

Práctica 5:

Orquestación de Microservicios con Docker, PostgreSQL y RabbitMQ

.....

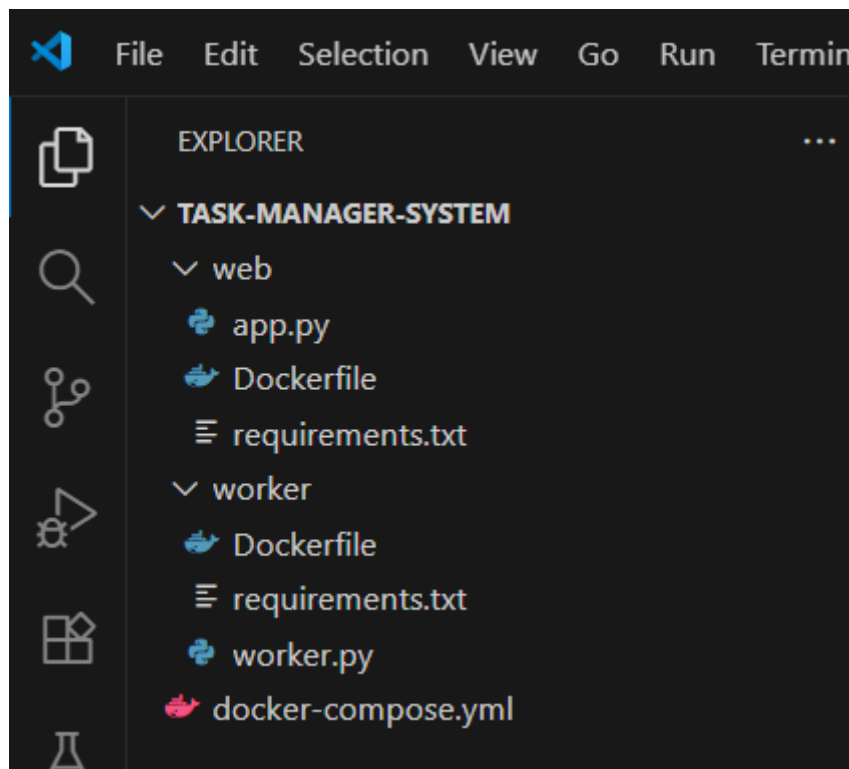
INTEGRACIÓN DE TECNOLOGÍAS Y SERVICIOS
INFORMÁTICOS

Daniel Salas Alonso

3. Desarrollo Práctico: Evolucionando el "Gestor de Tareas"

3.1. Estructura del Proyecto y Configuración Inicial

Se muestra la estructura inicial del proyecto guiado. Contiene dos directorios principales, web (para la API de Flask) y worker (para el consumidor de tareas), cada uno con su propio app.py/worker.py, Dockerfile y requirements.txt. El fichero docker-compose.yml se encuentra en la raíz para sincronizar todos los servicios.



3.2. Creación del Fichero docker-compose.yml

Este fichero define la configuración completa de los servicios.

```
version: '3.8'

services:
  # Servicio de la API Web (Flask)
  web:
    build: ./web
    container_name: task-manager-web
    ports:
      - "5001:5000"
```

```
volumes:
  - ./web:/app # Montaje para desarrollo en vivo
environment:
  - DATABASE_URL=postgresql://user:password@db:5432/taskdb
  - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
depends_on:
  - db
  - mq

# Servicio del Trabajador Asíncrono (Consumer)
worker:
  build: ./worker
  container_name: task-manager-worker
  volumes:
    - ./worker:/app # Montaje para desarrollo en vivo
  environment:
    - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
  depends_on:
    - mq

# Servicio de la Base de Datos
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" # Exponer para depuración externa

# Servicio del Bróker de Mensajes
mq:
  image: rabbitmq:3-management-alpine
  container_name: task-manager-mq
  ports:
    - "5672:5672" # Puerto para la comunicación AMQP
    - "15672:15672" # Puerto para la interfaz de gestión web

volumes:
  postgres_data:
```

3.3. Modificación del Servicio Web (web/)

Completemos los archivos creados anteriormente, utilizando el código otorgado en el ejercicio guiado para el servicio web.

```
# web/app.py
import os
import pika
import json
from flask import Flask, jsonify, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

# --- Configuración de la Base de Datos ---
app.config = os.environ.get('DATABASE_URL')
app.config = False
db = SQLAlchemy(app)

# --- Modelo de la Base de Datos ---
class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(120), nullable=False)
    description = db.Column(db.String(255), nullable=True)
    done = db.Column(db.Boolean, default=False)

    def to_dict(self):
        return {
            'id': self.id,
            'title': self.title,
            'description': self.description,
            'done': self.done
        }

# --- Configuración de RabbitMQ ---
RABBITMQ_URL = os.environ.get('RABBITMQ_URL')
def publish_message(queue_name, message):
    try:
        connection =
pika.BlockingConnection(pika.URLParameters(RABBITMQ_URL))
        channel = connection.channel()
        channel.queue_declare(queue=queue_name, durable=True)
        channel.basic_publish(
            exchange='',
            routing_key=queue_name,
            body=json.dumps(message),
            properties=pika.BasicProperties(delivery_mode=2) # make
message persistent
    )
```

```

        connection.close()
        print(f" [x] Sent message to queue '{queue_name}'")
    except Exception as e:
        print(f"Error publishing message: {e}")

# --- Endpoints de la API ---
@app.route('/tasks', methods= ['GET'])
def get_tasks():
    tasks = Task.query.all()
    return jsonify({'tasks': [task.to_dict() for task in tasks]})

@app.route('/tasks', methods= ['GET'])
def create_task():
    if not request.json or not 'title' in request.json:
        return jsonify({'error': 'Bad request: title is required'}), 400

    new_task = Task(
        title=request.json['title'],
        description=request.json.get('description', "")
    )
    db.session.add(new_task)
    db.session.commit()

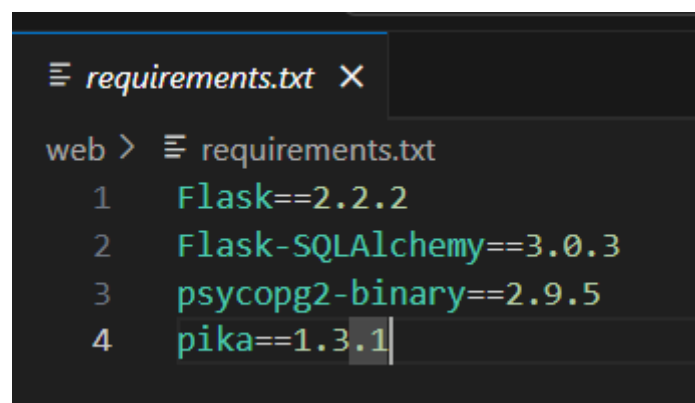
    # Publicar mensaje en RabbitMQ
    publish_message('task_created', new_task.to_dict())

    return jsonify({'task': new_task.to_dict()}), 201

#... (otros endpoints como get_task, delete_task pueden ser añadidos de
forma similar)

if __name__ == '__main__':
    with app.app_context():
        db.create_all() # Crea las tablas si no existen
    app.run(host='0.0.0.0', port=5000, debug=True)

```



The screenshot shows a code editor window with a tab titled "requirements.txt". The editor content shows a terminal-like prompt "web >" followed by the file name "requirements.txt". Below this, a list of dependencies is shown with line numbers 1 through 4. The dependencies are: Flask==2.2.2, Flask-SQLAlchemy==3.0.3, psycpg2-binary==2.9.5, and pika==1.3.1. The cursor is positioned at the end of the fourth line.

```

web > requirements.txt
1  Flask==2.2.2
2  Flask-SQLAlchemy==3.0.3
3  psycpg2-binary==2.9.5
4  pika==1.3.1

```

Muestra el Dockerfile para el servicio web. Es casi idéntico al del servicio worker, pero la instrucción final es CMD ["python", "app.py"] para ejecutar el script de la API en lugar del consumidor.

```
Dockerfile X
web > Dockerfile > ...
1 FROM python:3.9-slim-buster
2
3 WORKDIR /app
4
5 COPY requirements.txt .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 COPY . .
10
11 CMD ["python", "app.py"]
```

3.4. Creación del Servicio de Trabajador (worker/)

Completemos los archivos creados anteriormente, utilizando el código otorgado en el ejercicio guiado para el servicio trabajador.

```
# worker/worker.py
import os
import pika
import json
import time

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...")
            time.sleep(5)
```

```

channel = connection.channel()
channel.queue_declare(queue='task_created', durable=True)

def callback(ch, method, properties, body):
    task_data = json.loads(body)
    print(f" [x] Recibido y procesado nuevo task:
ID={task_data.get('id')}, Título='{task_data.get('title')}'")
    # Aquí iría la lógica de procesamiento (enviar email, etc.)
    ch.basic_ack(delivery_tag=method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='task_created',
on_message_callback=callback)

print(' [*] Esperando mensajes. Para salir presione CTRL+C')
channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrumpido')
    try:
        sys.exit(0)
    except SystemExit:
        os._exit(0)

```

```

requirements.txt X
worker > requirements.txt
1 pika==1.3.1

```

```

Dockerfile X
worker > Dockerfile > ...
1 FROM python:3.9-slim-buster
2
3 WORKDIR /app
4
5 COPY requirements.txt .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 COPY . .
10
11 CMD ["python", "worker.py"]

```

3.5. Ejecución y Verificación del Sistema

Ejecutamos el Docker-compose de nuestro sistema. Como el código otorgado en el ejercicio guiado era erróneo, hay fallos al iniciar el sistema. Las comprobaciones de bases de datos y RabbitMQ corroboran que el sistema no funciona correctamente.

```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P05/task-manager-system
$ docker-compose up --build
=> => exporting config sha256:45ac7aea02698844920eb93d79ec69273792b7ca85d2ec51e18ed999acc8e792 0.0s
=> => exporting attestation manifest sha256:ef5f71a9e1910dd00e893328d6064f3632830631411fd6ea63fb0a68dd077c9c 0.0s
=> => exporting manifest list sha256:455147c742c4dce358b69f87c57aaf600a894f1f73b8e84a385ca596c235edff 0.0s
=> => naming to docker.io/library/task-manager-system-web:latest 0.0s
=> => unpacking to docker.io/library/task-manager-system-web:latest 0.4s
=> [web] resolving provenance for metadata file 0.0s
[+] Running 8/8
✓ task-manager-system-worker Built 0.0s
✓ task-manager-system-web Built 0.0s
✓ Network task-manager-system default Created 0.0s
✓ Volume task-manager-system_postgres_data Created 0.0s
✓ Container task-manager-mq Created 0.1s
✓ Container task-manager-db Created 0.1s
✓ Container task-manager-worker Created 0.1s
✓ Container task-manager-web Created 0.1s
Attaching to task-manager-db, task-manager-mq, task-manager-web, task-manager-worker
task-manager-db | The files belonging to this database system will be owned by user "postgres".
task-manager-db | This user must also own the server process.
task-manager-db |
task-manager-db | The database cluster will be initialized with locale "en_US.utf8".
task-manager-db | The default database encoding has accordingly been set to "UTF8".
task-manager-db | The default text search configuration will be set to "english".
task-manager-db |
task-manager-db | Data page checksums are disabled.
task-manager-db |
task-manager-db | fixing permissions on existing directory /var/lib/postgresql/data ... ok
task-manager-db | creating subdirectories ... ok
task-manager-db | selecting dynamic shared memory implementation ... posix
task-manager-db | selecting default max_connections ... 100
task-manager-db | selecting default shared buffers ... 128kB
```

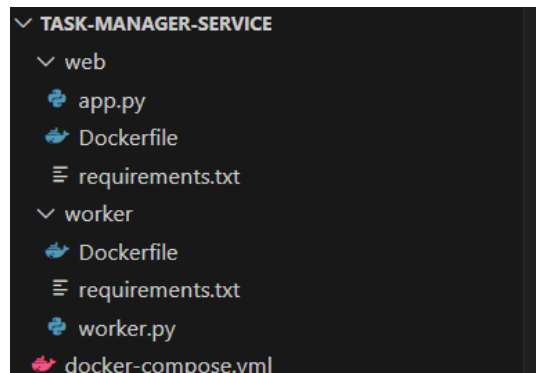
```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P05/task-manager-system
$ docker-compose up --build
task-manager-mq | 2025-11-12 15:53:04.308053+00:00 [info] <0.254.0> Copyright (c) 2007-2024 Broadcom Inc and/or its subsidiaries
task-manager-mq | 2025-11-12 15:53:04.308053+00:00 [info] <0.254.0> Licensed under the MPL 2.0. Website: https://rabbitmq.com
task-manager-mq | ## ## RabbitMQ 3.13.7
task-manager-mq | ## ##
task-manager-mq | ##### Copyright (c) 2007-2024 Broadcom Inc and/or its subsidiaries
task-manager-mq | ##### ##
task-manager-mq | ##### ## Licensed under the MPL 2.0. Website: https://rabbitmq.com
task-manager-mq |
task-manager-mq | Erlang: 26.2.5.16 [jit]
task-manager-mq | TLS Library: OpenSSL - OpenSSL 3.1.8 11 Feb 2025
task-manager-mq | Release series support status: see https://www.rabbitmq.com/release-information
task-manager-mq |
task-manager-mq | Doc guides: https://www.rabbitmq.com/docs
task-manager-mq | Support: https://www.rabbitmq.com/docs/contact
task-manager-mq | Tutorials: https://www.rabbitmq.com/tutorials
task-manager-mq | Monitoring: https://www.rabbitmq.com/docs/monitoring
task-manager-mq | Upgrading: https://www.rabbitmq.com/docs/upgrade
task-manager-mq |
task-manager-mq | Logs: <stdout>
task-manager-mq |
task-manager-mq | Config file(s): /etc/rabbitmq/conf.d/10-defaults.conf
task-manager-mq |
task-manager-mq | Starting broker...2025-11-12 15:53:04.309139+00:00 [info] <0.254.0>
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> node : rabbit@96faff4b54fe
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> home dir : /var/lib/rabbitmq
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> config file(s) : /etc/rabbitmq/conf.d/10-defaults.conf
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> cookie hash : JvWj71XA18H50Z3Xmz/gcg==
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> log(s) : <stdout>
task-manager-mq | 2025-11-12 15:53:04.309139+00:00 [info] <0.254.0> data dir : /var/lib/rabbitmq/mnesia/rabbit@96faff4b54fe
```


4. Ejercicios Propuestos

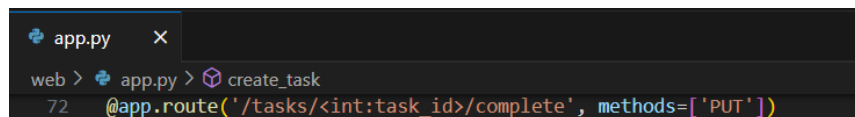
4.1. Ejercicio 1: Actualización de Tareas y Notificación (Dificultad: Baja)

1. *Añada un nuevo endpoint PUT /tasks/<int:task_id>/complete a la API web (web/app.py).*

Para este ejercicio utilizaremos un nuevo repositorio actualizado y con las fallas del ejercicio guiado arregladas. El nombre de este nuevo proyecto será “TASK-MANAGER-SERVICE” y su estructura inicial será idéntica el proyecto anterior:

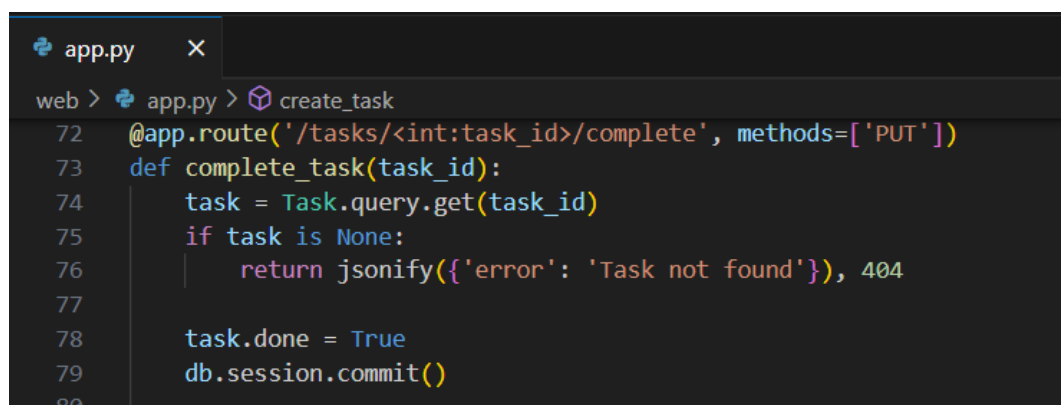


Se crea un nuevo endpoint PUT /tasks/<int:task_id>/complete en app.py.



2. *Este endpoint debe encontrar la tarea en la base de datos, cambiar su estado done a true y guardar el cambio.*

Se añade al endpoint la lógica de la función complete_task, la cual busca la tarea en la base de datos usando Task.query.get(task_id). Si la encuentra, actualiza su estado (task.done = True) y guarda el cambio en la base de datos con db.session.commit().



- Después de actualizar la base de datos, debe publicar un mensaje en una nueva cola de RabbitMQ llamada `task_completed`. El mensaje debe contener los datos de la tarea actualizada.

Al proceso anterior del método se añade una llamada a `publish_message('task_completed', task.to_dict())`. Esto envía un mensaje con los datos de la tarea actualizada a una nueva cola llamada `task_completed` en RabbitMQ.

```
app.py  X
web > app.py > create_task
72 @app.route('/tasks/<int:task_id>/complete', methods=['PUT'])
73 def complete_task(task_id):
74     task = Task.query.get(task_id)
75     if task is None:
76         return jsonify({'error': 'Task not found'}), 404
77
78     task.done = True
79     db.session.commit()
80
81     # Publicar mensaje en RabbitMQ
82     publish_message('task_completed', task.to_dict())
83
84     return jsonify({'task': task.to_dict()}), 200
85
```

```
app.py  X
web > app.py > publish_message
16 class Task(db.Model):
28     }
29
30 # --- Configuración de RabbitMQ ---
31 RABBITMQ_URL = os.environ.get('RABBITMQ_URL')
32 def publish_message(queue_name, message):
33     try:
34         connection = pika.BlockingConnection(pika.URLParameters(RABBITMQ_URL))
35         channel = connection.channel()
36
37         # Declare DLX exchange, dead-letter queue, and binding for task_created
38         if queue_name == 'task_created':
39             channel.exchange_declare(exchange='dlx', exchange_type='direct')
40             channel.queue_declare(queue='tasks_failed', durable=True)
41             channel.queue_bind(exchange='dlx', queue='tasks_failed', routing_key='tasks_failed')
42
43         # Declare the queue with DLX arguments if it's task_created
44         dlx_args = {'x-dead-letter-exchange': 'dlx', 'x-dead-letter-routing-key': 'tasks_failed'} if queue_name == 'task_created' else {}
45         channel.queue_declare(queue=queue_name, durable=True, arguments=dlx_args)
46
47         channel.basic_publish(
48             exchange='',
49             routing_key=queue_name,
50             body=json.dumps(message),
51             properties=pika.BasicProperties(delivery_mode=2) # make message persistent
52         )
53         connection.close()
54         print(f"[x] Sent message to queue '{queue_name}'")
55     except Exception as e:
56         print(f"Error publishing message: {e}")
57
```

4. Modifique el worker/worker.py para que también escuche en la cola `task_completed` y muestre un mensaje de log diferente, como `[+] Tarea completada: ID={...}`.

Se modifica el worker.py original para manejar la nueva cola.

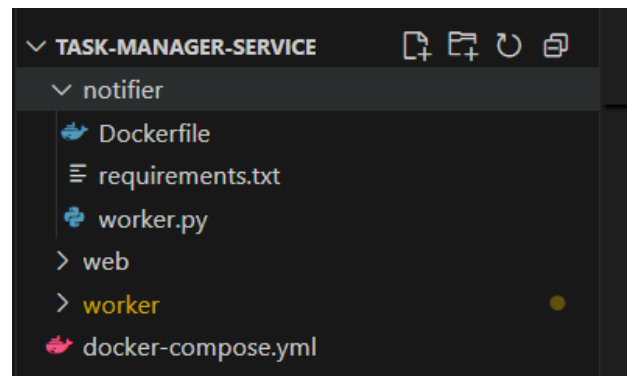
- **Declaración:** Declara la nueva cola `task_completed`.
- **Nuevo Callback:** Se añade una nueva función `callback_task_completed` que procesa los mensajes de esta nueva cola, imprimiendo un log: Tarea completada....
- **Consumo Dual:** El worker ahora consume de *dos* colas: `task_created` (usando `callback`) y `task_completed` (usando `callback_task_completed`)

```
worker.py 1 X
worker > worker.py > main
7 def main():
10
11 # Bucle para reintentar la conexión si RabbitMQ no está listo
12 while not connection:
13     try:
14         connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
15         print("Worker: Conectado a RabbitMQ.")
16     except pika.exceptions.AMQPConnectionError:
17         print("Worker: Esperando a RabbitMQ...")
18         time.sleep(5)
19
20 channel = connection.channel()
21 channel.queue_declare(queue='task_created', durable=True)
22 channel.queue_declare(queue='task_completed', durable=True)
23
24 def callback(ch, method, properties, body):
25     task_data = json.loads(body)
26     print(f" [X] Recibido y procesado nuevo task: ID={task_data.get('id')}, Título='{task_data.get('title')}'")
27     # Aquí iría la lógica de procesamiento (enviar email, etc.)
28     ch.basic_ack(delivery_tag=method.delivery_tag)
29
30 def callback_task_completed(ch, method, properties, body):
31     task_data = json.loads(body)
32     print(f" [+] Tarea completada: ID={task_data.get('id')}, Título= '{task_data.get('title')}'")
33     # Aquí iría la lógica de procesamiento (enviar notificación, etc.)
34     ch.basic_ack(delivery_tag=method.delivery_tag)
35
36 channel.basic_qos(prefetch_count=1)
37 channel.basic_consume(queue='task_created', on_message_callback=callback)
38 channel.basic_consume(queue='task_completed', on_message_callback=callback_task_completed)
39
40 print(' [*] Esperando mensajes. Para salir presione CTRL+C')
41 channel.start_consuming()
42
43 if __name__ == '__main__':
44     try:
```

4.2. Ejercicio 2: Servicio de Notificaciones por Email (Simulado) (Dificultad: Media)

1. Cree un nuevo directorio `notifier/` con su propio `worker.py`, `requirements.txt` y `Dockerfile`.

Se crea un nuevo directorio `notifier` para albergar un nuevo servicio con estructura idéntica a los otros servicios.



2. Añada este nuevo servicio al `docker-compose.yml`.

Se añade un nuevo servicio `notifier` al `docker-compose.yml`. Su configuración es análoga a la del `worker` (construido desde `./notifier`, depende de `mq`). Se añade una variable de entorno `NOTIFIER_WEBHOOK_URL`, la cual contendrá la url de una webhook que se utilizará más adelante.

```
docker-compose.yml M X
docker-compose.yml
2  services:
4  web:
18
19  # Servicio del Trabajador Asíncrono (Consumer)
20  > Run Service
21  worker:
22    build: ./worker
23    restart: always
24    container_name: task-manager-worker
25    volumes:
26      - ./worker:/app # Montaje para desarrollo en vivo
27    environment:
28      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
29    depends_on:
30      - mq
31
32  # Servicio del Trabajador Asíncrono Notificador (Consumer)
33  > Run Service
34  notifier:
35    build: ./notifier
36    restart: always
37    container_name: task-manager-notifier
38    volumes:
39      - ./notifier:/app # Montaje para desarrollo en vivo
40    environment:
41      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
42      - NOTIFIER_WEBHOOK_URL=https://webhook.site/76e736e5-410f-430b-b878-6a98b8fde81e
43    depends_on:
44      - mq
```

3. El servicio notifier debe consumir mensajes de la cola `task_completed` (del ejercicio anterior).

El código para el nuevo servicio notifier se muestra. Inicialmente, este código es una copia del worker modificado del Ejercicio 1, ya que escucha en ambas colas (`task_created` y `task_completed`). La intención es que este servicio maneje las notificaciones.

```
# notifier/worker.py
import os
import pika
import json
import time

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...")
            time.sleep(5)

    channel = connection.channel()
    channel.queue_declare(queue='task_created', durable=True)
    channel.queue_declare(queue='task_completed', durable=True)

    def callback(ch, method, properties, body):
        task_data = json.loads(body)
        print(f" [x] Recibido y procesado nuevo task:
ID={task_data.get('id')}, Título='{task_data.get('title')}'")
        # Aquí iría la lógica de procesamiento (enviar email, etc.)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    def callback_task_completed(ch, method, properties, body):
        task_data = json.loads(body)
        print(f" [+] Tarea completada: ID={task_data.get('id')},
Título= '{task_data.get('title')}'")
        # Aquí iría la lógica de procesamiento (enviar notificación,
etc.)
        ch.basic_ack(delivery_tag=method.delivery_tag)
```

```

        channel.basic_qos(prefetch_count=1)
        channel.basic_consume(queue='task_created',
on_message_callback=callback)
        channel.basic_consume(queue='task_completed',
on_message_callback=callback_task_completed)

        print(' [*] Esperando mensajes. Para salir presione CTRL+C')
        channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrumpido')
    try:
        sys.exit(0)
    except SystemExit:
        os._exit(0)

```

4. El servicio worker original ya no debe escuchar en task_completed.

Se modifica el worker original para que ya no escuche en task_completed. Se eliminan la declaración de la cola, la función callback_task_completed y la línea channel.basic_consume para esa cola. Worker/worker.py se muestra en la captura escuchando únicamente en task_created.

```

worker.py 1 x
worker > worker.py > ...
1  # worker/worker.py
2  import os
3  import pika
4  import json
5  import time
6
7  def main():
8      rabbitmq_url = os.environ.get('RABBITMQ_URL')
9      connection = None
10
11     # Bucle para reintentar la conexión si RabbitMQ no está listo
12     while not connection:
13         try:
14             connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
15             print("Worker: Conectado a RabbitMQ.")
16         except pika.exceptions.AMQPConnectionError:
17             print("Worker: Esperando a RabbitMQ...")
18             time.sleep(5)
19
20     channel = connection.channel()
21     channel.queue_declare(queue='task_created', durable=True)
22
23     def callback(ch, method, properties, body):
24         task_data = json.loads(body)
25         print(f" [x] Recibido y procesado nuevo task: ID={task_data.get('id')}, Título='{task_data.get('title')}'")
26         # Aquí iría la lógica de procesamiento (enviar email, etc.)
27         ch.basic_ack(delivery_tag=method.delivery_tag)
28
29     channel.basic_qos(prefetch_count=1)
30     channel.basic_consume(queue='task_created', on_message_callback=callback)
31
32     print(' [*] Esperando mensajes. Para salir presione CTRL+C')
33     channel.start_consuming()
34
35 if __name__ == '__main__':
36     try:
37         main()

```

5. Cuando el servicio notifier recibe un mensaje, en lugar de imprimir en la consola, debe simular el envío de un email. Puede hacerlo haciendo una petición POST a un servicio como <https://webhook.site/>, que le proporcionará una URL única para recibir peticiones.

Se añade una variable `webhook_url` que se lee del entorno (NOTIFIER_WEBHOOK_URL).

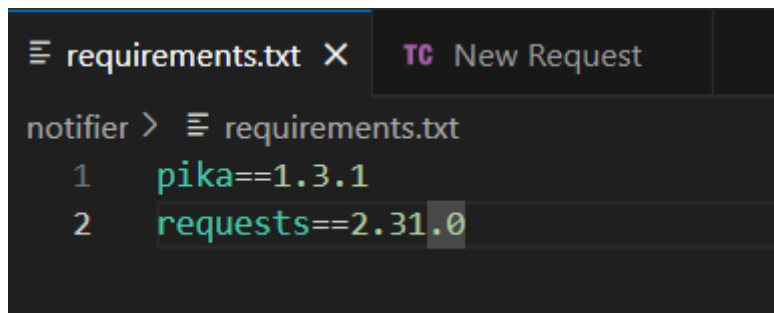
```
webhook_url = os.environ.get('NOTIFIER_WEBHOOK_URL')
if not webhook_url:
    print("Warning: NOTIFIER_WEBHOOK_URL no está definido; no se enviarán webhooks.")
```

Dentro de los callbacks se construye un payload JSON (Cuerpo del mensaje que incluye los campos de este y sus valores asociados). Se utiliza la librería `requests` para enviar este payload mediante un POST a la `webhook_url`.

```
def callback(ch, method, properties, body):
    task_data = json.loads(body)
    print(f" [x] Recibido y procesado nuevo task: ID={task_data.get('id')}, Título='{task_data.get('title')}'")
    # Simular envío de email mediante POST al webhook
    try:
        payload = {"event": "task_created", "task": task_data}
        if webhook_url:
            requests.post(webhook_url, json=payload, timeout=5)
    except Exception as e:
        print(f"Error enviando webhook (task_created): {e}")
    ch.basic_ack(delivery_tag=method.delivery_tag)

def callback_task_completed(ch, method, properties, body):
    task_data = json.loads(body)
    print(f" [+] Tarea completada: ID={task_data.get('id')}, Título= '{task_data.get('title')}'")
    # Simular envío de email mediante POST al webhook
    try:
        payload = {"event": "task_completed", "task": task_data}
        if webhook_url:
            requests.post(webhook_url, json=payload, timeout=5)
    except Exception as e:
        print(f"Error enviando webhook (task_completed): {e}")
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

El fichero de requisitos del notifier ahora incluye `requests` además de `pika`.



The screenshot shows a terminal window with a dark background. At the top, there's a header bar with a hamburger menu icon, the text 'requirements.txt', a close button 'X', and a tab labeled 'TC New Request'. Below the header, the prompt 'notifier >' is followed by another hamburger menu icon and 'requirements.txt'. The file content is listed as follows:

```
1  pika==1.3.1
2  requests==2.31.0
```

Se muestra el código completo del notificador. Importa requests. En la función main, obtiene la webhook_url y declara ambas colas. En callback (para task_created) y callback_task_completed (para task_completed), se prepara un payload y se envía vía requests.post.

El notifier/worker.py se conecta a RabbitMQ y escucha mensajes en dos colas distintas: task_created y task_completed.

Cuando recibe un mensaje (ya sea de una tarea nueva o una completada), prepara un payload JSON y lo envía a una URL de webhook externa (simulando el envío de un email o una alerta).

Además, confirma (ack) el mensaje a RabbitMQ incluso si el envío al webhook falla. Esto evita que la cola se bloquee y que el mensaje se intente procesar repetidamente solo porque el sistema de notificación externo está caído.

```
# notifier/worker.py
import os
import pika
import json
import time
import requests
import sys

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...")
            time.sleep(5)

    # ADD: webhook URL para simular envío de email (p.e.
    https://webhook.site/xxxxxx)
    webhook_url = os.environ.get('NOTIFIER_WEBHOOK_URL')
    if not webhook_url:
        print("Warning: NOTIFIER_WEBHOOK_URL no está definido; no se
enviarán webhooks.")
```



```

channel = connection.channel()
channel.queue_declare(queue='task_created', durable=True)
channel.queue_declare(queue='task_completed', durable=True)

def callback(ch, method, properties, body):
    task_data = json.loads(body)
    print(f" [x] Recibido y procesado nuevo task:
ID={task_data.get('id')}, Título='{task_data.get('title')}'")
    # Simular envío de email mediante POST al webhook
    try:
        payload = {"event": "task_created", "task": task_data}
        if webhook_url:
            requests.post(webhook_url, json=payload, timeout=5)
    except Exception as e:
        print(f"Error enviando webhook (task_created): {e}")
    ch.basic_ack(delivery_tag=method.delivery_tag)

def callback_task_completed(ch, method, properties, body):
    task_data = json.loads(body)
    print(f" [+] Tarea completada: ID={task_data.get('id')},
Título= '{task_data.get('title')}'")
    # Simular envío de email mediante POST al webhook
    try:
        payload = {"event": "task_completed", "task": task_data}
        if webhook_url:
            requests.post(webhook_url, json=payload, timeout=5)
    except Exception as e:
        print(f"Error enviando webhook (task_completed): {e}")
    ch.basic_ack(delivery_tag=method.delivery_tag)

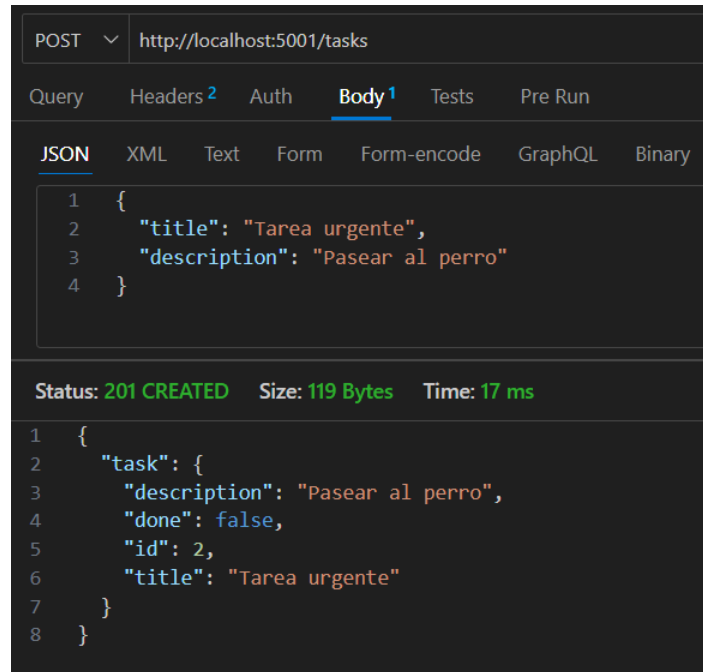
channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='task_created',
on_message_callback=callback)
channel.basic_consume(queue='task_completed',
on_message_callback=callback_task_completed)

print(' [*] Esperando mensajes. Para salir presione CTRL+C')
channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrumpido')
    try:
        sys.exit(0)
    except SystemExit:
        os._exit(0)

```

Para estas pruebas básicas de las operaciones POST, GET y PUT se ha utilizado la extensión Thunder Client de Visual Code. En primera instancia, vemos como se utiliza, de forma satisfactoria, la ruta POST /tasks para crear la "Tarea urgente" (ID 2). Responde con 201 CREATED.



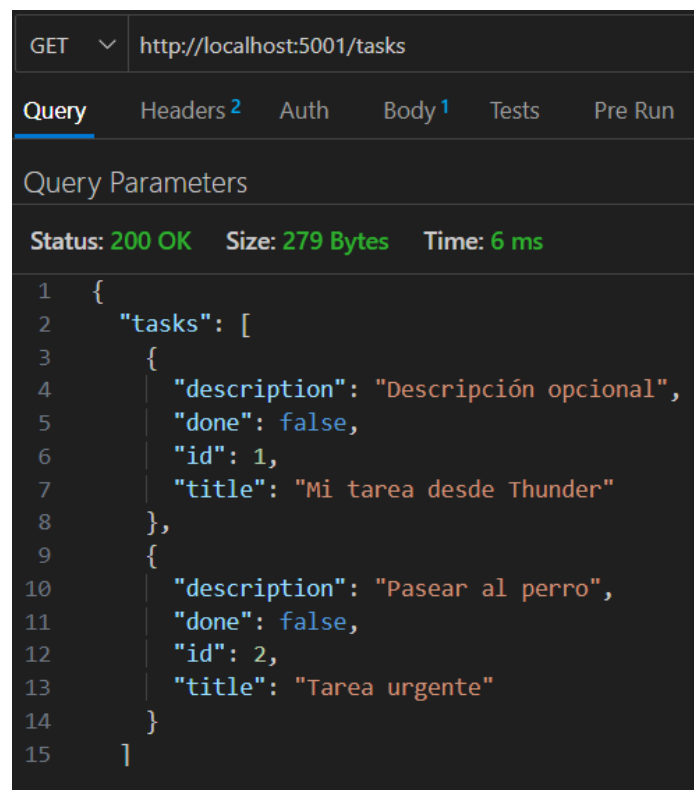
```
POST http://localhost:5001/tasks

{
  "title": "Tarea urgente",
  "description": "Pasear al perro"
}
```

Status: 201 CREATED Size: 119 Bytes Time: 17 ms

```
{
  "task": {
    "description": "Pasear al perro",
    "done": false,
    "id": 2,
    "title": "Tarea urgente"
  }
}
```

GET /tasks para listar las tareas. Muestra la tarea 1 y la nueva tarea 2.



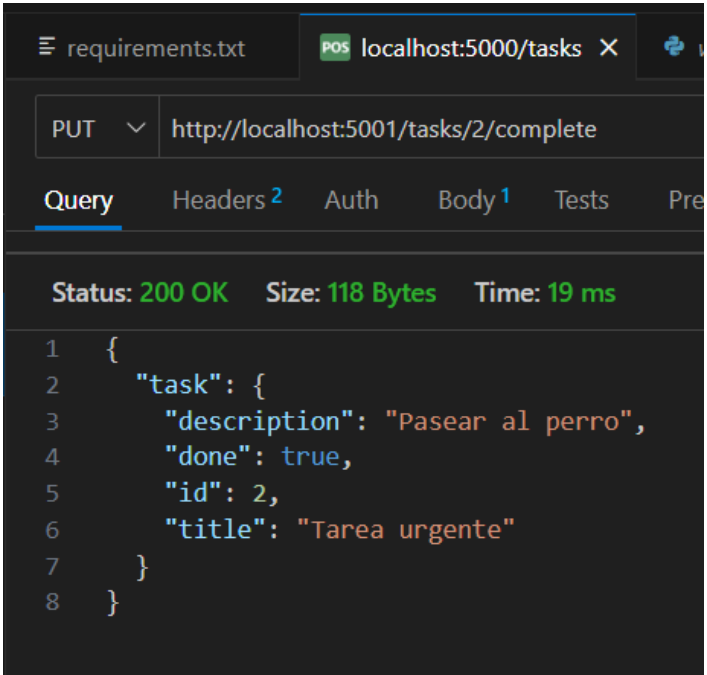
```
GET http://localhost:5001/tasks

Query Parameters
```

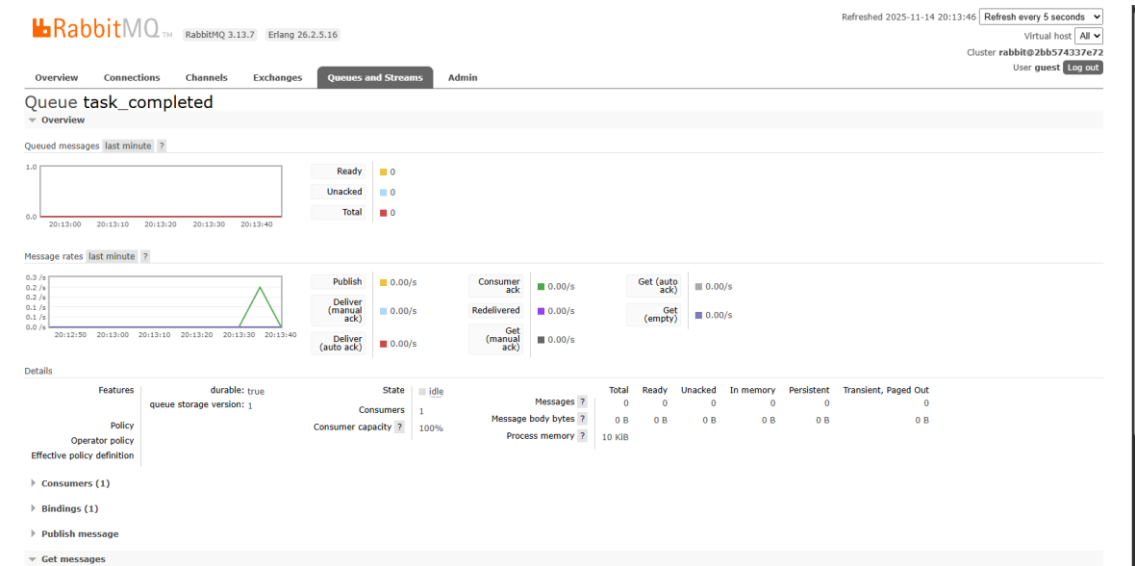
Status: 200 OK Size: 279 Bytes Time: 6 ms

```
{
  "tasks": [
    {
      "description": "Descripción opcional",
      "done": false,
      "id": 1,
      "title": "Mi tarea desde Thunder"
    },
    {
      "description": "Pasear al perro",
      "done": false,
      "id": 2,
      "title": "Tarea urgente"
    }
  ]
}
```

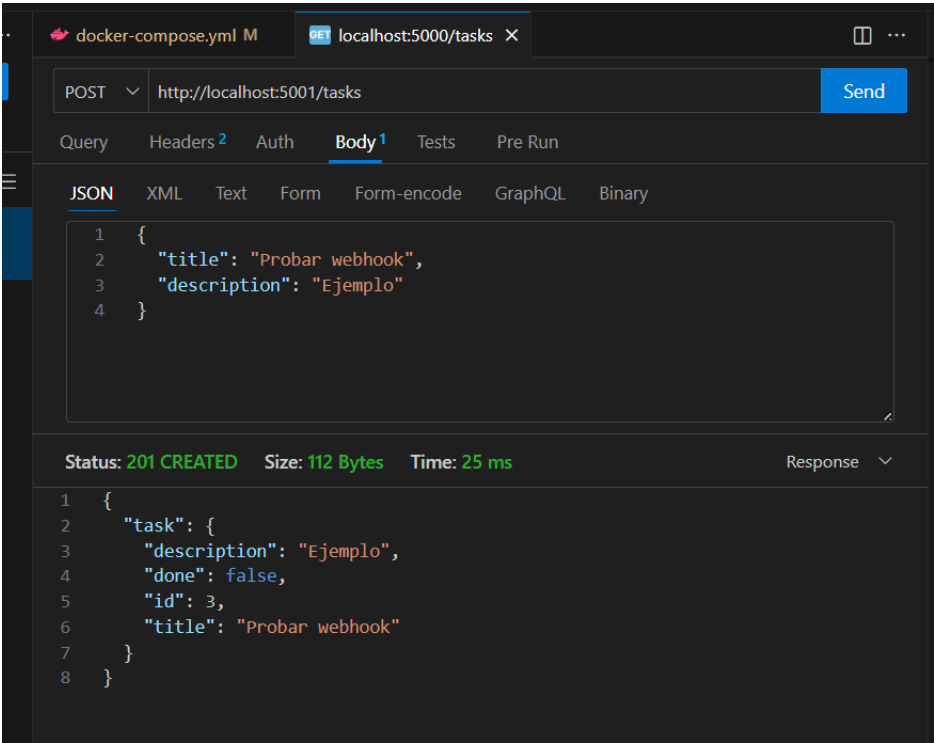
PUT /tasks/2/complete para completar la tarea. Responde con 200 OK y muestra done: true.



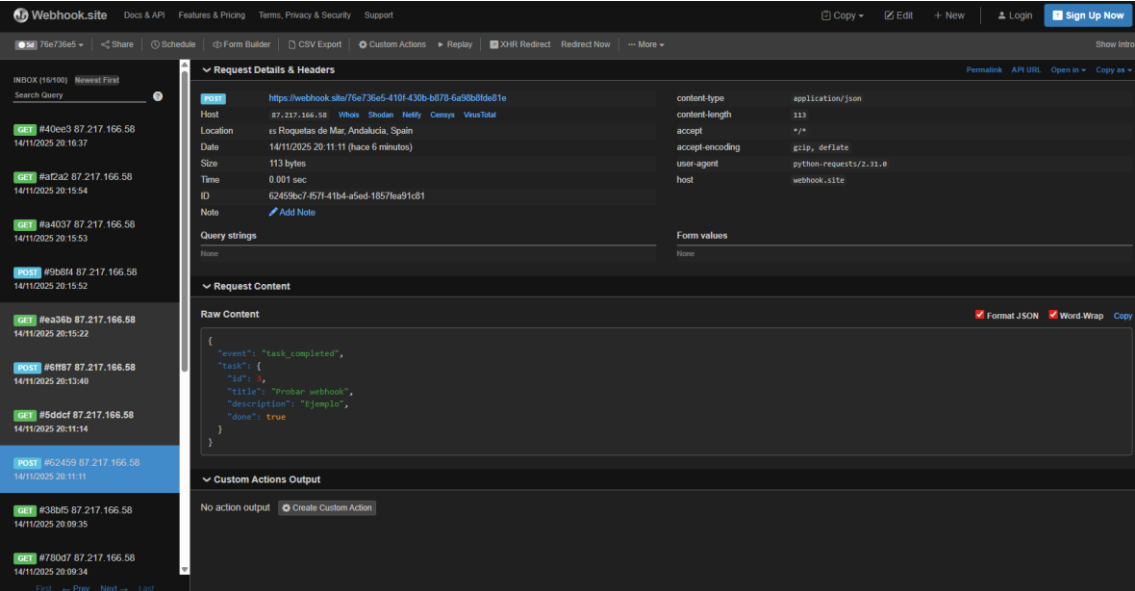
Desde la interfaz de RabbirMQ, podemos observar que el mensaje anterior ha sido recibido por la cola task_completed, al marcar la tarea como completada mediante el PUT.



Por último, creamos otro mensaje para verificar que la url de nuestro Webhook recibe la tarea cuando esta es creada:



Verificamos que, en efecto, la petición de la creación de la tarea se ve reflejada en la cola de INBOX de nuestro webhook:



4.3. Ejercicio 3: Persistencia de Mensajes y "Dead Letter Queue" (Dificultad: Alta)

1. *Investigue sobre las "Dead Letter Exchanges" (DLX) en RabbitMQ.*

Cuando un productor envía un mensaje, no lo envía directamente a una cola, sino que se lo entrega al Exchange. La tarea del Exchange es recibir ese mensaje y decidir a qué cola o colas debe enrutarlo, en base a las reglas especificadas.

Un Dead Letter Exchange (DLX) es simplemente un exchange normal que se designa para recibir mensajes "muertos", es decir, mensajes que no pudieron ser procesados. Si una cola está configurada con un DLX, cualquier mensaje que sea rechazado (con nack), expire, o exceda el límite de la cola, será automáticamente reenviado al DLX en lugar de ser descartado. El principal beneficio es la resiliencia: se evita la pérdida de mensajes fallidos, permitiendo que sean analizados o reprocesados por un servicio especializado.

2. *Modifique la declaración de la cola `task_created` en el servicio web y worker para que, si un mensaje es rechazado (nack) por el consumidor, se envíe a una DLX y, de ahí, a una cola de "letras muertas" (e.g., `tasks_failed`).*

Se modifica la función `publish_message` en la API web.

- Si la cola es `task_created`, primero declara un exchange llamado `dlx` y una cola llamada `tasks_failed`.
- Vincula (`queue_bind`) la cola `tasks_failed` al exchange `dlx` con una clave de enrutamiento `tasks_failed`.
- Al declarar la cola principal `task_created`, le añade arguments especiales: `x-dead-letter-exchange: 'dlx'` y `x-dead-letter-routing-key: 'tasks_failed'`

Así aseguramos que DLX se inicialice cuando se va a publicar.

```
# web/app.py
import os
import pika
import json
from flask import Flask, jsonify, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
```

```

# --- Configuración de la Base de Datos ---
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

# --- Modelo de la Base de Datos ---
class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(120), nullable=False)
    description = db.Column(db.String(255), nullable=True)
    done = db.Column(db.Boolean, default=False)

    def to_dict(self):
        return {
            'id': self.id,
            'title': self.title,
            'description': self.description,
            'done': self.done
        }

# --- Configuración de RabbitMQ ---
RABBITMQ_URL = os.environ.get('RABBITMQ_URL')
def publish_message(queue_name, message):
    try:
        connection =
pika.BlockingConnection(pika.URLParameters(RABBITMQ_URL))
        channel = connection.channel()

        # Declare DLX exchange
        if queue_name == 'task_created':
            channel.exchange_declare(exchange='dlx',
exchange_type='direct')
            channel.queue_declare(queue='tasks_failed', durable=True)
            channel.queue_bind(exchange='dlx', queue='tasks_failed',
routing_key='tasks_failed')

            # Declare the queue with DLX arguments if it's task_created
            dlx_args = {'x-dead-letter-exchange': 'dlx', 'x-dead-letter-
routing-key': 'tasks_failed'} if queue_name == 'task_created' else {}
            channel.queue_declare(queue=queue_name, durable=True,
arguments=dlx_args)

        channel.basic_publish(
            exchange='',
            routing_key=queue_name,
            body=json.dumps(message),
            properties=pika.BasicProperties(delivery_mode=2)
        )
        connection.close()

```

```

        print(f" [x] Sent message to queue '{queue_name}''")
    except Exception as e:
        print(f"Error publishing message: {e}")

# --- Endpoints de la API ---
@app.route('/tasks', methods=['GET'])
def get_tasks():
    tasks = Task.query.all()
    return jsonify({'tasks': [task.to_dict() for task in tasks]})

@app.route('/tasks', methods=['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        return jsonify({'error': 'Bad request: title is required'}), 400

    new_task = Task(
        title=request.json['title'],
        description=request.json.get('description', "")
    )
    db.session.add(new_task)
    db.session.commit()

    # Publicar mensaje en RabbitMQ
    publish_message('task_created', new_task.to_dict())

    return jsonify({'task': new_task.to_dict()}), 201

# Endpoint para completar una tarea (4.1)
@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({'task': task.to_dict()}), 200

#... (otros endpoints como get_task, delete_task pueden ser añadidos de
forma similar)

if __name__ == '__main__':
    with app.app_context():
        db.create_all() # Crea las tablas si no existen
    app.run(host='0.0.0.0', port=5000, debug=True)

```

Se aplica una lógica similar en el worker/worker.py. En main, declara el dlx, la cola tasks_failed y el queue_bind . Al declarar la cola task_created, añade los mismos arguments de DLX.

```
# worker/worker.py
import os
import pika
import json
import time

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...")
            time.sleep(5)

    channel = connection.channel()

    # Declare DLX exchange, dead-letter queue, and binding
    channel.exchange_declare(exchange='dlx', exchange_type='direct')
    channel.queue_declare(queue='tasks_failed', durable=True)
    channel.queue_bind(exchange='dlx', queue='tasks_failed',
routing_key='tasks_failed')

    # Declare task_created queue with DLX arguments
    channel.queue_declare(queue='task_created', durable=True,
arguments={'x-dead-letter-exchange': 'dlx', 'x-dead-letter-routing-key':
'tasks_failed'})

    def callback(ch, method, properties, body):
        task_data = json.loads(body)
        print(f" [x] Recibido y procesado nuevo task:
ID={task_data.get('id')}, Título='{task_data.get('title')}'")
        # Aquí iría la lógica de procesamiento (enviar email, etc.)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    channel.basic_qos(prefetch_count=1)
    channel.basic_consume(queue='task_created',
on_message_callback=callback)
```



```

    print(' [*] Esperando mensajes. Para salir presione CTRL+C')
    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrumpido')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)

```

El servicio notifier también debe estar al tanto de la configuración de la cola. En su función main, declara el dlx, la cola tasks_failed y el queue_bind. Al declarar task_created, añade los arguments de DLX. La cola de tareas completadas no se modifica, ya que solo se especifica únicamente en el enunciado la creación de tareas.

```

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...")
            time.sleep(5)

    # ADD: webhook URL para simular envío de email (p.e.
https://webhook.site/xxxxxx)
    webhook_url = os.environ.get('NOTIFIER_WEBHOOK_URL')
    if not webhook_url:
        print("Warning: NOTIFIER_WEBHOOK_URL no está definido; no se
enviarán webhooks.")

    channel = connection.channel()

    # Declare DLX exchange, dead-letter queue, and binding
    channel.exchange_declare(exchange='dlx', exchange_type='direct')
    channel.queue_declare(queue='tasks_failed', durable=True)

```

```

    channel.queue_bind(exchange='dlx', queue='tasks_failed',
routing_key='tasks_failed')

    # Declare queues with DLX arguments for task_created
    channel.queue_declare(queue='task_created', durable=True,
arguments={'x-dead-letter-exchange': 'dlx', 'x-dead-letter-routing-key':
'tasks_failed'})
    channel.queue_declare(queue='task_completed', durable=True)

    def callback(ch, method, properties, body):
        task_data = json.loads(body)
        print(f" [x] Recibido y procesado nuevo task:
ID={task_data.get('id')}, Título='{task_data.get('title')}'")
        # Simular envío de email mediante POST al webhook
        try:
            payload = {"event": "task_created", "task": task_data}
            if webhook_url:
                requests.post(webhook_url, json=payload, timeout=5)
        except Exception as e:
            print(f"Error enviando webhook (task_created): {e}")
        ch.basic_ack(delivery_tag=method.delivery_tag)

    def callback_task_completed(ch, method, properties, body):
        task_data = json.loads(body)
        print(f" [+] Tarea completada: ID={task_data.get('id')},
Título= '{task_data.get('title')}'")
        # Simular envío de email mediante POST al webhook
        try:
            payload = {"event": "task_completed", "task": task_data}
            if webhook_url:
                requests.post(webhook_url, json=payload, timeout=5)
        except Exception as e:
            print(f"Error enviando webhook (task_completed): {e}")
        ch.basic_ack(delivery_tag=method.delivery_tag)

    channel.basic_qos(prefetch_count=1)
    channel.basic_consume(queue='task_created',
on_message_callback=callback)
    channel.basic_consume(queue='task_completed',
on_message_callback=callback_task_completed)

    print(' [*] Esperando mensajes. Para salir presione CTRL+C')
    channel.start_consuming()

```

3. *Modifique el worker.py para que, si una tarea recibida no tiene un campo title (simulando un mensaje malformado), rechace el mensaje (channel.basic_nack) sin volver a encolarlo.*

En worker/worker.py dentro del callback, se añade una comprobación (if 'title' not in task_data). Si falta el título, imprime un error y rechaza el mensaje con ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False). El requeue=False sirve para que el mensaje no vuelva a la cola original y sea enviado al DLX.

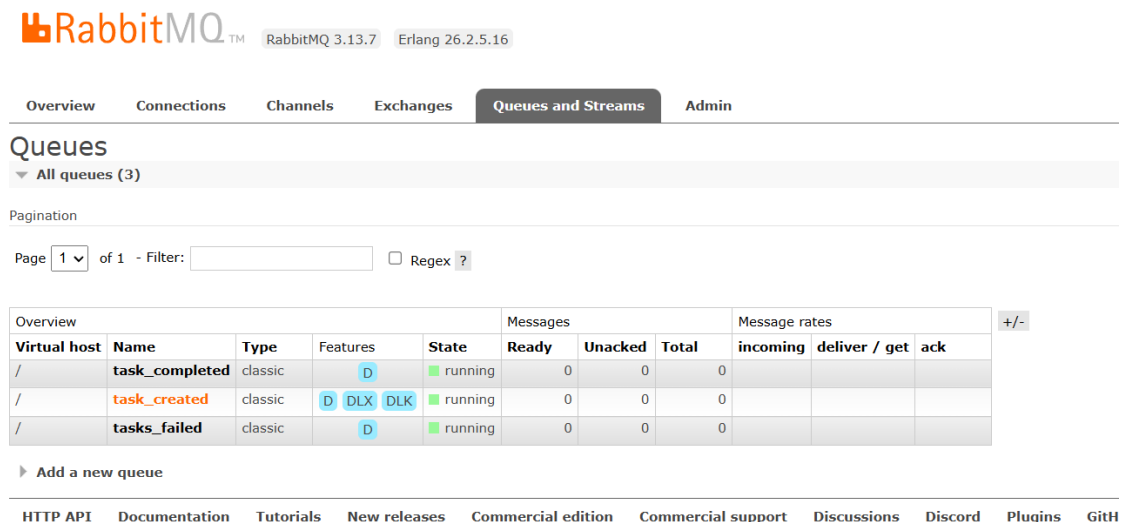
```
def callback(ch, method, properties, body):  
    task_data = json.loads(body)  
  
    if 'title' not in task_data:  
        print(f" [x] Mensaje malformado rechazado: falta campo 'title'.  
Enviando a DLX.")  
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)  
        return  
  
    print(f" [x] Recibido y procesado nuevo task:  
ID={task_data.get('id')}, Título='{task_data.get('title')}'")  
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

La misma lógica de comprobación y nack se añade al callback del notifier/worker.py.

```
def callback(ch, method, properties, body):  
    task_data = json.loads(body)  
  
    if 'title' not in task_data:  
        print(f" [x] Mensaje malformado rechazado: falta campo 'title'.  
Enviando a DLX.")  
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)  
        return  
  
    print(f" [x] Recibido y procesado nuevo task:  
ID={task_data.get('id')}, Título='{task_data.get('title')}'")  
    try:  
        payload = {"event": "task_created", "task": task_data}  
        if webhook_url:  
            requests.post(webhook_url, json=payload, timeout=5)  
    except Exception as e:  
        print(f"Error enviando webhook (task_created): {e}")  
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

4. Verifique en la interfaz de RabbitMQ que, al enviar un mensaje sin title, este termina en la cola `tasks_failed`.

Iniciamos la web local de la interfaz de RabbitMQ. En la pestaña “Queues and Streams” vemos que existen 3 colas: `task_completed`, `task_created` (con las etiquetas D de durable y DLX), y `tasks_failed`. Seleccionamos `task_created` para crear una tarea.



RabbitMQ 3.13.7 Erlang 26.2.5.16

Overview Connections Channels Exchanges **Queues and Streams** Admin

Queues

▼ All queues (3)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview					Messages			Message rates				
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	+/-	
/	task_completed	classic	D	running	0	0	0					
/	task_created	classic	D DLX DLK	running	0	0	0					
/	tasks_failed	classic	D	running	0	0	0					

► Add a new queue

HTTP API Documentation Tutorials New releases Commercial edition Commercial support Discussions Discord Plugins GitHub

Se publica manualmente un mensaje en `task_created` que no tiene el campo `title`:

▼ Publish message

Message will be published to the default exchange with routing key `task_created`, routing it to this queue.

Delivery mode: 1 - Non-persistent ▼

Headers: ? = String ▼

Properties: ? =

Payload:

```
{
  "id": 3,
  "description": "Tarea sin titulo"
}
```

Payload encoding: String (default) ▼

Publish message

Volvemos a la interfaz anterior y seleccionamos `tasks_failed` para verificar que el mensaje ha ido a esa cola.

RabbitMQ

RabbitMQ 3.13.7

Erlang 26.2.5.16

Overview

Connections

Channels

Exchanges

Queues and Streams

Admin

Queues

All queues (3)

Page 1 of 1 - Filter: ☐ Regex ?

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	task_completed	classic	D	running	0	0	0			
/	task_created	classic	D DLX DLK	running	0	0	0	0.00/s	0.00/s	0.00/s
/	tasks_failed	classic	D	running	1	0	1		0.00/s	0.00/s

Add a new queue

HTTP API

Documentation

Tutorials

New releases

Commercial edition

Commercial support

Discussions

Discord

Plugins

GitHub

Inmediatamente, la cola `tasks_failed` muestra un "Total" de 1 mensaje, indicando que el rechazo y el DLX funcionaron.

RabbitMQ

RabbitMQ 3.13.7

Erlang 26.2.5.16

Overview

Connections

Channels

Exchanges

Queues and Streams

Admin

Queue tasks_failed

Overview

Queued messages last minute ?

Ready

Unacked

Total

1

0

1

Message rates last minute ?

Currently idle

Details

Features	State	Messages	Total	Ready	Unacked	In memory	Persistent	Transient, Paged Out
Features	State	Messages	1	1	0	1	0	0
Policy	Consumers	Message body bytes	50 B	50 B	0 B	50 B	0 B	0 B
Operator policy	Consumer capacity	Process memory	7.5 KiB					
Effective policy definition								

Consumers (0)

Bindings (2)

Publish message

Al ir a "Get Message(s)" en la cola, se puede inspeccionar el mensaje. Verificamos que el payload es el del mensaje anterior.

 RabbitMQ 3.13.7 Erlang 26.2.5.16

Overview

Connections

Channels

Exchanges

Queues and Streams

Admin

▼ Get messages

Warning: getting messages from a queue is a destructive action. ?

Ack Mode:

Nack message requeue true ▼

Encoding:

Auto string / base64 ▼ ?

Messages:

1

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange

Routing Key

Redelivered

Properties

dlx

tasks_failed

0

delivery_mode: 1

headers:

x-death:

count: 1

exchange:

queue: task_created

reason: rejected

routing-keys: task_created

time: 1763050629

x-first-death-exchange:

x-first-death-queue: task_created

x-first-death-reason: rejected

x-last-death-exchange:

x-last-death-queue: task_created

x-last-death-reason: rejected

Payload

50 bytes

Encoding: string

{

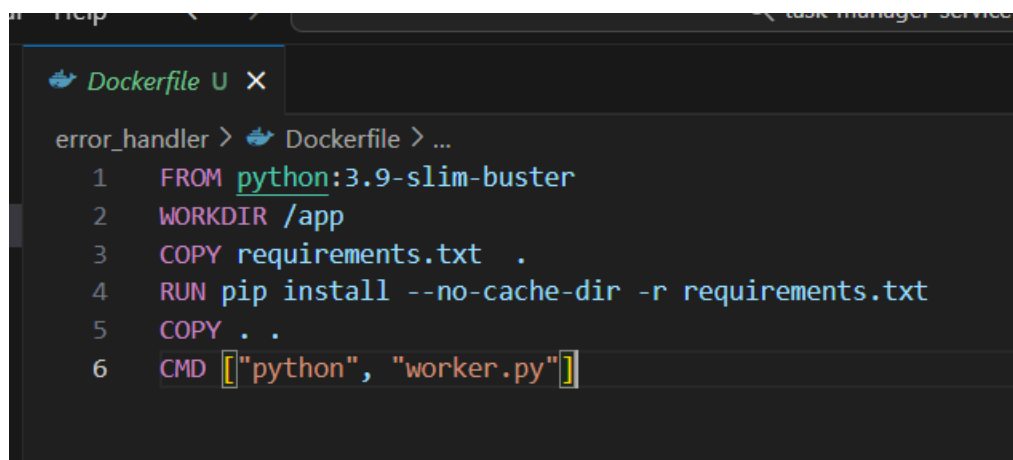
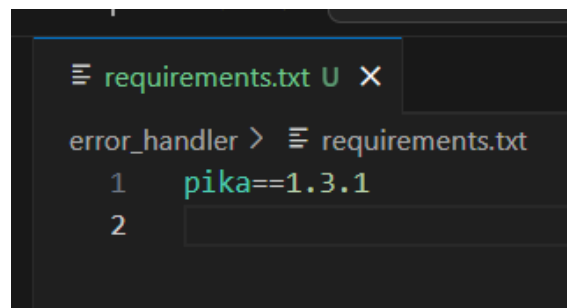
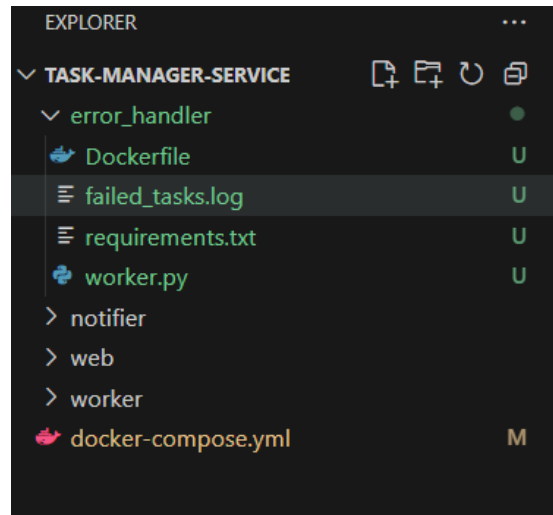
"id": 3,

"description": "Tarea sin titulo"

}

5. (Bonus) Cree un cuarto servicio, `error_handler`, que consuma de la cola `tasks_failed` y registre los mensajes fallidos en un fichero de log o en una tabla separada de la base de datos.

Se crea un cuarto servicio en el directorio `error_handler` con una estructura idéntica a los anteriores, a excepción de que, este, cuenta con un archivo log que recopilará los mensajes fallidos. También se muestra su Dockerfile (con la misma estructura que los servicios anteriores), el archivo `failed_tasks.log` vacío y un `requirements.txt` que solo contiene pika.



Se muestra el código para este nuevo consumidor.

- **Configuración:** Importa logging. En main, se conecta a RabbitMQ .
- **Logging:** Configura logging.basicConfig para escribir en el fichero failed_tasks.log .
- **Consumo:** Declara *únicamente* la cola tasks_failed.
- **Callback:** La función callback toma el mensaje fallido, lo registra en el fichero (logging.info), lo imprime en consola y envía el ack, consumiendo así el mensaje de la cola de errores.
- Inicia el consumo *solo* de la cola tasks_failed.

```
# error_handler/worker.py
import os
import pika
import json
import time
import logging
import sys

def main():
    rabbitmq_url = os.environ.get('RABBITMQ_URL')
    connection = None

    # Bucle para reintentar la conexión si RabbitMQ no está listo
    while not connection:
        try:
            connection =
pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Error Handler: Conectado a RabbitMQ.")
        except pika.exceptions.AMQPConnectionError:
            print("Error Handler: Esperando a RabbitMQ...")
            time.sleep(5)

    channel = connection.channel()

    # Declare the tasks_failed queue (it may already exist from other
services, but declare for safety)
    channel.queue_declare(queue='tasks_failed', durable=True)

    # Set up logging to a file
    logging.basicConfig(
        filename='failed_tasks.log',
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )
```



```

def callback(ch, method, properties, body):
    task_data = json.loads(body)
    logging.info(f"Mensaje fallido recibido: {task_data}")
    print(f" [x] Mensaje fallido registrado: {task_data}")
    ch.basic_ack(delivery_tag=method.delivery_tag)

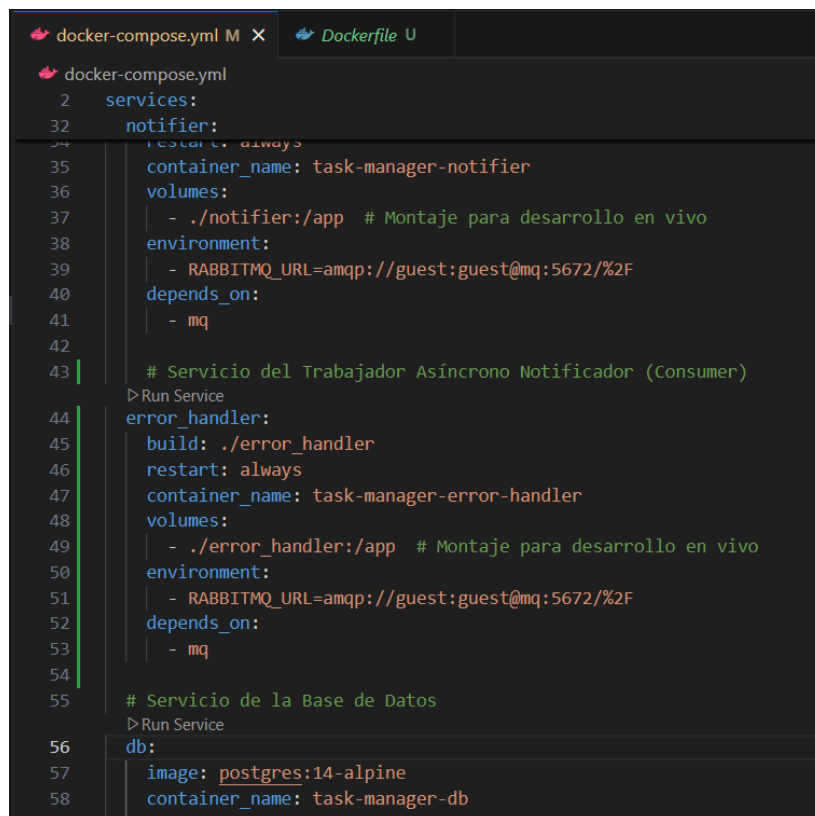
channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='tasks_failed',
on_message_callback=callback)

print(' [*] Error Handler esperando mensajes fallidos. Para salir
presione CTRL+C')
channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrumpido')
    try:
        sys.exit(0)
    except SystemExit:
        os._exit(0)

```

Se añade el nuevo servicio error_handler al docker-compose.yml, configurado de forma similar a los otros workers.

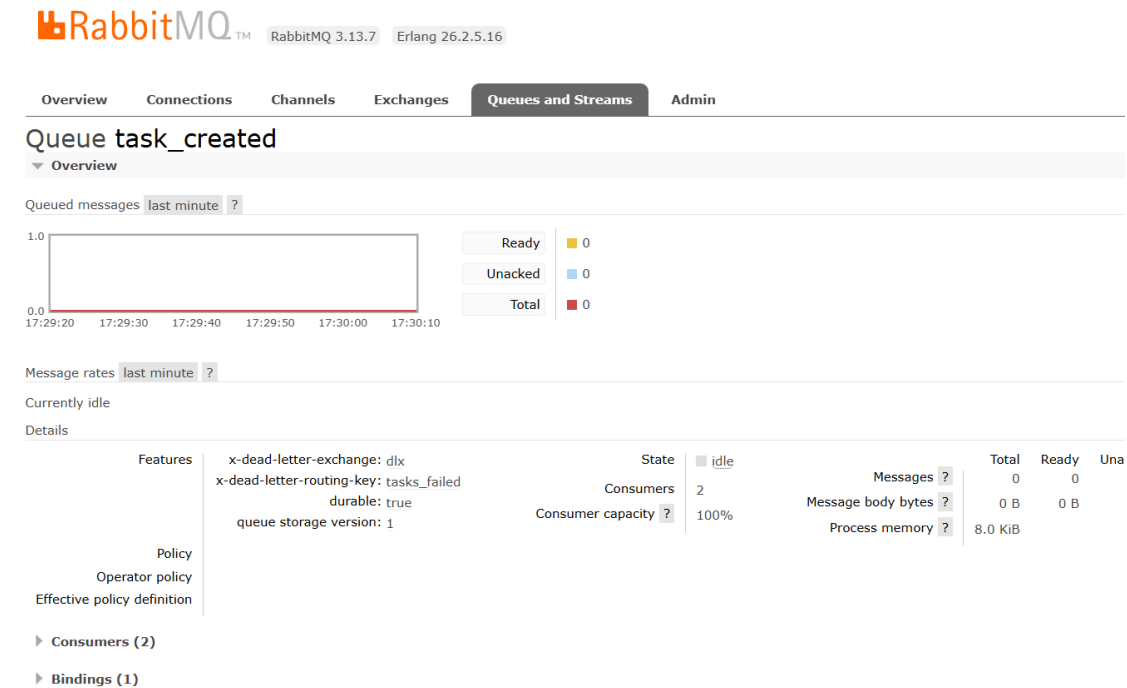


```

docker-compose.yml M X Dockerfile U
docker-compose.yml
2  services:
32  notifier:
33      restart: always
34      container_name: task-manager-notifier
35      volumes:
36      - ./notifier:/app # Montaje para desarrollo en vivo
37      environment:
38      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
39      depends_on:
40      - mq
41
42
43  # Servicio del Trabajador Asíncrono Notificador (Consumer)
44  error_handler:
45      build: ./error_handler
46      restart: always
47      container_name: task-manager-error-handler
48      volumes:
49      - ./error_handler:/app # Montaje para desarrollo en vivo
50      environment:
51      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
52      depends_on:
53      - mq
54
55  # Servicio de la Base de Datos
56  db:
57      image: postgres:14-alpine
58      container_name: task-manager-db
59      volumes:

```

En la interfaz de RabbitMQ, accedemos a la cola task_created para crear un mensaje erroneo.



Se publica otro mensaje sin especificar el campo título, en el apartado Publish message.

Publish message

Message will be published to the default exchange with routing key **task_created**, routing it to this queue.

Delivery mode: **1 - Non-persistent**

Headers: ? = String

Properties: ? =

Payload:

```
{
  "id": 4,
  "description": "Prueba log error handler"
}
```

Payload encoding: **String (default)**

Publish message

La cola `tasks_failed` muestra actividad del paso del mensaje que acabamos de crear, aunque observamos que hay 0 mensajes listos. Esto indica que el mensaje llegó y fue consumido instantáneamente por el nuevo `error_handler`.



Verificamos que la cola de `tasks_failed` está vacía.

The screenshot shows the RabbitMQ web interface for the `tasks_failed` queue, specifically the Details tab. The top navigation bar includes Overview, Connections, Channels, Exchanges, Queues and Streams (selected), and Admin. The queue name `tasks_failed` is displayed. The Details tab shows the Operator policy, Effective policy definition, and Process memory (7.2 KiB). A message box states "Queue is empty" with a Close button. The Get messages section shows a warning: "getting messages from a queue is a destructive action." and a form to get messages. The form includes fields for Ack Mode (Nack message requeue true), Encoding (Auto string / base64), and Messages (1). The Get Message(s) button is visible. The bottom section includes links for Move messages, Delete, Purge, and Runtime Metrics (Advanced).

`Failed_tasks.log` contiene la línea de log generada por el `error_handler`, registrando exitosamente el mensaje fallido y sus datos:

```
failed_tasks.log U X
error_handler > failed_tasks.log
1 2025-11-13 16:31:28,361 - INFO - Mensaje fallido recibido: {'id': 4, 'description': 'Prueba log_error_handler'}
```