

Universidad de Almería

Máster en Ingeniería Informática

Integración de Tecnologías y Servicios Informáticos

Práctica 5

Orquestación de Microservicios con Docker Compose

Autor: Johan Eduardo Cala Torra

Fecha: 8 de diciembre de 2025

Índice

1. Introducción	2
1.1. Objetivos de Aprendizaje	2
2. Fundamentos Teóricos	3
2.1. Docker Compose	3
2.2. RabbitMQ y Colas de Mensajes	3
2.3. Arquitectura del Sistema	3
3. Flujo de Trabajo Guiado: Sistema de Gestión de Tareas	4
3.1. Estructura del Proyecto	4
3.2. Paso 1: Configuración de Docker Compose	4
3.3. Paso 1: Configuración de Docker Compose	4
3.3.1. Servicio Web (API Flask)	5
3.3.2. Servicio Worker (Procesador Asíncrono)	6
3.3.3. Servicio Notifier (Notificaciones)	7
3.3.4. Servicio Error Handler (Dead Letter Queue)	8
3.3.5. Servicio Database (PostgreSQL)	9
3.3.6. Servicio Message Queue (RabbitMQ)	9
3.4. Resumen de Servicios Configurados	9
3.5. Paso 2: API Flask (Servicio Web)	10
3.6. Paso 3: Worker (Consumidor de Mensajes)	11
3.7. Paso 4: Verificación del Sistema	12
3.7.1. Contenedores en Ejecución	12
3.7.2. Creación de Tareas	12
3.7.3. Base de Datos PostgreSQL	12
3.7.4. RabbitMQ Management UI	12
4. Ejercicio 1: Endpoint para Completar Tareas	13
4.1. Implementación	13
4.2. Prueba del Endpoint	13
5. Ejercicio 2: Servicio Notifier	14
5.1. Arquitectura	14
5.2. Implementación	14
6. Ejercicio 3: Dead Letter Queue	15
6.1. Configuración del DLX	15
6.2. Error Handler	15
7. Credenciales de Acceso	16
7.1. RabbitMQ Management UI	16
7.2. PostgreSQL	16
7.3. API Flask	16
8. Conclusiones	17
8.1. Lecciones Aprendidas	17
8.2. Comandos Útiles	17

1. Introducción

Esta práctica se centra en la **orquestación de microservicios** utilizando Docker Compose. Se construye un sistema distribuido completo que incluye una API REST, workers asíncronos, base de datos PostgreSQL y un broker de mensajes RabbitMQ.

1.1. Objetivos de Aprendizaje

- **Docker Compose:** Orquestar múltiples contenedores como un sistema unificado.
- **Comunicación Asíncrona:** Implementar colas de mensajes con RabbitMQ.
- **Persistencia de Datos:** Configurar PostgreSQL con volúmenes Docker.
- **Microservicios:** Diseñar servicios desacoplados que se comunican mediante eventos.
- **Dead Letter Queue:** Manejar mensajes fallidos de forma robusta.

2. Fundamentos Teóricos

2.1. Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones Docker multi-contenedor. Utiliza un archivo YAML para configurar los servicios de la aplicación.

Conceptos clave:

- **Services:** Contenedores que forman la aplicación
- **Networks:** Redes virtuales para comunicación entre contenedores
- **Volumes:** Almacenamiento persistente para datos
- **depends_on:** Define dependencias entre servicios

2.2. RabbitMQ y Colas de Mensajes

RabbitMQ es un broker de mensajes que implementa el protocolo AMQP. Permite la comunicación asíncrona entre servicios.

Patrones implementados:

- **Work Queues:** Distribución de tareas entre workers
- **Dead Letter Exchange:** Manejo de mensajes fallidos
- **Message Acknowledgment:** Confirmación de procesamiento

2.3. Arquitectura del Sistema

El sistema implementado consta de 6 servicios:

1. **web:** API Flask con endpoints REST
2. **worker:** Consumidor de mensajes de RabbitMQ
3. **notifier:** Servicio de notificaciones
4. **error-handler:** Procesador de Dead Letter Queue
5. **db:** Base de datos PostgreSQL
6. **mq:** Broker RabbitMQ

3. Flujo de Trabajo Guiado: Sistema de Gestión de Tareas

3.1. Estructura del Proyecto

```
task-manager-service/  
  docker-compose.yml  
  web/  
    app.py  
    requirements.txt  
    Dockerfile  
  worker/  
    worker.py  
    requirements.txt  
    Dockerfile  
  notifier/  
    notifier.py  
    requirements.txt  
    Dockerfile  
  error-handler/  
    error_handler.py  
    requirements.txt  
    Dockerfile
```

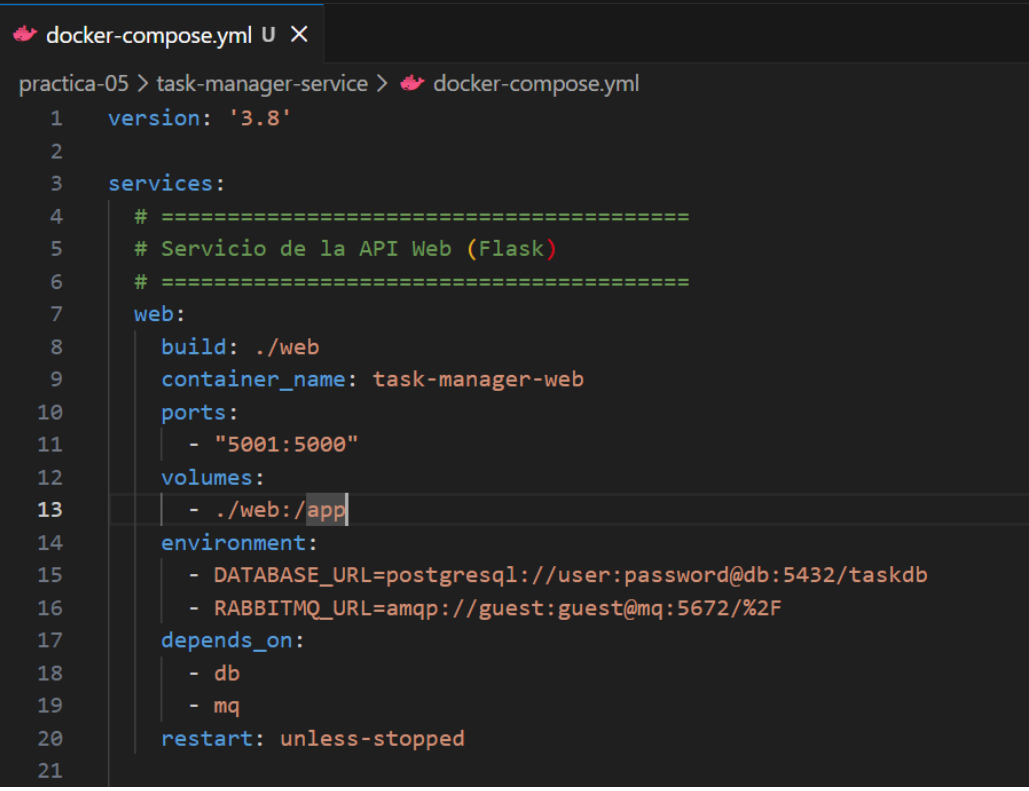
3.2. Paso 1: Configuración de Docker Compose

El archivo `docker-compose.yml` define todos los servicios del sistema:

3.3. Paso 1: Configuración de Docker Compose

El archivo `docker-compose.yml` define todos los servicios del sistema. A continuación se muestra la configuración de cada uno de los 6 servicios:

3.3.1. Servicio Web (API Flask)

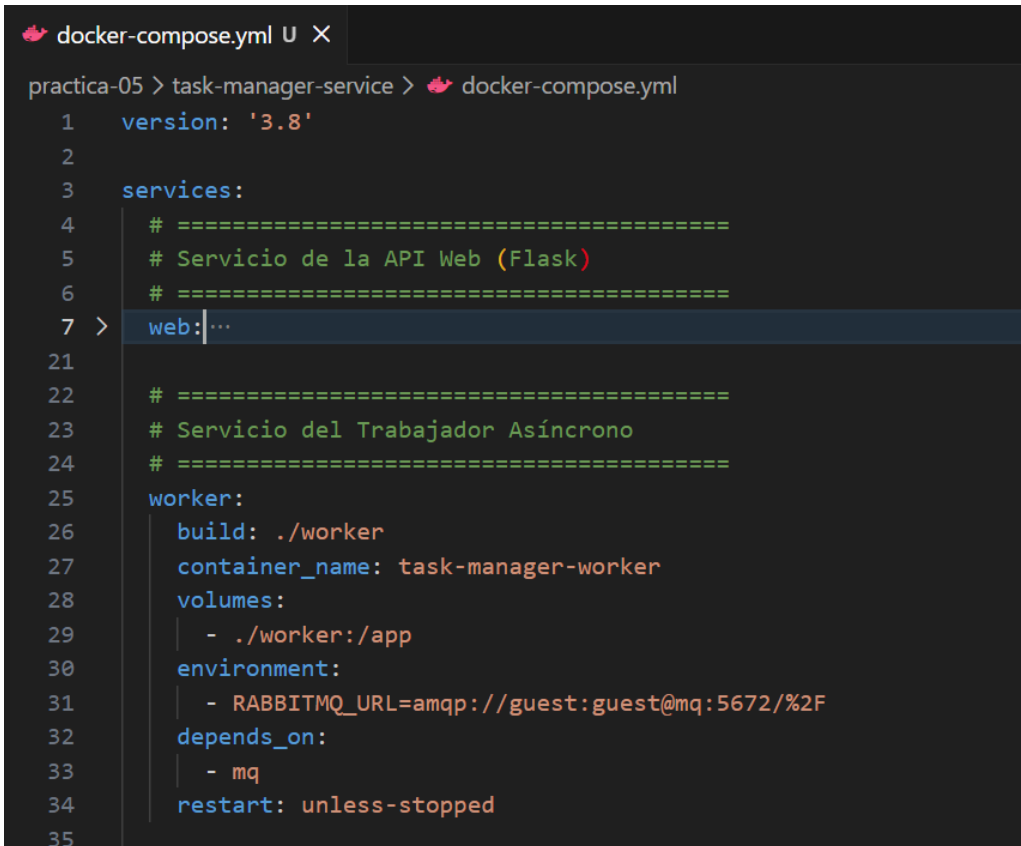


```
1  version: '3.8'
2
3  services:
4      # =====
5      # Servicio de la API Web (Flask)
6      # =====
7      web:
8          build: ./web
9          container_name: task-manager-web
10         ports:
11             - "5001:5000"
12         volumes:
13             - ./web:/app
14         environment:
15             - DATABASE_URL=postgresql://user:password@db:5432/taskdb
16             - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
17         depends_on:
18             - db
19             - mq
20         restart: unless-stopped
21
```

Figura 1: Configuración del servicio Web - API Flask en puerto 5001

El servicio `web` expone la API REST en el puerto 5001 y se conecta tanto a PostgreSQL como a RabbitMQ para gestionar las tareas.

3.3.2. Servicio Worker (Procesador Asíncrono)

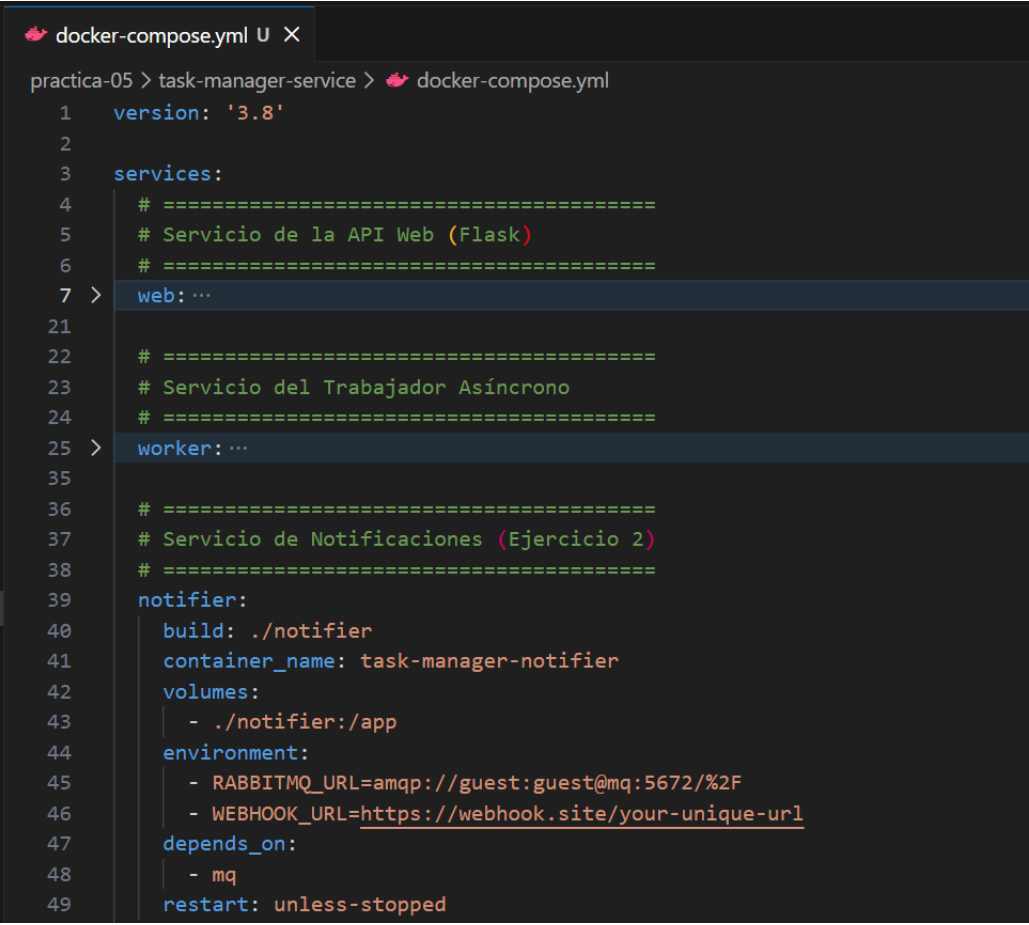


```
docker-compose.yml U X
practica-05 > task-manager-service > docker-compose.yml
1  version: '3.8'
2
3  services:
4      # =====
5      # Servicio de la API Web (Flask)
6      # =====
7  > web:|...
21
22      # =====
23      # Servicio del Trabajador Asíncrono
24      # =====
25      worker:
26          build: ./worker
27          container_name: task-manager-worker
28          volumes:
29              - ./worker:/app
30          environment:
31              - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
32          depends_on:
33              - mq
34          restart: unless-stopped
35
```

Figura 2: Configuración del servicio Worker - Consumidor de colas RabbitMQ

El `worker` consume mensajes de las colas `task_created` y `task_completed`, procesando las tareas de forma asíncrona.

3.3.3. Servicio Notifier (Notificaciones)

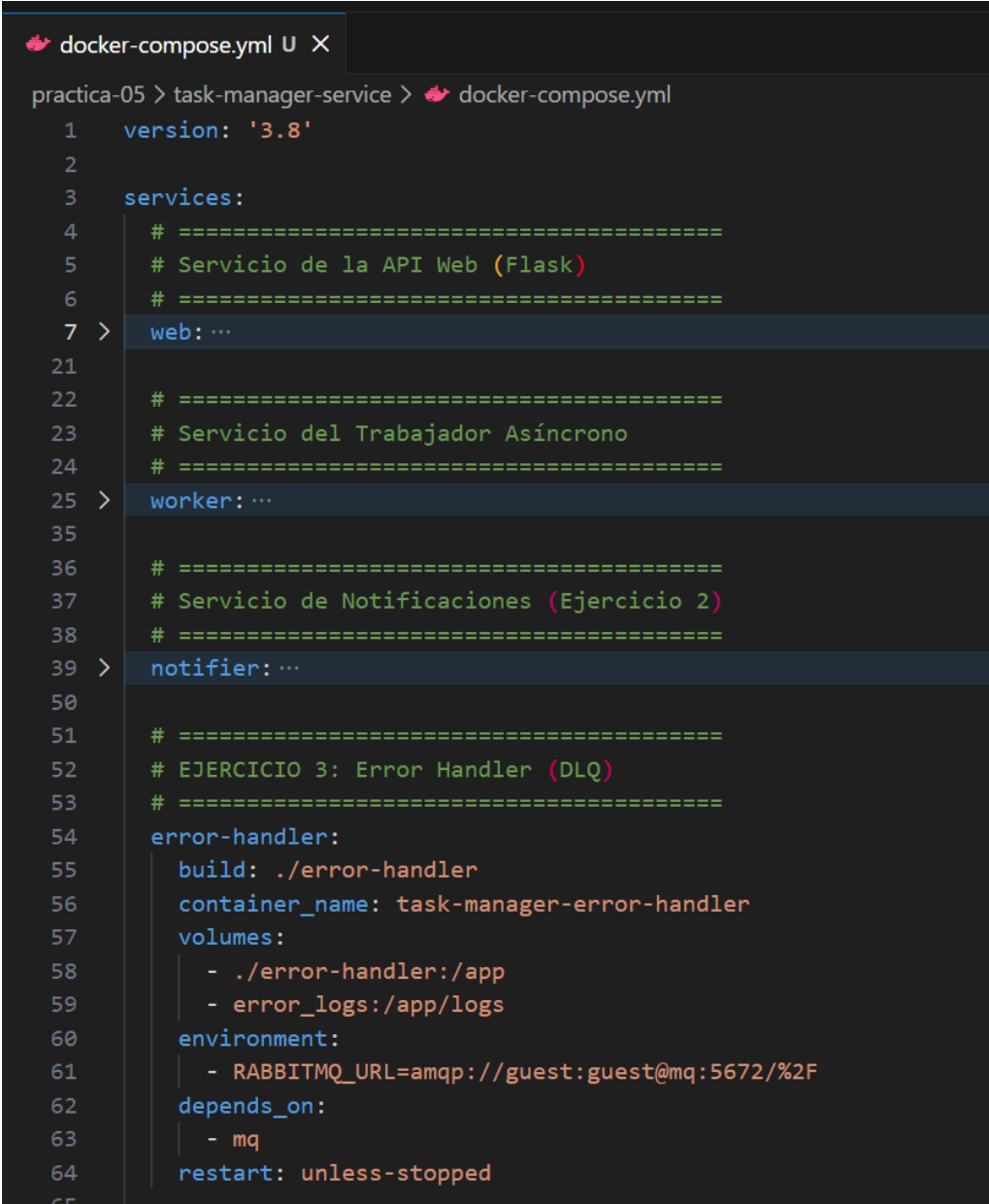


```
docker-compose.yml U X
practica-05 > task-manager-service > docker-compose.yml
1  version: '3.8'
2
3  services:
4      # =====
5      # Servicio de la API Web (Flask)
6      # =====
7  > web: ...
21
22      # =====
23      # Servicio del Trabajador Asíncrono
24      # =====
25  > worker: ...
35
36      # =====
37      # Servicio de Notificaciones (Ejercicio 2)
38      # =====
39  notifier:
40      build: ./notifier
41      container_name: task-manager-notifier
42      volumes:
43      | - ./notifier:/app
44      environment:
45      | - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
46      | - WEBHOOK_URL=https://webhook.site/your-unique-url
47      depends_on:
48      | - mq
49      restart: unless-stopped
```

Figura 3: Configuración del servicio Notifier - Sistema de notificaciones por webhook

El servicio `notifier` escucha la cola `task_completed` y envía notificaciones mediante webhooks externos.

3.3.4. Servicio Error Handler (Dead Letter Queue)



```
practica-05 > task-manager-service > docker-compose.yml
1  version: '3.8'
2
3  services:
4      # =====
5      # Servicio de la API Web (Flask)
6      # =====
7  > web: ...
21
22      # =====
23      # Servicio del Trabajador Asíncrono
24      # =====
25  > worker: ...
35
36      # =====
37      # Servicio de Notificaciones (Ejercicio 2)
38      # =====
39  > notifier: ...
50
51      # =====
52      # EJERCICIO 3: Error Handler (DLQ)
53      # =====
54      error-handler:
55          build: ./error-handler
56          container_name: task-manager-error-handler
57          volumes:
58              - ./error-handler:/app
59              - error_logs:/app/logs
60          environment:
61              - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
62          depends_on:
63              - mq
64          restart: unless-stopped
65
```

Figura 4: Configuración del servicio Error Handler - Gestión de mensajes fallidos

El `error-handler` procesa la Dead Letter Queue (`tasks_failed`), registrando los errores en archivos de log persistentes.

3.3.5. Servicio Database (PostgreSQL)

```
# =====  
# Servicio de la Base de Datos (PostgreSQL)  
# =====  
db:  
  image: postgres:14-alpine  
  container_name: task-manager-db  
  volumes:  
    - postgres_data:/var/lib/postgresql/data/  
  environment:  
    - POSTGRES_USER=user  
    - POSTGRES_PASSWORD=password  
    - POSTGRES_DB=taskdb  
  ports:  
    - "5433:5432"  
  restart: unless-stopped
```

Figura 5: Configuración del servicio Database - PostgreSQL 14 Alpine

La base de datos db utiliza PostgreSQL 14 Alpine, expuesta en el puerto 5433 con volumen persistente para los datos.

3.3.6. Servicio Message Queue (RabbitMQ)

```
81  
82 # =====  
83 # Servicio del Bróker de Mensajes (RabbitMQ)  
84 # =====  
85 mq:  
86   image: rabbitmq:3-management-alpine  
87   container_name: task-manager-mq  
88   ports:  
89     - "5672:5672" # Puerto AMQP  
90     - "15672:15672" # Puerto Management UI  
91   restart: unless-stopped  
92  
93 volumes:  
94   postgres_data:  
95   error_logs:
```

Figura 6: Configuración del servicio RabbitMQ - Broker de mensajes con interfaz de administración

El broker mq ejecuta RabbitMQ 3 con interfaz de gestión, exponiendo los puertos 5672 (AMQP) y 15672 (Management UI).

3.4. Resumen de Servicios Configurados

Servicios configurados:

- **web:** Puerto 5001:5000, conecta a PostgreSQL y RabbitMQ
- **worker:** Consume colas `task_created` y `task_completed`
- **notifier:** Consume cola `task_completed` para notificaciones
- **error-handler:** Consume cola `tasks_failed` (DLQ)
- **db:** PostgreSQL 14 Alpine, puerto 5433:5432
- **mq:** RabbitMQ 3 Management, puertos 5672 y 15672

3.5. Paso 2: API Flask (Servicio Web)

La API expone los siguientes endpoints:

Método	Endpoint	Descripción
GET	/health	Health check
GET	/tasks	Listar todas las tareas
GET	/tasks/{id}	Obtener una tarea
POST	/tasks	Crear nueva tarea
PUT	/tasks/{id}/complete	Completar tarea
DELETE	/tasks/{id}	Eliminar tarea

Cuadro 1: Endpoints de la API REST

```
PS C:\Windows\System32> curl http://localhost:5001/health

StatusCode      : 200
StatusDescription : OK
Content         : {
                  "status": "healthy"
                }
RawContent      : HTTP/1.1 200 OK
                  Connection: close
                  Content-Length: 26
                  Content-Type: application/json
                  Date: Mon, 08 Dec 2025 22:39:06 GMT
                  Server: Werkzeug/2.2.2 Python/3.9.17
                  {
                    "status": "healthy"
                  }
Forms           : {}
Headers         : {[Connection, close], [Content-Length, 26], [Content-Type, application/json], [Date, Mon, 08 Dec
                  2025 22:39:06 GMT]...}
Images          : {}
InputFields     : {}
Links           : {}
ParsedHtml      : mshtml.HTMLDocumentClass
RawContentLength : 26

PS C:\Windows\System32> |
```

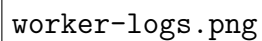
Figura 7: Health check de la API respondiendo correctamente

3.6. Paso 3: Worker (Consumidor de Mensajes)

El worker se conecta a RabbitMQ y consume mensajes de las colas:

```
# Callback para task_created
def callback_created(ch, method, properties, body):
    task_data = json.loads(body)
    print(f"_{x}_ Nueva tarea creada: ID={task_data.get('id')}")
    ch.basic_ack(delivery_tag=method.delivery_tag)

# Callback para task_completed
def callback_completed(ch, method, properties, body):
    task_data = json.loads(body)
    print(f"_{+}_ Tarea completada: ID={task_data.get('id')}")
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

The image is a placeholder for a screenshot of worker logs. The text 'worker-logs.png' is visible in the bottom left corner of the image area.

worker-logs.png

Figura 8: Logs del Worker procesando mensajes de RabbitMQ

3.7. Paso 4: Verificación del Sistema

3.7.1. Contenedores en Ejecución

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker ps
```

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORTS
f4597a3fd277	task-manager-service-error-handler	task-manager-error-handler	"python -u error_han..."	35 minutes ago	Up 35 minutes	
e85fb0baf639	task-manager-service-worker	task-manager-worker	"python -u worker.py"	35 minutes ago	Up 35 minutes	
fe3255dfcb0e	task-manager-service-notifier	task-manager-notifier	"python -u notifier..."	35 minutes ago	Up 35 minutes	
fd01a8fce63d	task-manager-service-web	task-manager-web	"python app.py"	43 minutes ago	Up 41 minutes	0.0.0.0:5001->5000/tcp, [::]:5001->5000/tcp
197278e65b7e	rabbitmq:3-management-alpine	rabbitmq:3-management-alpine	"docker-entrypoint.s..."	43 minutes ago	Up 41 minutes	0.0.0.0:5672->5672/tcp, [::]:5672->5672/tcp, 0.0.0.0:15672->15672/tcp
77f1452f49a2	postgres:14-alpine	postgres:14-alpine	"docker-entrypoint.s..."	43 minutes ago	Up 41 minutes	0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp
79d300d12834	n8nio/n8n	n8n-practicas	"tini -- /docker-ent..."	13 hours ago	Up 13 hours	0.0.0.0:5678->5678/tcp, [::]:5678->5678/tcp

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service>
```

Figura 9: Todos los contenedores ejecutándose correctamente

3.7.2. Creación de Tareas

```
curl -X POST http://localhost:5001/tasks \
-H "Content-Type: application/json" \
-d '{"title": "Configurar sistema", "description": "Docker Compose"}'
```

```
n8n-practicas
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> Invoke-WebRequest -Uri 'http://localhost:5001/tasks' -Method POST -ContentType 'application/json' -Body '{
"task": {
  "title": "Configurar sistema multi-contenedor test",
  "description": "Usar Docker Compose test"
}
}'
task
-----
@{description=User Docker Compose test; done=False; id=6; title=Configurar sistema multi-contenedor test}
```

Figura 10: Respuesta al crear una nueva tarea

3.7.3. Base de Datos PostgreSQL

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker exec task-manager-db psql -U user -d taskdb -c "SELECT * FROM tasks;"
```

id	title	description	done
2	Implementar Dead Letter Queue	Ejercicio 3 - Manejo de mensajes fallidos	f
3	Crear servicio Notifier	Ejercicio 2 - Notificaciones via webhook	f
4	Tarea de prueba para logs	Verificar que worker y notifier funcionan	t
5	Configurar sistema multi-contenedor	Usar Docker Compose	f
6	Configurar sistema multi-contenedor test	Usar Docker Compose test	f

```
(5 rows)
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service>
```

Figura 11: Tareas almacenadas en PostgreSQL

3.7.4. RabbitMQ Management UI

Virtual host	Name	Type	Features	State	Ready	Unacked	Total	Message rates
								incoming deliver / get ack
/	task_completed	classic	D	running	0	0	0	0.00/s 0.00/s 0.00/s
/	task_created	classic	D	running	0	0	0	0.00/s 0.00/s 0.00/s
/	tasks_failed	classic	D	running	0	0	0	

Figura 12: Colas de RabbitMQ en la interfaz de administración

4. Ejercicio 1: Endpoint para Completar Tareas

Objetivo: Implementar un endpoint PUT `/tasks/<id>/complete` que marque una tarea como completada y publique un mensaje en la cola `task_completed`.

4.1. Implementación

Se añadió el siguiente endpoint en `app.py`:

```
@app.route('/tasks/<int:task_id>/complete', methods=['PUT'])
def complete_task(task_id):
    task = Task.query.get(task_id)
    if task is None:
        return jsonify({'error': 'Task not found'}), 404

    task.done = True
    db.session.commit()

    # Publicar mensaje en RabbitMQ
    publish_message('task_completed', task.to_dict())

    return jsonify({
        'message': 'Task marked as completed',
        'task': task.to_dict()
    }), 200
```

4.2. Prueba del Endpoint

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker exec task-manager-db psql -U user -d taskdb -c "SELECT * FROM tasks;"
 id | title | description | done
----+-----+-----+----
  2 | Implementar Dead Letter Queue | Ejercicio 3 - Manejo de mensajes fallidos | f
  3 | Crear servicio Notifier | Ejercicio 2 - Notificaciones via webhook | f
  4 | Tarea de prueba para logs | Verificar que worker y notifier funcionan | t
  5 | Configurar sistema multi-contenedor | Usar Docker Compose | f
  6 | Configurar sistema multi-contenedor test | Usar Docker Compose test | f
(6 rows)

PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> Invoke-WebRequest -Uri "http://localhost:5001/tasks/6/complete" -Method PUT
message task
Task marked as completed @[{description=Usar Docker Compose test; done=True; id=6; title=Configurar sistema multi-contenedor test}]

PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service>
```

Figura 13: Respuesta al completar una tarea

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker logs task-manager-worker
Worker: Conectado a RabbitMQ.
[*] Worker esperando mensajes. Para salir presione CTRL+C
[x] Nueva tarea creada: ID=4, Titulo='Tarea de prueba para logs'
[+] Tarea completada: ID=4, Titulo='Tarea de prueba para logs'
[x] Nueva tarea creada: ID=5, Titulo='Configurar sistema multi-contenedor'
[x] Nueva tarea creada: ID=6, Titulo='Configurar sistema multi-contenedor test'
[+] Tarea completada: ID=6, Titulo='Configurar sistema multi-contenedor test'
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service>
```

Figura 14: Worker procesando el mensaje de tarea completada

5. Ejercicio 2: Servicio Notifier

Objetivo: Crear un servicio independiente que consuma la cola `task_completed` y envíe notificaciones mediante webhook.

5.1. Arquitectura

El servicio Notifier es un consumidor independiente que:

1. Se conecta a RabbitMQ
2. Consume mensajes de la cola `task_completed`
3. Envía notificaciones a un webhook externo

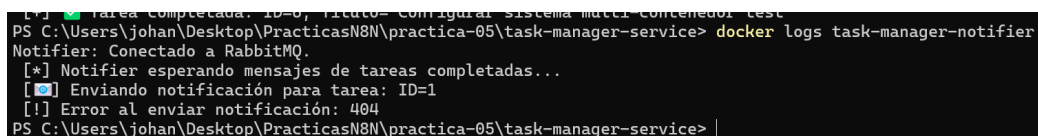
5.2. Implementación

```
def callback(ch, method, properties, body):
    task_data = json.loads(body)
    print(f"[mail] Enviando notificacion para tarea: ID={
        task_data.get('id')}")

    notification_data = {
        'type': 'task_completed',
        'task_id': task_data.get('id'),
        'task_title': task_data.get('title'),
        'message': f"La tarea '{task_data.get('title')}' ha sido
            completada."
    }

    response = requests.post(webhook_url, json=notification_data,
        timeout=5)
    if response.status_code == 200:
        print(f"[ok] Notificacion enviada exitosamente")

    ch.basic_ack(delivery_tag=method.delivery_tag)
```



```
[*] Tarea completada: ID=0, Título= Configurar sistema multi-contenedor test
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker logs task-manager-notifier
Notifier: Conectado a RabbitMQ.
[*] Notifier esperando mensajes de tareas completadas...
[+] Enviando notificación para tarea: ID=1
[!] Error al enviar notificación: 404
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> |
```

Figura 15: Logs del servicio Notifier enviando notificaciones

6. Ejercicio 3: Dead Letter Queue

Objetivo: Implementar un sistema de Dead Letter Queue para manejar mensajes que no pueden ser procesados correctamente.

6.1. Configuración del DLX

Se configuró un Dead Letter Exchange en el worker:

```
# Declarar Dead Letter Exchange
channel.exchange_declare(exchange='dlx_exchange', exchange_type='
    direct', durable=True)

# Declarar cola de tareas fallidas
channel.queue_declare(queue='tasks_failed', durable=True)
channel.queue_bind(exchange='dlx_exchange', queue='tasks_failed',
    routing_key='task_created')

# Cola principal con DLX configurado
channel.queue_declare(
    queue='task_created',
    durable=True,
    arguments={
        'x-dead-letter-exchange': 'dlx_exchange',
        'x-dead-letter-routing-key': 'task_created'
    }
)
```

6.2. Error Handler

El servicio Error Handler consume la cola `tasks_failed` y registra los errores:

```
def callback(ch, method, properties, body):
    task_data = json.loads(body)
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    log_entry = f"[{timestamp}] FAILED TASK: {json.dumps(
        task_data)}\n"

    with open('/app/logs/failed_tasks.log', 'a') as f:
        f.write(log_entry)

    print(f"[warning] Tarea fallida registrada: {task_data}")
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

```
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker logs task-manager-error-handler
Error Handler: Conectado a RabbitMQ.
[*] Error Handler esperando mensajes fallidos...
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> |
```

Figura 16: Error Handler procesando mensajes de la Dead Letter Queue

7. Credenciales de Acceso

7.1. RabbitMQ Management UI

Campo	Valor
URL	http://localhost:15672
Usuario	guest
Contraseña	guest

Cuadro 2: Credenciales de RabbitMQ

7.2. PostgreSQL

Campo	Valor
Host	localhost
Puerto	5433
Base de datos	taskdb
Usuario	user
Contraseña	password

Cuadro 3: Credenciales de PostgreSQL

7.3. API Flask

Campo	Valor
URL Base	http://localhost:5001
Health Check	http://localhost:5001/health

Cuadro 4: URLs de la API

8. Conclusiones

En esta práctica se han alcanzado los siguientes objetivos:

1. **Orquestación con Docker Compose:** Se configuraron 6 servicios interconectados
2. **Comunicación asíncrona:** Implementación de colas RabbitMQ para desacoplar servicios
3. **Persistencia de datos:** PostgreSQL con volúmenes Docker para almacenamiento persistente
4. **Manejo de errores:** Dead Letter Queue para mensajes fallidos
5. **Microservicios:** Arquitectura modular con servicios independientes

8.1. Lecciones Aprendidas

- **PYTHONUNBUFFERED:** Es necesario configurar esta variable para ver logs en tiempo real en Docker
- **depends_on:** No garantiza que el servicio esté listo, solo que el contenedor haya iniciado
- **Competing Consumers:** Múltiples consumidores de la misma cola compiten por los mensajes
- **Message Acknowledgment:** Es crucial confirmar el procesamiento de mensajes para evitar pérdidas
- **Volúmenes Docker:** Permiten persistir datos entre reinicios de contenedores

8.2. Comandos Útiles

```
# Iniciar el sistema
docker-compose up -d --build

# Ver logs de todos los servicios
docker-compose logs -f

# Ver estado de contenedores
docker-compose ps

# Consultar base de datos
docker exec task-manager-db psql -U user -d taskdb -c "SELECT * FROM tasks;"

# Detener el sistema
docker-compose down
```