Reto 3 – Comunicación Asincrónica y Eventos

Contexto

Este reto continúa el desarrollo del **sistema de onboarding y offboarding de empleados**. En retos anteriores se construyeron:

- Un servicio básico de empleados con Docker (Reto 1)
- Orquestación con Docker Compose, persistencia y comunicación REST entre servicios (Reto 2)

En este tercer reto se avanzará hacia una **arquitectura orientada a eventos**, incorporando:

- Un **message broker** como infraestructura de mensajería
- **Publicación de eventos** desde el servicio existente
- **Dos nuevos servicios** que reaccionan a eventos de forma autónoma
- El patrón **fan-out**: un solo evento dispara acciones en múltiples servicios

Objetivo

Incorporar comunicación asincrónica basada en eventos al sistema, de forma que los microservicios se comuniquen de manera desacoplada a través de un message broker. Al finalizar este reto, el sistema debe demostrar que un solo evento (como la creación de un empleado) desencadena reacciones automáticas e independientes en diferentes microservicios.

Requisitos generales

La solución desarrollada debe cumplir con los siguientes requisitos:

- Incorporar un **message broker** al `docker-compose.yml`
- El servicio de Empleados debe **publicar eventos** cuando se crea o elimina un empleado

- Implementar un **Servicio de Notificaciones** que consuma eventos y registre las notificaciones enviadas
- Implementar un **Servicio de Gestión de Perfiles** que consuma eventos y gestione el perfil del empleado
- Los estudiantes deben **investigar y seleccionar** el message broker a utilizar
- Documentar los endpoints de los nuevos servicios utilizando **OpenAPI (Swagger)**
- Permitir el despliegue completo mediante un único comando

1. Message Broker

Investigación y selección

Cada equipo debe investigar al menos **tres** de las siguientes opciones de message broker y seleccionar **una** para su implementación:

Broker	Características clave
RabbitMQ	Protocolo AMQP, exchanges y colas, management UI, amplia documentación
Apache Kafka	Alto throughput, persistencia de mensajes, ideal para streaming
Redis Streams	Ligero, bajo overhead, ideal si ya se usa Redis
NATS	Ultra-ligero, alta performance, cloud-native

Requisitos

- Incluir el broker como servicio en el `docker-compose.yml`
- Configurar la interfaz de administración del broker (si está disponible)
- Documentar en el README **por qué se eligió** ese broker sobre las alternativas

Ejemplo de configuración (RabbitMQ como referencia)

```yaml
services:
  # ... servicios existentes del Reto 2 ...

  message-broker:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"     # Puerto del broker
      - "15672:15672"   # Interfaz de administración
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: admin
    networks:
      - microservices-network
```

> **Importante:** Este es solo un ejemplo de referencia con RabbitMQ. Si elige otro broker, la configuración será diferente. El estudiante debe investigar y configurar el broker seleccionado.

2. Publicación de eventos desde el Servicio de Empleados

Evolución del servicio

El Servicio de Empleados (desarrollado en el Reto 2) debe evolucionar para **publicar eventos** cuando ocurran acciones relevantes:

Acción	Evento a publicar	Datos del evento
Crear un empleado	`empleado.creado`	`{ id, nombre, email, departamentoId, fechaIngreso }`
Eliminar un empleado	`empleado.eliminado`	`{ id, nombre, email }`

Comportamiento esperado

1. Cuando se ejecuta `POST /empleados` exitosamente:
   - Se persiste el empleado en la base de datos (como en el Reto 2)
   - Se valida el departamento (como en el Reto 2)
   - **Nuevo:** Se publica el evento `empleado.creado` en el broker
2. Cuando se ejecuta `DELETE /empleados/{id}` (nuevo endpoint):
   - Se elimina o marca como inactivo el empleado
   - **Nuevo:** Se publica el evento `empleado.eliminado` en el broker

Consideraciones

- La publicación del evento debe ocurrir **después** de que la operación en base de datos sea exitosa
- Si la publicación del evento falla, el servicio debe registrar el error pero **no revertir** la operación de base de datos
- El formato del evento (JSON) debe estar documentado

3. Servicio de Notificaciones (nuevo servicio)

Un microservicio que **solo consume eventos**. Este servicio no es invocado por REST desde otros servicios, sino que reacciona autónomamente a los eventos del sistema.

Eventos que debe consumir

Evento	Acción
`empleado.creado`	Registrar y simular envío de email de bienvenida
`empleado.eliminado`	Registrar y simular notificación de desvinculación

Simulación de notificaciones

Para simplificar, la "notificación" se simula mediante un **log estructurado** en la consola del servicio:

```
[NOTIFICACIÓN] Tipo: BIENVENIDA | Para: juan@empresa.com | Mensaje: "Bienvenido Juan Pér
[NOTIFICACIÓN] Tipo: DESVINCULACIÓN | Para: juan@empresa.com | Mensaje: "Su cuenta ha si
```

Registro de notificaciones

El servicio debe mantener un historial de notificaciones enviadas:

Método	Ruta	Descripción
`GET`	`/notificaciones`	Lista todas las notificaciones registradas
`GET`	`/notificaciones/{empleadoId}`	Lista notificaciones de un empleado específico

Estructura de una notificación

```
{
  "id": "string",
  "tipo": "BIENVENIDA | DESVINCULACION",
  "destinatario": "string (email)",
  "mensaje": "string",
  "fechaEnvio": "datetime",
  "empleadoId": "string"
}
```

Contenerización y base de datos

- Debe crear un Dockerfile para el servicio
- Debe tener su propia base de datos para el historial de notificaciones
- La configuración debe obtenerse de variables de entorno

> **Bonus (opcional):** Implementar envío de email real utilizando un servicio SMTP (como Mailhog en Docker para pruebas locales).

4. Servicio de Gestión de Perfiles (nuevo servicio)

Un microservicio que **combina comunicación asincrónica y sincrónica**: consume eventos para crear datos automáticamente, y expone endpoints REST para que los datos puedan ser consultados y actualizados.

Evento que debe consumir

Evento	Acción
`empleado.creado`	Crear un perfil por defecto para el nuevo empleado

Perfil por defecto

Cuando se recibe el evento `empleado.creado`, se crea automáticamente un perfil con los datos básicos:

```
{
  "id": "string",
  "empleadoId": "string",
  "nombre": "string (del evento)",
  "email": "string (del evento)",
  "telefono": "",
  "direccion": "",
  "ciudad": "",
  "biografia": "",
  "fechaCreacion": "datetime"
}
```

Endpoints requeridos

Método	Ruta	Descripción
`GET`	`/perfiles/{empleadoId}`	Consulta el perfil de un empleado
`PUT`	`/perfiles/{empleadoId}`	Actualiza el perfil (teléfono, dirección, biografía, etc.)
`GET`	`/perfiles`	Lista todos los perfiles

Respuestas esperadas

- **Consulta exitosa:** `200 OK` con el perfil del empleado
- **Perfil no existe:** `404 Not Found` con mensaje descriptivo
- **Actualización exitosa:** `200 OK` con el perfil actualizado

Contenerización y base de datos

- Debe crear un Dockerfile para el servicio
- Debe tener su propia base de datos
- La configuración debe obtenerse de variables de entorno

5. Pruebas del Sistema

Verificar el funcionamiento completo del flujo asincrónico:

Flujo de prueba sugerido

1. Iniciar todos los servicios: `docker-compose up --build`
2. Verificar que el broker está activo y accesible (management UI)
3. Crear un departamento (como en el Reto 2):

```
curl -X POST http://localhost:8081/departamentos \
  -H "Content-Type: application/json" \
  -d '{"id": "IT", "nombre": "Tecnología", "descripcion": "Departamento de TI"}'
```

4. Crear un empleado:

```
curl -X POST http://localhost:8080/empleados \
  -H "Content-Type: application/json" \
  -d '{"id": "E001", "nombre": "Juan Pérez", "email": "juan@empresa.com", "departame
```

5. **Verificar que el evento fue procesado:**
   - Consultar el perfil creado automáticamente: `GET http://localhost:8083/perfiles/E001`
   - Consultar la notificación registrada: `GET http://localhost:8084/notificaciones/E001`
   - Revisar los logs del servicio de notificaciones para ver la notificación simulada
6. Actualizar el perfil del empleado:

```
curl -X PUT http://localhost:8083/perfiles/E001 \
  -H "Content-Type: application/json" \
  -d '{"telefono": "3001234567", "ciudad": "Armenia", "biografia": "Ingeniero de sis
```

7. Eliminar un empleado y verificar la notificación:

```
curl -X DELETE http://localhost:8080/empleados/E001
# Verificar notificación de desvinculación
curl http://localhost:8084/notificaciones/E001
```

8. Reiniciar los contenedores y verificar que los datos persisten

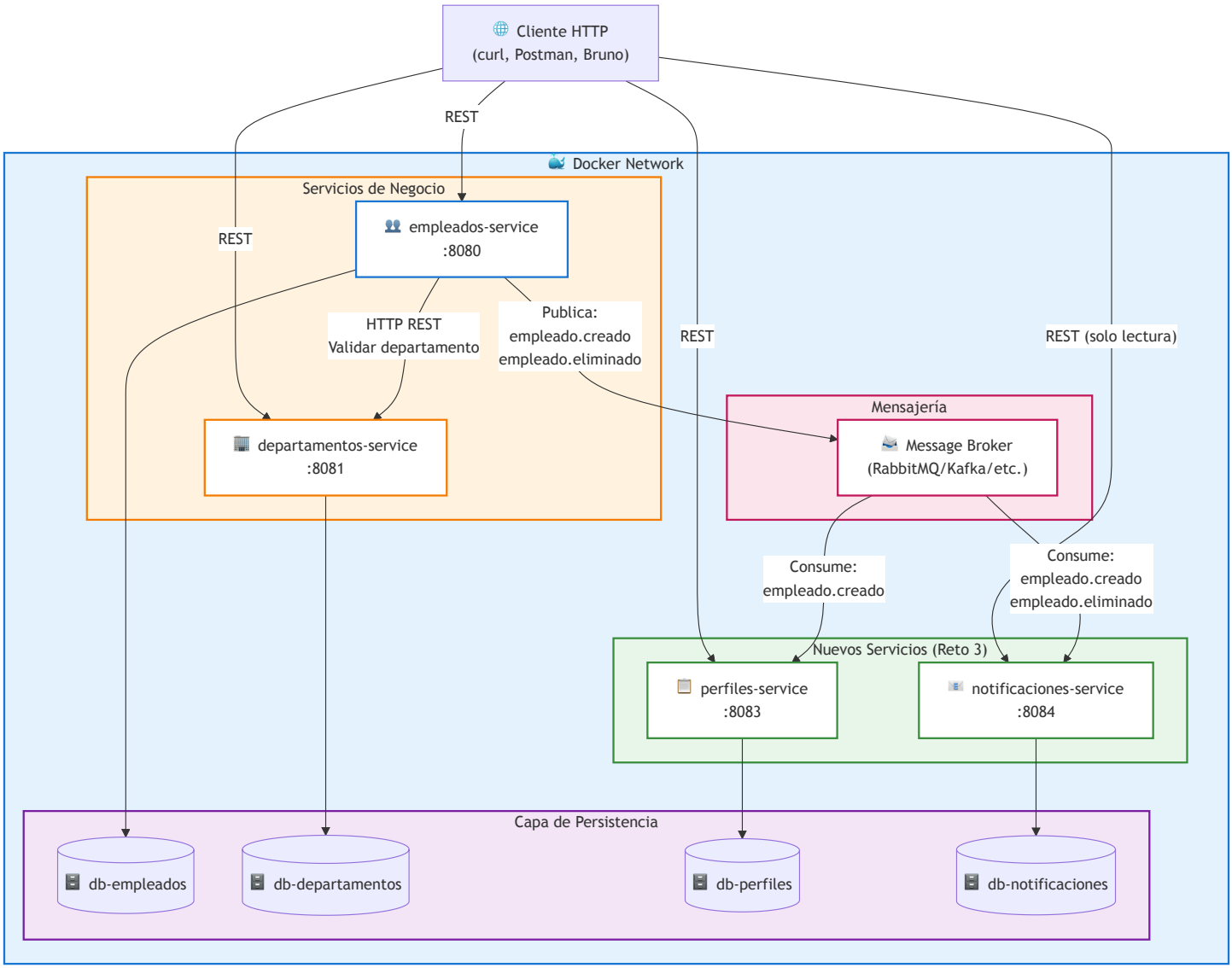
Entregables

- Versione adecuadamente su código en repositorios de GitHub
- Cree un README.md actualizado que incluya:
  - Justificación de la elección del message broker
  - Instrucciones de despliegue del sistema completo
  - Documentación de los eventos (nombre, estructura, productores y consumidores)
  - Instrucciones de prueba para verificar el flujo asincrónico

Consideraciones

- Cada servicio debe tener su propio `Dockerfile`
- Cada servicio debe tener su propia base de datos
- El servicio de Notificaciones debe ser **puramente reactivo** (no es llamado por otros servicios via REST)
- El servicio de Perfiles combina ambos estilos: reactivo (consume eventos) + REST (expone endpoints)

Diagrama de Arquitectura Esperada



Criterios de Evaluación

El reto se evalúa sobre **5 puntos**, correspondiendo **1 punto** a cada uno de los elementos principales:

#	Elemento	Valor	Aspectos a evaluar
1	**Message Broker**	1.0	Configuración correcta en compose, investigación documentada, justificación de la elección
2	**Publicación de eventos**	1.0	El servicio de Empleados publica eventos correctamente al crear/eliminar empleados

#	Elemento	Valor	Aspectos a evaluar
3	**Servicio de Notificaciones**	1.0	Consumo de eventos, registro de notificaciones, log de simulación, endpoints de consulta y **documentación OpenAPI**
4	**Servicio de Perfiles**	1.0	Consumo de evento, creación de perfil por defecto, endpoints REST funcionales y **documentación OpenAPI**
5	**Pruebas y documentación**	1.0	Flujo completo demostrable, documentación de eventos, README actualizado, correcto funcionamiento de **Swagger UI**

> **Nota:** La documentación (README.md) y el versionamiento adecuado en GitHub son requisitos transversales que afectan la evaluación de cada elemento.