Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sistemas Distribuidos

Práctica 7:

“Microservicios”

Alumno:

Villarreal Razo Carlos Gabriel 2022630459

Grupo: 7CM1

Maestro: Carreto Arellano Chadwick

Fecha de realización: 14 de abril de 2025

Fecha de entrega: 16 de abril de 2025

Antecedentes.

En prácticas anteriores se trabajó con servicios web REST, los cuales, facilitaron la comunicación entre clientes y servidores utilizando protocolos abiertos como HTTP y formatos de intercambio de datos como JSON. Esta arquitectura permitió que diferentes aplicaciones pudieran interactuar de forma sencilla y flexible, independientemente de su tecnología subyacente, sin embargo, a medida que se agregan nuevas funcionalidades, los cambios afectan múltiples partes del sistema, haciendo que el desarrollo, el despliegue y el mantenimiento sean más complicados y arriesgados.

Para solucionar estos problemas, surgió el enfoque de microservicios. Los microservicios consisten en dividir una aplicación grande en servicios pequeños, independientes y especializados, que se comunican entre sí a través de protocolos ligeros, como HTTP o mensajería asíncrona —mediante colas como RabbitMQ—. Cada microservicio puede ser desarrollado, desplegado y escalado de manera independiente, lo que incrementa la agilidad, la resiliencia y la mantenibilidad del sistema.

En este modelo, los servicios no sólo están desacoplados a nivel de API, sino también a nivel de infraestructura y bases de datos, permitiendo que cada componente evolucione de manera autónoma. Para organizar y facilitar la comunicación entre múltiples servicios, es común utilizar un API Gateway, que actúa como punto de entrada único para los clientes externos, ocultando la complejidad interna del sistema.

Planteamiento del Problema

En sistemas tradicionales basados en servicios web monolíticos, todas las funcionalidades de una aplicación suelen estar integradas en un mismo servidor o proyecto, lo cual, genera varios problemas conforme el sistema crece. Así, la adición de nuevas funcionalidades puede hacer que el código

se vuelva difícil de mantener, los despliegues se tornen más riesgosos, y el escalado se realice de manera ineficiente, ya que se deben replicar componentes que no necesariamente requieren mayores recursos.

Para evitar estas limitaciones y mejorar la organización, el despliegue y el mantenimiento de los sistemas distribuidos, se planteó la necesidad de dividir la aplicación en microservicios especializados. Por lo cual, en esta práctica, el objetivo es construir un sistema compuesto por servicios independientes, donde cada uno gestione una funcionalidad específica: uno relacionado con usuarios y otro relacionado con pagos.

Además, se busca integrar un API Gateway que sirva como punto de entrada centralizado para las solicitudes de los clientes, organizando el tráfico y redirigiéndolo al microservicio correspondiente de forma transparente. Para lograr una comunicación eficiente y desacoplada entre servicios, se utilizará RabbitMQ, implementando un mecanismo de mensajería asíncrona que permita intercambiar información sin necesidad de mantener conexiones directas entre los servicios.

Propuesta de Solución.

Se busca desarrollar un sistema distribuido basado en la arquitectura de microservicios, dividiendo las funcionalidades de la aplicación en servicios independientes que interactúan entre sí a través de un mecanismo de mensajería asíncrona. Cada microservicio será autónomo, con responsabilidades bien definidas, y podrá ser desplegado o escalado de forma individual.

La solución estará compuesta por los siguientes elementos principales:

- **Servicio de Usuarios**: encargado de gestionar operaciones relacionadas con la creación y consulta de usuarios.

- **Servicio de Pagos**: responsable de procesar pagos asociados a los usuarios registrados.
- **API Gateway**: actuará como único punto de entrada para el cliente, recibiendo las solicitudes y redirigiéndolas al microservicio correspondiente según la operación solicitada.

Además, la comunicación entre los servicios se realizará de forma asíncrona mediante RabbitMQ, permitiendo que los microservicios se mantengan desacoplados y sean más resilientes ante fallos. La transmisión de mensajes a través de permitirá que estos puedan seguir operando incluso si uno de ellos se encuentra temporalmente inactivo.

Cada servicio se implementará utilizando Flask como framework ligero para construir APIs REST internas. A través de esta estructura, se logrará un sistema modular, escalable, y preparado para crecer mediante la adición de nuevos microservicios en el futuro.

Materiales y Métodos.

Hardware.

- Procesador Inter Core i5 de 11ª generación.
- Tarjeta gráfica NVIDIA GTX 1650.
- 16 GB de RAM DDR4.

Software.

- Lenguaje: Java 17.
- Visual Studio Code (VSCode).
- Compilador y ejecución a través de la terminal integrada de VSCode.

Paquetes.

- Flask.
- Requests.

- Jsonify.
- Pika.

Métodos de usuarios.py.

- **enviarMensaje():** permite ingresar mensaje a la cola de comunicación entre microservicios.
- **obtenerUsuario():** permite obtener los datos de un usuario.
- **pagarUsuario():** Permite agregar un usuario a la cola de pagos que será consumida por otro microservicio.

Métodos pagos.py.

- **consumirMensajes()**: permite leer todos los mensajes existentes en la cola de comunicación.

Desarrollo de la Solución.

Para la implementación de esta práctica, se diseñaron tres componentes principales: el servicio de usuarios (usuarios.py), el servicio de pagos (pagos.py) y el API Gateway (apiGateway.py). Cada microservicio se construyó utilizando Flask, exponiendo rutas específicas para realizar operaciones como el registro de usuarios, la consulta de usuarios, el registro de pagos y la consulta de pagos. Internamente, ambos microservicios pueden comunicarse de manera indirecta a través de *RabbitMQ*, utilizando colas de mensajes para desacoplar el flujo de información y permitir que las operaciones se realicen de forma asíncrona.

El API Gateway actúa como único punto de entrada para los clientes, recibiendo solicitudes HTTP y redirigiéndolas al microservicio correspondiente dependiendo del tipo de operación. Así, el cliente no necesita conocer la dirección ni la lógica interna de cada servicio, logrando un mayor nivel de abstracción y simplificación. Además, el uso de colas con *pika* permite manejar eventos de manera desacoplada, lo que hace posible la escalabilidad del sistema y su adaptación para integrar nuevos servicios en el futuro. Con esta

estructura, se simula de manera básica el funcionamiento de una arquitectura moderna de microservicios.

usuarios.py.

```
from flask import Flask, jsonify
import pika
import json

usuarios_app = Flask(__name__)

usuarios = {
    1: {"nombre": "Willis", "saldo": 500},
    2: {"nombre": "Ana", "saldo": 700}
}

def enviarMensaje(user_id):
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()
    channel.queue_declare(queue='pagos')

    mensaje = json.dumps({"user_id": user_id})
    channel.basic_publish(exchange='', routing_key='pagos', body=mensaje)
    print(f"Mensaje enviado a la cola de pagos: {mensaje}")

    connection.close()

@usuarios_app.route("/usuarios/<int:user_id>", methods=["GET"])
def obtenerUsuario(user_id):
    user = usuarios.get(user_id)
    if not user:
        return jsonify({"error": "Usuario no encontrado"}), 404
    return jsonify(user)

@usuarios_app.route("/usuarios/<int:user_id>/pagar", methods=["POST"])
def pagarUsuario(user_id):
    if user_id not in usuarios:
        return jsonify({"error": "Usuario no encontrado"}), 404

    enviarMensaje(user_id)
    return jsonify({"mensaje": "Solicitud de pago enviada", "usuario": usuarios[user_id]})

if __name__ == "__main__":
    usuarios_app.run(port=5000)
```

pagos.py.

```python
from flask import Flask, jsonify
import pika
import json

pagos_app = Flask(__name__)

def procesarPago(user_id):
    print(f"Procesando pago para el usuario {user_id}")

def callback(ch, method, properties, body):
    data = json.loads(body)
    user_id = data["user_id"]
    procesarPago(user_id)
    ch.basic_ack(delivery_tag=method.delivery_tag)

def consumirMensajes():
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()
    channel.queue_declare(queue='pagos')

    channel.basic_consume(queue='pagos', on_message_callback=callback)

    print("Microservicio de pagos esperando mensajes...")
    channel.start_consuming()

@pagos_app.route("/pagos/procesar", methods=["POST"])
def iniciarProcesamiento():
    consumirMensajes()
    return jsonify({"mensaje": "Procesamiento de pagos iniciado"})

if __name__ == "__main__":
    pagos_app.run(port=5001)
```

apiGateway.py.

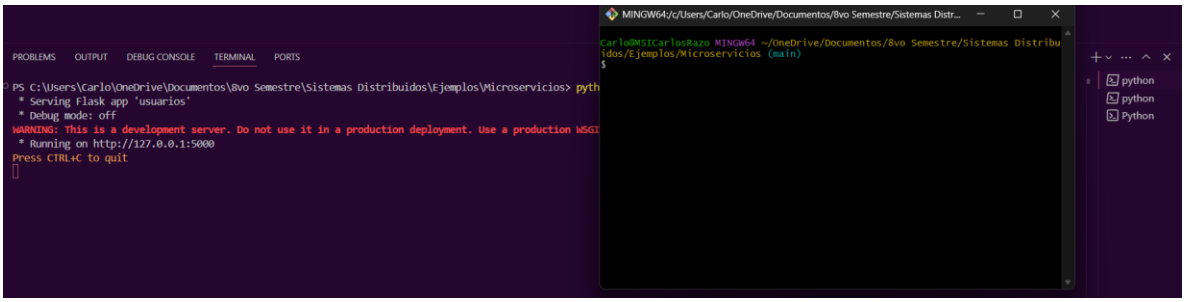
```python
from flask import Flask, jsonify
import requests

gateway_app = Flask(__name__)

SERVICIOUSUARIOS = "http://127.0.0.1:5000"
SERVICIOPAGOS = "http://127.0.0.1:5001"

@gateway_app.route("/usuarios/<int:user_id>", methods=["GET"])
```

```python
def obtenerUsuario(user_id):
    response = requests.get(f"{SERVICIOUSUARIOS}/usuarios/{user_id}")
    return jsonify(response.json())

@gateway_app.route("/usuarios/<int:user_id>/pagar", methods=["POST"])
def pagarUsuario(user_id):
    response = requests.post(f"{SERVICIOUSUARIOS}/usuarios/{user_id}/pagar")
    return jsonify(response.json())

@gateway_app.route("/pagos/procesar", methods=["POST"])
def procesarPagos():
    response = requests.post(f"{SERVICIOPAGOS}/pagos/procesar")
    return jsonify(response.json())

if __name__ == "__main__":
    gateway_app.run(port=5002)
```

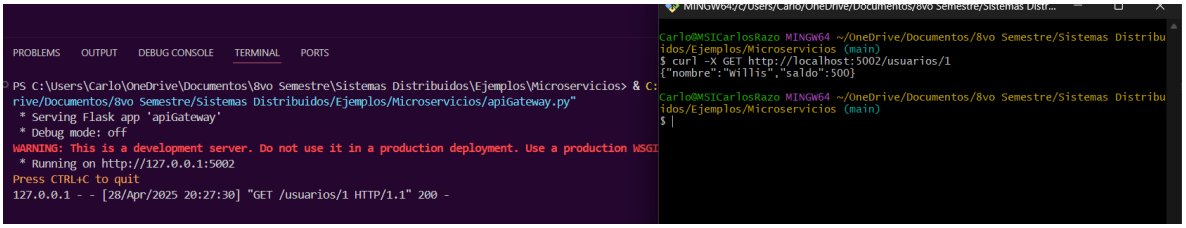
Link del Repositorio de GitHub de la Práctica.

https://github.com/ClsRazo/PracticaMicroservicios.git

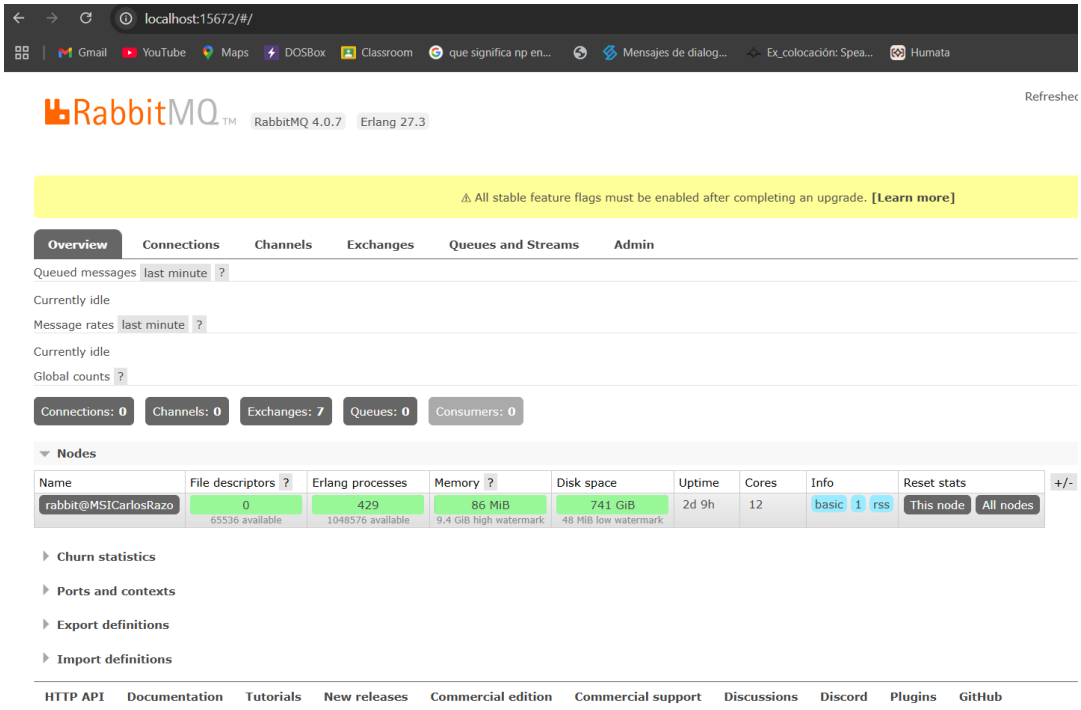
Resultados.



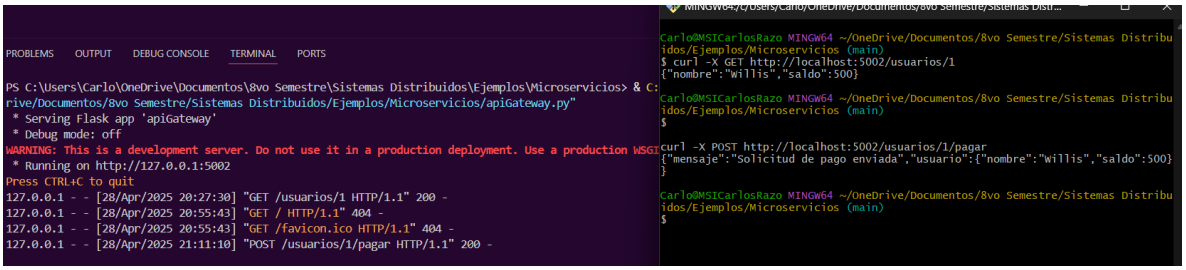
Obtenemos un usuario:



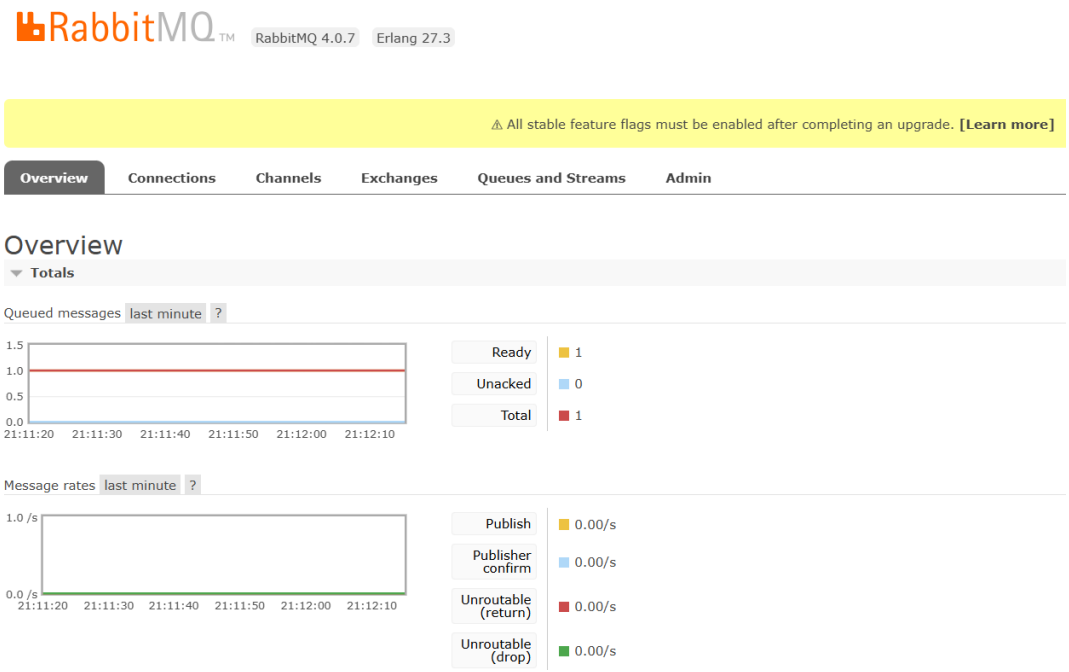
Vista de rabbitMQ antes de empezar a encolar mensajes:



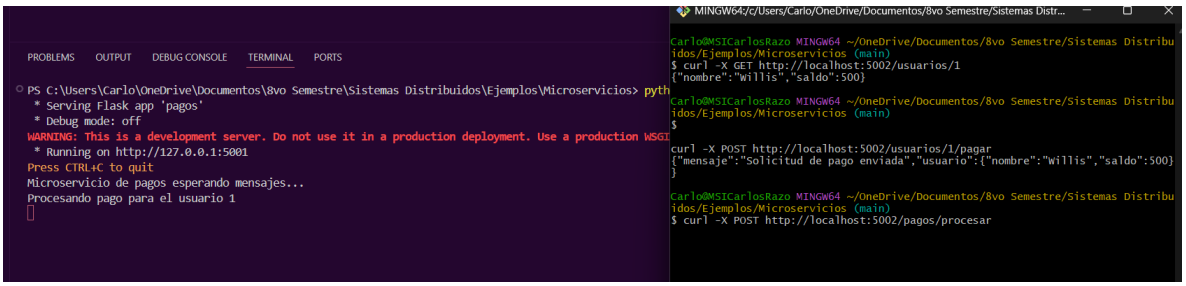
Mandamos un mensaje de pago a la cola:



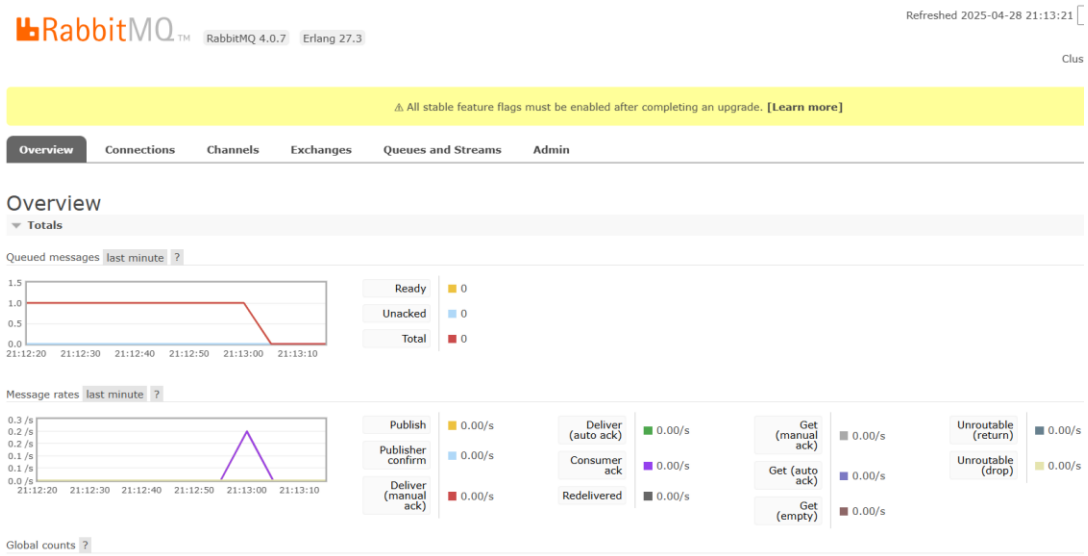
Revisamos la cola de mensajes en rabbitMQ:



Consumimos el mensaje desde el otro microservicio (procesamos el pago):



Vista de rabbitMQ:



Conclusiones.

Derivado de la culminación de la práctica, logré comprender los fundamentos y beneficios de la arquitectura de microservicios, así como su diferencia respecto a los modelos tradicionales de servicios monolíticos. La implementación de servicios independientes para la gestión de usuarios y pagos, junto con un API Gateway que organiza y canaliza las solicitudes, permitió observar cómo se puede construir un sistema más modular, flexible y escalable, donde cada componente puede evolucionar o escalar de forma autónoma.

Además, mediante la integración de RabbitMQ como sistema de mensajería asíncrona, entendí la importancia de desacoplar la comunicación entre servicios, permitiendo que trabajen de manera más resiliente y tolerante a fallos. Por lo cual, esta práctica me permitió aplicar conceptos clave como la separación de responsabilidades, la comunicación mediante colas de mensajes, y la organización de sistemas distribuidos modernos, sentando una base sólida para el desarrollo de aplicaciones en arquitecturas de microservicios reales.