także jak tworzyć serwer http z obsługą różnych rodzajów zapytań.

Implementacja zawiera:

- Utworzenie 3 topic’ów

- Zastosowanie różnych qos
- Imlementację REST API z wykorzystaniem fast-api
- Komunikację z API z poziomu aplikacji klienckiej (fastapi_client.py)
- Usługi REST API pobierające dane
- Usługę REST API modyfikującą dane
- Usługę REST API usuwającą dane
- Wywołanie usług REST API przez klienta MQTT przetwarzane przez klienta
- Zapisywanie i odczytywanie danych od MQTT z bazy danych
- Zapisywanie i odczytywanie danych od REST API z bazy danych