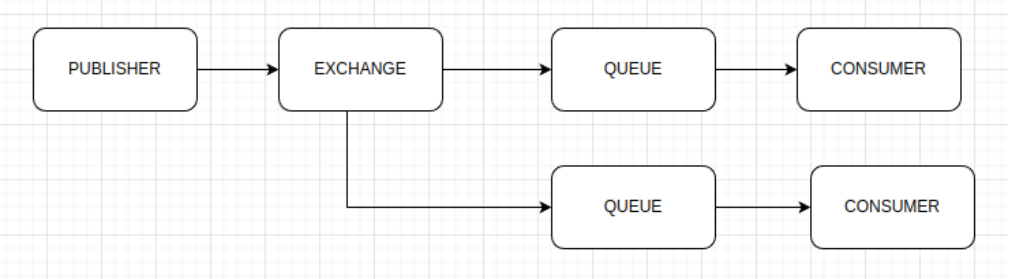
Actividades:
git : https://github.com/Pidual/Taller-RabbitMQ
1. Revisión conceptual:

- **¿Qué es RabbitMQ y cuál es su función en una arquitectura distribuida?**
  Que RabbitMQ es un software de mensajería que permite enviar y recibir mensajes entre aplicaciones
- **¿Qué ventajas ofrece frente a llamadas HTTP directas entre servicios?**
  Cuenta con ruteado inteligente no se tienen que codificar manualmente la dirección, También su balanceador de cargas el cual detecta múltiples instancias, además de su escalabilidad.
- **¿Qué son las colas, exchanges, publishers y consumers?**
  - **Colas:** Es donde los mensajes se almacenan temporalmente hasta que algún consumer los procese.
    Las colas no envían los mensajes por sí mismas, esperan que un consumer los pida.
  - **Exchanges:** Es quien recibe los mensajes del publisher y decide a qué cola enviarlos, basándose en ciertas reglas (binding y routing key).
  - **Publishers:** Es quien envía los mensajes. No envía mensajes directamente a una cola, sino a un exchange.
  - **Consumers:** Es quien recibe y procesa los mensajes desde una cola.



2. Análisis del sistema actual:
identificar en la arquitectura del parcial actual:

- **¿Quién produce eventos?**
  El cliente uno y cliente dos

- **¿Quién consume estos eventos?**
  registro-app

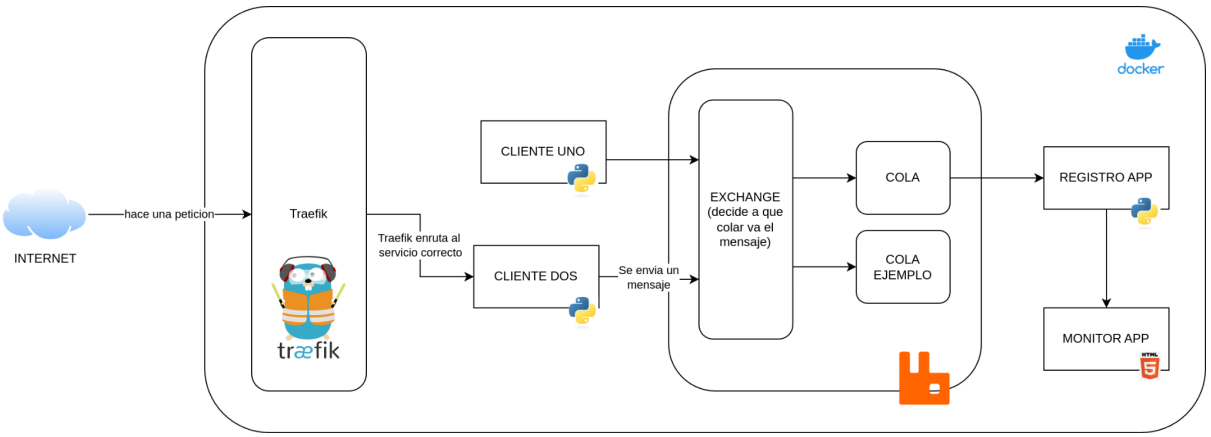
- **¿Dónde existen acoplamientos directos que podrían desacoplarse?**
  En cliente app donde se quema la dirección del servicio registro

```
SERVICE_ID = os.environ.get('SERVICE_ID', 'unknown')
API_REGISTRO_URL = "http://registro-app:5000/registro"
```

3. Propuesta de rediseño:

- Cada equipo debe presentar un esquema actualizado donde
- Los servicio-cliente-X ya no llamen directamente al servicio-analiticas, sino publiquen mensajes a una cola RabbitMQ.
- El REGISTRO APP actúa como consumidor de esa cola, procesando los eventos recibidos.
- El MONITOR continúa consultando el estado actual del servicio de analíticas (sin conectarse a RabbitMQ directamente).

Incluir al menos:
Nombre de exchange y colas.
Formato de mensaje.
Lógica del consumidor (cómo procesa y almacena los conteos).



4. Inicio de implementación:
Usar una imagen oficial de RabbitMQ en el docker-compose.yml.
Configurar la conexión del servicio-cliente-X para enviar mensajes a la cola.
Implementar un consumidor básico en servicio-analiticas que escuche los mensajes y actualice los contadores en memoria.

1.¿Qué beneficio aporta RabbitMQ en comparación con el modelo de solicitud directa HTTP?
Esto nos permite desacoplar los servicios y es más escalable, por que podemos ir agregando más consumidores sin cambiar el código.

2. ¿Qué problemas podrían surgir si se caen algunos servicios?
Sin RabbitMQ (HTTP directo)si el consumidor se cae falla inmediatamente. mientras que con rabbitMQ,si el consumidor se cae, los mensajes quedan en la cola y se procesarán cuando vuelva a estar disponible.

3. ¿Cómo ayuda RabbitMQ a mejorar la resiliencia del sistema?
Este actúa como un buffer permitiendo que continúe la operación parcial del sistema, aunque un servicio esté temporalmente fuera de línea, gracias a su persistencia de mensaje, cola.

4. ¿Cómo cambiaría la lógica de escalabilidad con esta nueva arquitectura?
En lugar de tener que escalar tanto los servicios emisores como receptores de forma coordinada (como ocurre en el modelo de comunicación directa por HTTP), ahora podemos escalar los consumidores de manera independiente.

5. ¿Qué formato de mensaje es más conveniente y por qué (JSON, texto plano, etc.)?

- Es **estructurado y legible**, lo que facilita el análisis, la depuración y la integración con otros servicios.
- Es **ampliamente soportado** por librerías en múltiples lenguajes.
- Permite incluir múltiples datos de forma clara.

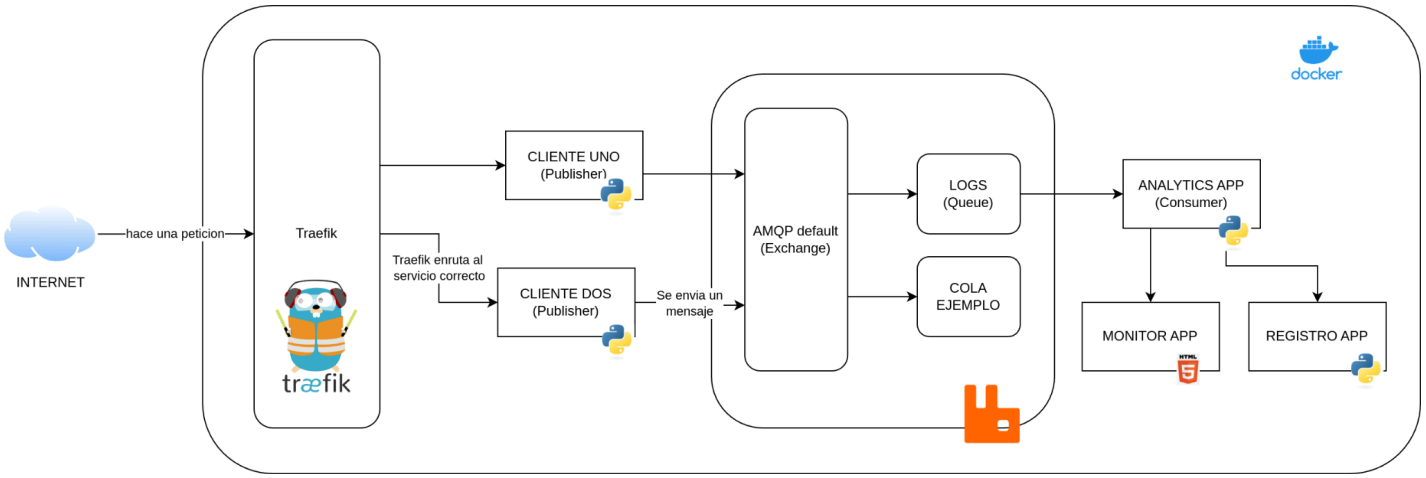
Requisitos técnicos mínimos para el rediseño:
RabbitMQ como contenedor accesible dentro de la red del proyecto.

Conexión desde cliente usando una librería adecuada (por ejemplo, amqplib en Node.js).

Al menos una cola configurada y en uso.
Publicador funcional desde un cliente.
Consumidor funcional desde el servicio analítico.

- **Entrega al finalizar:**

Diagrama de arquitectura con RabbitMQ incluido



Avance funcional: publicación y consumo básico.
https://github.com/Pidual/Taller-RabbitMQ

Documento corto por equipo explicando:
Cambios hechos respecto a la versión original:

- Agregamos el servicio de rabbitMQ en docker compose
- Asignamos una clave un usuario y una contraseña
- Agregamos el servicio analytics app que es el encargado de responder los mensajes por la terminal

Justificación del diseño propuesto.
RabbitMQ, es como dejar notas en una bandeja: uno deja el mensaje y el otro lo lee cuando puede. Esto permite cambiar, escalar o mejorar cada servicio sin preocuparse si el otro está despierto o no

Si algo se cae, no pasa nada. Los mensajes no se pierden, quedan guardados como en una caja fuerte. Cuando el servicio vuelve, puede seguir leyendo desde donde se quedó.
Si hay mucho trabajo, RabbitMQ lo reparte entre varios “ayudantes” (consumidores).

Así no se sobrecarga nadie y el trabajo avanza más rápido. No hace falta meter mano para balancear, él solito organiza quién hace qué. El consumidor intenta varias veces si algo falla (como alguien insistente tocando el timbre). Además, todo queda bien anotado en logs, para saber qué pasó y cuándo.

Lecciones aprendidas sobre RabbitMQ: Durante el desarrollo se aprendió a cómo controlar las peticiones, de mejor manera adicionalmente. se pudo entender más de la estructura del docker compose, también como es el manejo de las peticiones por orden.