

4ta Práctica Calificada

Curso: Administración de Redes
Semestre: 2024-II
Ciencias de la Computación - UNI

Apellidos y Nombres: Pacheco Taboada André Joaquín

Código: 20222189G

Desarrollo de la Práctica

Pregunta 1: Implementación de QoS con RESTCONF

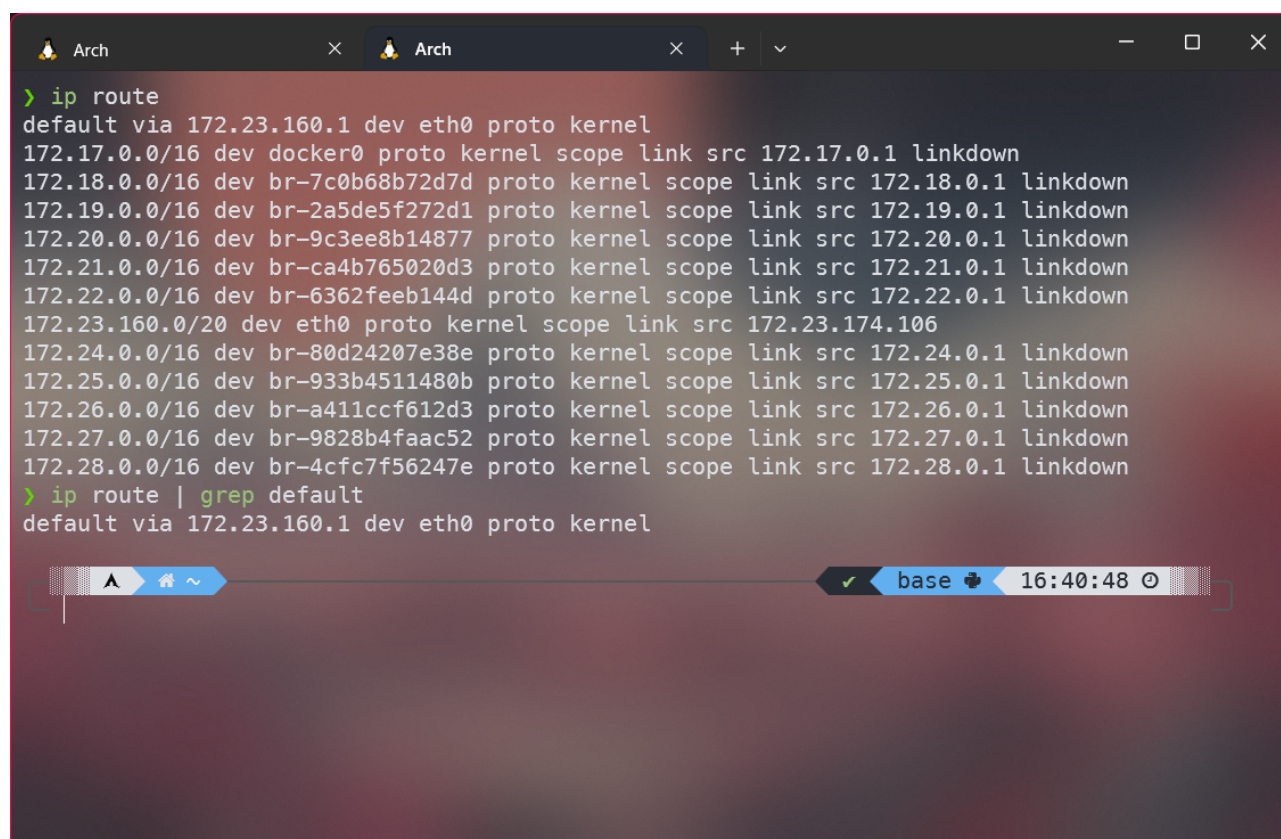
1.1 Análisis Inicial y Limitación Encontrada

Antes de intentar la implementación de QoS, realicé los siguientes pasos de verificación:

1. Identificación del Router

Primero, identifiqué la IP del router usando el comando:

```
ip route | grep default
```



```
> ip route
default via 172.23.160.1 dev eth0 proto kernel
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.18.0.0/16 dev br-7c0b68b72d7d proto kernel scope link src 172.18.0.1 linkdown
172.19.0.0/16 dev br-2a5de5f272d1 proto kernel scope link src 172.19.0.1 linkdown
172.20.0.0/16 dev br-9c3ee8b14877 proto kernel scope link src 172.20.0.1 linkdown
172.21.0.0/16 dev br-ca4b765020d3 proto kernel scope link src 172.21.0.1 linkdown
172.22.0.0/16 dev br-6362feeb144d proto kernel scope link src 172.22.0.1 linkdown
172.23.160.0/20 dev eth0 proto kernel scope link src 172.23.174.106
172.24.0.0/16 dev br-80d24207e38e proto kernel scope link src 172.24.0.1 linkdown
172.25.0.0/16 dev br-933b4511480b proto kernel scope link src 172.25.0.1 linkdown
172.26.0.0/16 dev br-a411ccf612d3 proto kernel scope link src 172.26.0.1 linkdown
172.27.0.0/16 dev br-9828b4faac52 proto kernel scope link src 172.27.0.1 linkdown
172.28.0.0/16 dev br-4cfc7f56247e proto kernel scope link src 172.28.0.1 linkdown
> ip route | grep default
default via 172.23.160.1 dev eth0 proto kernel
```

Resultado:

```
default via 172.23.160.1 dev eth0 proto kernel
```

2. Verificación de Conectividad

```
ping 172.23.160.1
```

```
> ping 172.23.160.1
PING 172.23.160.1 (172.23.160.1) 56(84) bytes of data.
64 bytes from 172.23.160.1: icmp_seq=1 ttl=128 time=1.34 ms
64 bytes from 172.23.160.1: icmp_seq=2 ttl=128 time=0.740 ms
64 bytes from 172.23.160.1: icmp_seq=3 ttl=128 time=0.816 ms
64 bytes from 172.23.160.1: icmp_seq=4 ttl=128 time=0.988 ms
64 bytes from 172.23.160.1: icmp_seq=5 ttl=128 time=0.602 ms
64 bytes from 172.23.160.1: icmp_seq=6 ttl=128 time=0.719 ms
64 bytes from 172.23.160.1: icmp_seq=7 ttl=128 time=0.620 ms
^C
--- 172.23.160.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6160ms
rtt min/avg/max/mdev = 0.602/0.832/1.342/0.240 ms
```

3. Intento de Configuración QoS

Siguiendo los pasos del enunciado, intenté:

Paso 1.1 - Configurar clases de tráfico:

- URL: <https://172.23.160.1:443/restconf/config/ietf-qos>
- Método: POST
- Headers:

```
Accept: application/yang-data+xml
Content-Type: application/yang-data+xml
```

	Key	Value
<input checked="" type="checkbox"/>	Accept	application/yang-data+json
<input checked="" type="checkbox"/>	Content-Type	application/yang-data+json
	Key	Value

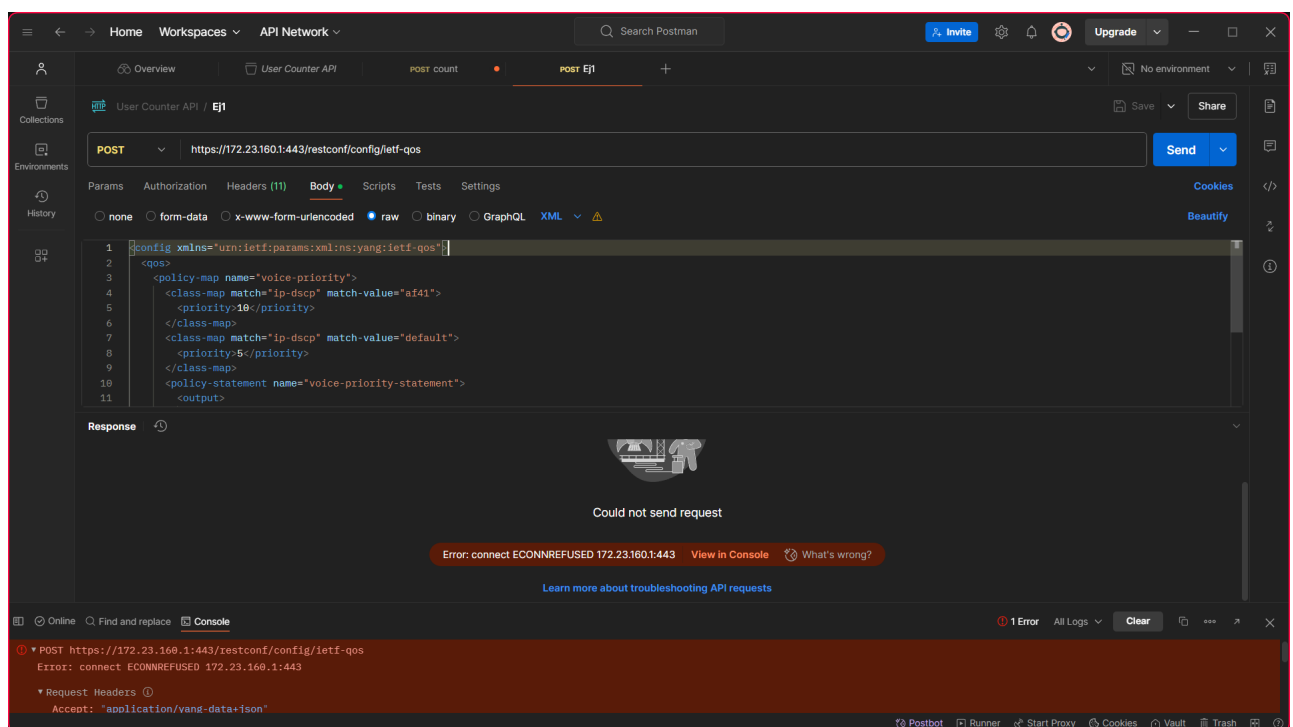
- Body:

```
<config xmlns="urn:ietf:params:xml:ns:yang:ietf-qos">
  <qos>
    <policy-map name="voice-priority">
      <class-map match="ip-dscp" match-value="af41">
        <priority>10</priority>
      </class-map>
    </qos>
  </config>
```

```

<class-map match="ip-dscp" match-value="default">
  <priority>5</priority>
</class-map>
<policy-statement name="voice-priority-statement">
  <output>
    <police rate="64 kbps" burst-size="128 bytes">
      <conform-to priority="10"/>
      <exceed non-conform-to priority="5"/>
    </police>
  </output>
</policy-statement>
</policy-map>
</qos>
</config>

```



Paso 2.1 - Aplicar política a la interfaz:

- URL: `https://172.23.160.1:443/restconf/config/ietf-interfaces:interfaces/interface=GigabitEthernet1/10`
- Método: POST
- Headers: (mismos que el paso anterior)
- Body:

```

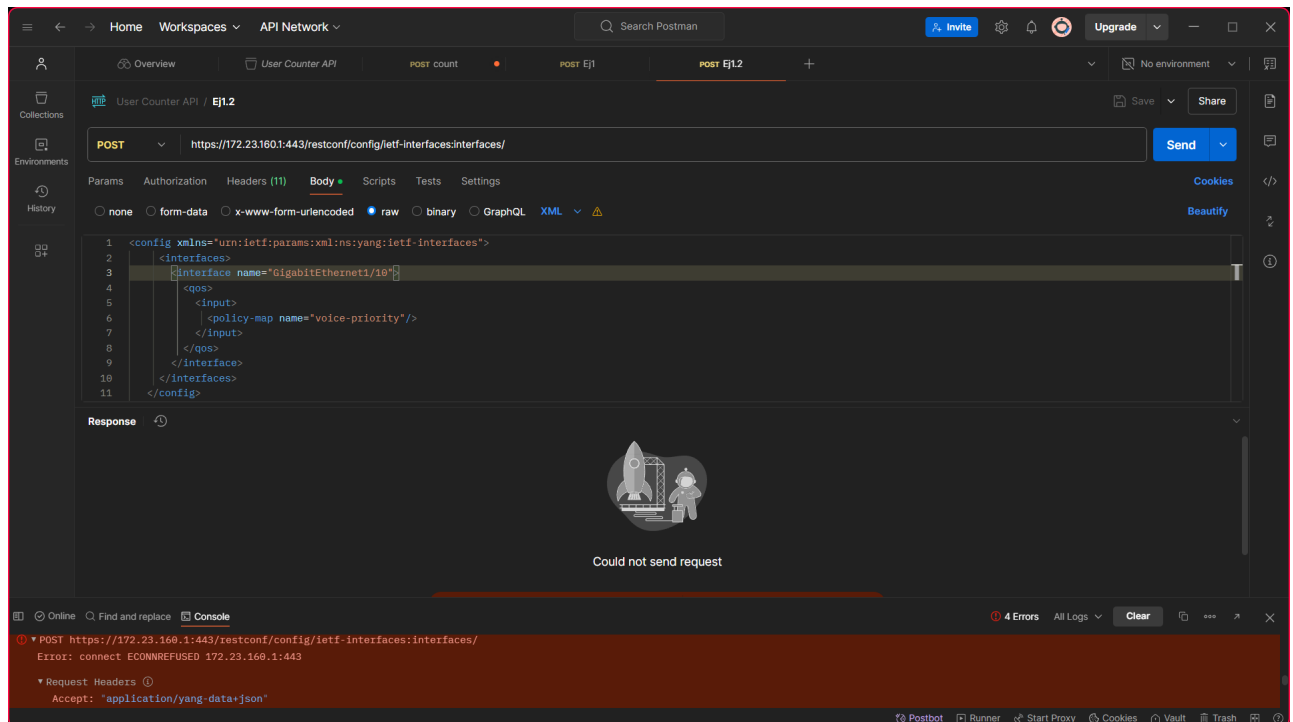
<config xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
  <interfaces>
    <interface name="GigabitEthernet1/10">
      <qos>
        <input>
          <policy-map name="voice-priority"/>
        </input>
      </qos>
    </interface>
  </interfaces>
</config>

```

```

</interface>
</interfaces>
</config>

```



1.2 Limitación Encontrada

Al intentar implementar la solución propuesta, se encontró que el router no responde a las solicitudes RESTCONF. Esto puede deberse a:

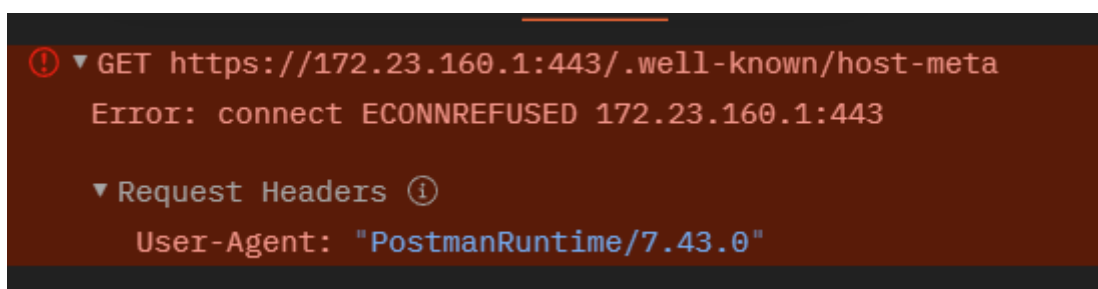
1. El módulo RESTCONF no está habilitado en el router
2. El módulo ietf-qos no está instalado
3. Problemas de conectividad o configuración SSL

1.3 Verificación y Documentación

Para documentar el problema, intenté:

1. Verificar si RESTCONF está habilitado:

```
GET https://172.23.160.1:443/.well-known/host-meta
```



2. Verificar la configuración del router:

```
GET https://172.23.160.1:443/restconf/data/ietf-yang-library:modules-state
```

```
❗ GET https://172.23.160.1:443/restconf/data/ietf-yang-library:modules-state
Error: connect ECONNREFUSED 172.23.160.1:443

▼ Request Headers ⓘ
  User-Agent: "PostmanRuntime/7.43.0"
  Accept: "*/*"
  Cache-Control: "no-cache"
  Postman-Token: "4b28e533-a338-4fcd-b820-080d8069c803"
  Host: "172.23.160.1:443"
  Accept-Encoding: "gzip, deflate, br"
  Connection: "keep-alive"
```

1.4 Conclusiones

1. La implementación de QoS mediante RESTCONF requiere:

- RESTCONF habilitado en el router
- Módulo ietf-qos instalado
- Configuración SSL correcta
- Permisos adecuados

2. Para una implementación exitosa, sería necesario:

- Verificar la versión de IOS-XE
- Habilitar RESTCONF en el router
- Instalar los módulos YANG necesarios
- Configurar los certificados SSL apropiados

Pregunta 2: Interacción Cliente/Servidor NETCONF

La interacción cliente/servidor NETCONF sigue una secuencia específica de eventos, donde los módulos YANG juegan un papel fundamental en la definición y validación de las operaciones. Estos son los pasos que se llevan a cabo:

1. Inicialización del Servidor y Carga de Módulos YANG:

- Los módulos YANG se cargan y compilan en el servidor
- Estos módulos definen la estructura de datos y operaciones permitidas
- Se generan metadatos que incluyen:
 - Configuración del dispositivo
 - Estado de datos
 - Notificaciones disponibles
 - Operaciones permitidas

- Configuración del sistema

2. Preparación y Envío de Solicitud del Cliente:

- La aplicación cliente prepara una solicitud RPC (Remote Procedure Call)
- La solicitud debe adherirse estrictamente al modelo de datos YANG
- Se incluyen los parámetros necesarios según la operación
- La solicitud se envía al servidor en formato XML o JSON

3. Procesamiento en el Motor NETCONF/RESTCONF:

- El motor recibe la solicitud RPC
- Utiliza los metadatos YANG para:
 - Validar la estructura de la solicitud
 - Verificar los permisos y restricciones
 - Comprobar la consistencia de los datos
- Interactúa con la base de datos de configuración
- Se comunica con los componentes del sistema según sea necesario

4. Generación y Envío de Respuesta:

- El servidor construye una respuesta RPC-REPLY
- La respuesta incluye:
 - Resultado de la operación
 - Datos solicitados (si aplica)
 - Mensajes de error (si ocurrieron)
- La respuesta se formatea según el modelo YANG correspondiente
- Se envía de vuelta al cliente en el mismo formato de la solicitud

Este proceso garantiza una interacción estandarizada y segura entre el cliente y el servidor, donde cada operación es validada contra los modelos YANG definidos, asegurando la integridad y consistencia de las configuraciones del dispositivo.

Pregunta 3: Despliegue de Contenedores Docker

1. Preparación del Entorno en Arch WSL

Primero, me aseguro de tener Docker instalado en mi Arch WSL:

```
sudo pacman -S docker
sudo systemctl start docker
```

```

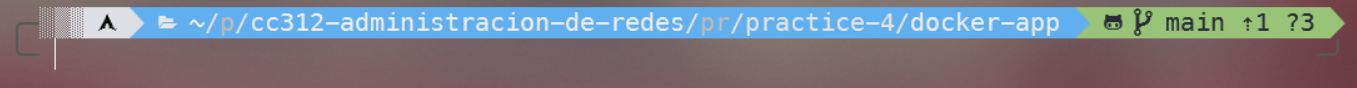
> cd projects/cc312-administracion-de-redes/practices/practice-4/docker-app
> sudo pacman -S docker
warning: docker-1:27.3.1-1 is up to date -- reinstalling
resolving dependencies...
looking for conflicting packages...

Packages (1) docker-1:27.3.1-1

Total Installed Size: 108.35 MiB
Net Upgrade Size:      0.00 MiB

:: Proceed with installation? [Y/n] n
> sudo systemctl start docker

```



2. Estructura del Proyecto

Creo la siguiente estructura de directorios dentro de mi carpeta de la pc4:

```

docker-app/
├── api1/
│   ├── Dockerfile
│   ├── requirements.txt
│   └── app.py
├── api2/
│   ├── Dockerfile
│   ├── requirements.txt
│   └── app.py
├── db/
│   └── init.sql
└── docker-compose.yml

```

3. Implementación de los Servicios

Estaré usando FastAPI para las APIs.

API 1 (Inicio)

```

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import httpx

app = FastAPI()

class User(BaseModel):
    user: str

@app.post("/count")
async def count_user(user_data: User):

```



```

    async with httpx.AsyncClient() as client:
        try:
            response = await client.post(
                'http://api2:5001/register',
                json={'user': user_data.user}
            )
            return response.json()
        except httpx.RequestError:
            raise HTTPException(status_code=500, detail="Error al comunicarse con
API 2")

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=5000)

```

```

FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5000"]

```

```

fastapi
uvicorn
httpx
pydantic

```

API 2 (Registro)

```

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import asyncpg
from typing import Dict

app = FastAPI()

class User(BaseModel):
    user: str

async def get_db_pool():
    return await asyncpg.create_pool(
        host="db",
        database="userdb",
        user="postgres",
        password="postgres"
    )

```



```

@app.on_event("startup")
async def startup():
    app.state.pool = await get_db_pool()

@app.on_event("shutdown")
async def shutdown():
    await app.state.pool.close()

@app.post("/register")
async def register_user(user_data: User) -> Dict[str, int]:
    async with app.state.pool.acquire() as conn:
        # Inserto el usuario
        await conn.execute(
            "INSERT INTO users (username) VALUES ($1)",
            user_data.user
        )
        # Cuento ocurrencias
        count = await conn.fetchval(
            "SELECT COUNT(*) FROM users WHERE username = $1",
            user_data.user
        )
        return {"count": count}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=5001)

```

```

FROM python:3.9-slim
WORKDIR /app
RUN apt-get update && apt-get install -y curl
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5001"]

```

```

fastapi
uvicorn
asynpg
pydantic

```

Base de Datos

```

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100) NOT NULL,

```

```

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

Docker Compose

```

version: '3.8'

services:
  api1:
    build: ./api1
    ports:
      - "5000:5000"
    depends_on:
      api2:
        condition: service_healthy

  api2:
    build: ./api2
    ports:
      - "5001:5001"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://localhost:5001/docs" ]
      interval: 10s
      timeout: 5s
      retries: 5

  db:
    image: postgres:13
    environment:
      POSTGRES_DB: userdb
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U postgres" ]
      interval: 5s
      timeout: 5s
      retries: 5

```

4. Despliegue

Para desplegar los servicios, ejecuto:

```
cd docker-app
docker compose up --build
```

Los servicios se iniciarán en el siguiente orden:

1. La base de datos se inicia y espera hasta estar lista para aceptar conexiones
2. API2 se inicia una vez que la base de datos está saludable
3. API1 se inicia después de que API2 esté funcionando correctamente

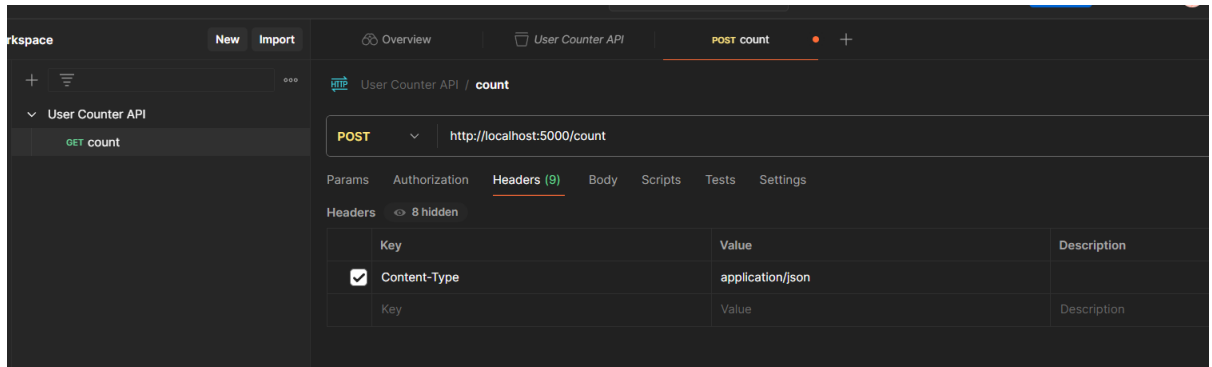
```
Arch
> docker compose up --build
WARN[0000] /home/andre/projects/cc312-administracion-de-redes/practices/practice-4/docker-
app/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please re
move it to avoid potential confusion
[+] Running 0/0
[+] Running 0/1 Building 0.1s
[+] Building 0.2s (11/12) docker:default
[+] Building 0.5s (20/20) FINISHED docker:default
=> [api2 internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 260B 0.0s
=> [api1 internal] load metadata for docker.io/library/python:3.9-slim 0.0s
=> [api2 internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [api1 1/5] FROM docker.io/library/python:3.9-slim 0.0s
=> [api2 internal] load build context 0.0s
=> => transferring context: 1.50kB 0.0s
=> CACHED [api1 2/5] WORKDIR /app 0.0s
=> CACHED [api2 3/6] RUN apt-get update && apt-get install -y curl 0.0s
=> CACHED [api2 4/6] COPY requirements.txt . 0.0s
=> CACHED [api2 5/6] RUN pip install -r requirements.txt 0.0s
=> CACHED [api2 6/6] COPY . . 0.0s
=> [api2] exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:463426cd3b5cac54ad4363ffb57fa26bae7e795f0ac4ef565da5c39 0.0s
=> => naming to docker.io/library/docker-app-api2 0.0s
=> [api2] resolving provenance for metadata file 0.0s
```

```
db-1 | 2024-12-01 21:19:01.530 UTC [1] LOG: starting PostgreSQL 13.17 (Debian 13.17-1.
pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
db-1 | 2024-12-01 21:19:01.530 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port
5432
db-1 | 2024-12-01 21:19:01.530 UTC [1] LOG: listening on IPv6 address ":::", port 5432
db-1 | 2024-12-01 21:19:01.538 UTC [1] LOG: listening on Unix socket "/var/run/postgre
sql/.s.PGSQL.5432"
db-1 | 2024-12-01 21:19:01.545 UTC [27] LOG: database system was shut down at 2024-12-
01 21:18:26 UTC
db-1 | 2024-12-01 21:19:01.553 UTC [1] LOG: database system is ready to accept connect
ions
api2-1 | INFO: Started server process [1]
api2-1 | INFO: Waiting for application startup.
api2-1 | INFO: Application startup complete.
api2-1 | INFO: Uvicorn running on http://0.0.0.0:5001 (Press CTRL+C to quit)
api2-1 | INFO: 127.0.0.1:59658 - "GET /docs HTTP/1.1" 200 OK
api1-1 | INFO: Started server process [1]
api1-1 | INFO: Waiting for application startup.
api1-1 | INFO: Application startup complete.
api1-1 | INFO: Uvicorn running on http://0.0.0.0:5000 (Press CTRL+C to quit)
api2-1 | INFO: 127.0.0.1:46462 - "GET /docs HTTP/1.1" 200 OK
```

5. Pruebas

Para probar el sistema, voy a usar Postman:

1. Abro Postman y creo una nueva colección llamada "User Counter API"
2. Dentro de la colección, creo una nueva solicitud POST
3. Configuro la URL: `http://localhost:5000/count`
4. En la pestaña "Headers", agrego:
 - Key: `Content-Type`
 - Value: `application/json`



5. En la pestaña "Body":
 - Selecciono "raw"
 - Selecciono tipo "JSON"
 - Ingreso el siguiente JSON para cada prueba:

```
{
  "user": "Carlos"
}
```

Realizar múltiples solicitudes cambiando el valor de "user" para probar el contador:

- Primera solicitud con "user": "Carlos"

The screenshot shows a REST client interface for a 'User Counter API'. The request is a POST to 'http://localhost:5000/count' with a JSON body:

```
{  "user": "Carlos"}
```

. The response is a 200 OK status with a JSON body:

```
{  "count": 1}
```

. The status bar indicates a 200 OK response, 133 ms latency, and 136 B body size.

- Segunda solicitud con "user": "Carlos"

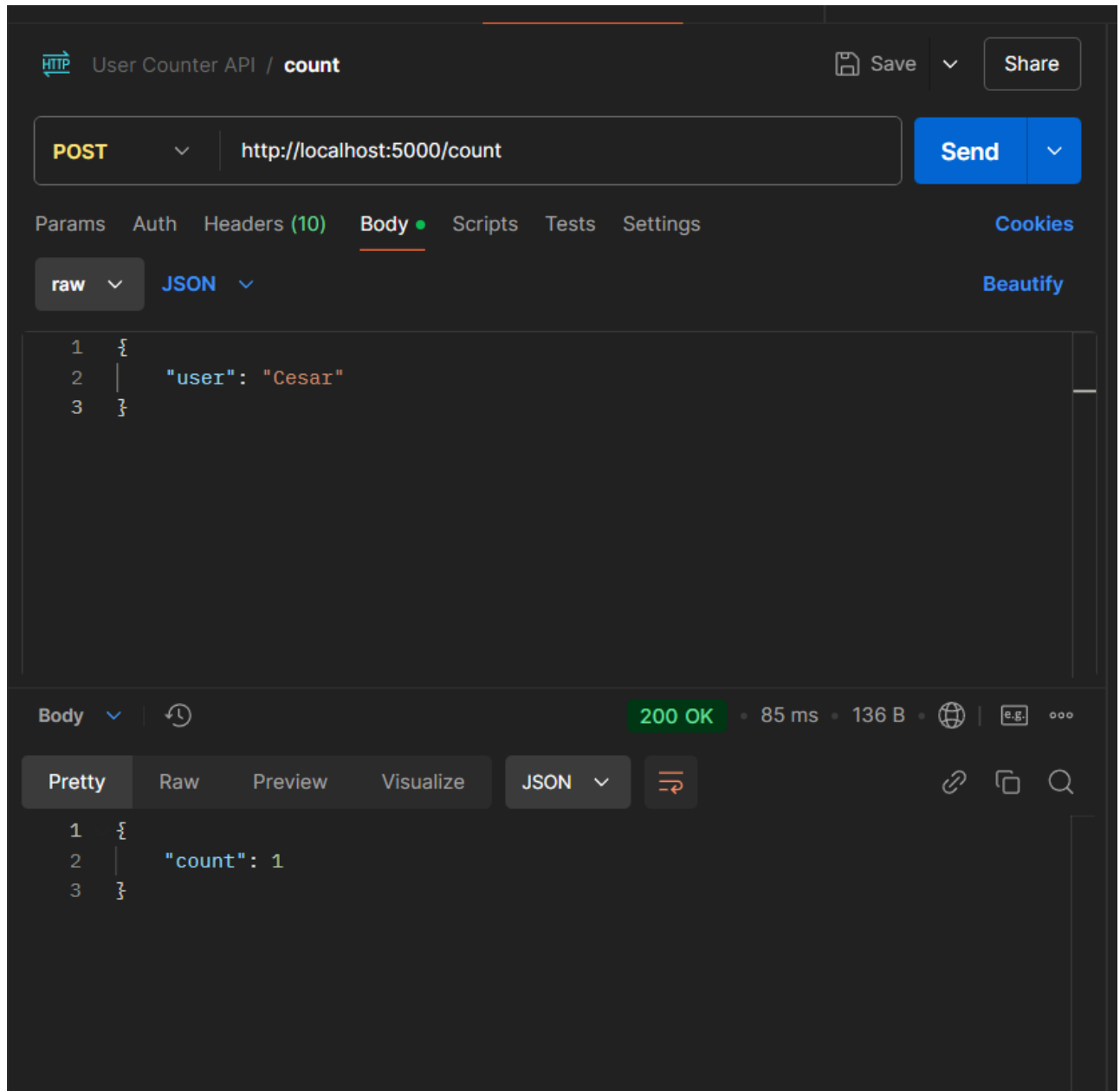
The screenshot shows the response body for the second request. The JSON body is:

```
{  "count": 2}
```

. The status bar indicates a 200 OK response, 133 ms latency, and 136 B body size.

Ahora muestra count: 2

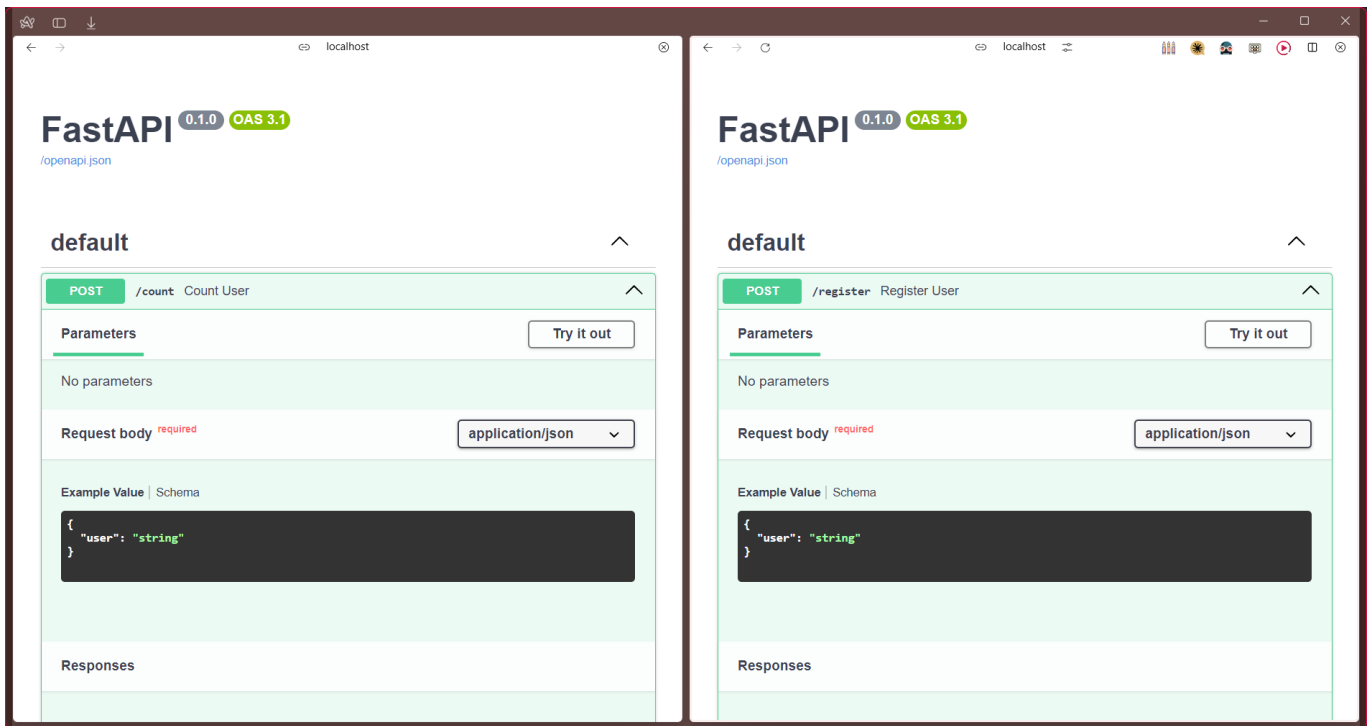
- Nueva solicitud con "user": "Cesar"



Muestra count: 1

También puedo acceder a la documentación automática de la API en:

- API 1: <http://localhost:5000/docs>
- API 2: <http://localhost:5001/docs>



Este sistema implementa una arquitectura de microservicios donde:

- API 1 actúa como punto de entrada y se comunica con API 2
- API 2 maneja la lógica de negocio y la interacción con la base de datos
- La base de datos PostgreSQL almacena y cuenta los registros de usuarios

La comunicación entre servicios se realiza a través de HTTP, y los contenedores están conectados mediante la red de Docker Compose.