Práctica 5:

Orquestación de Microservicios con Docker, PostgreSQL y RabbitMQ

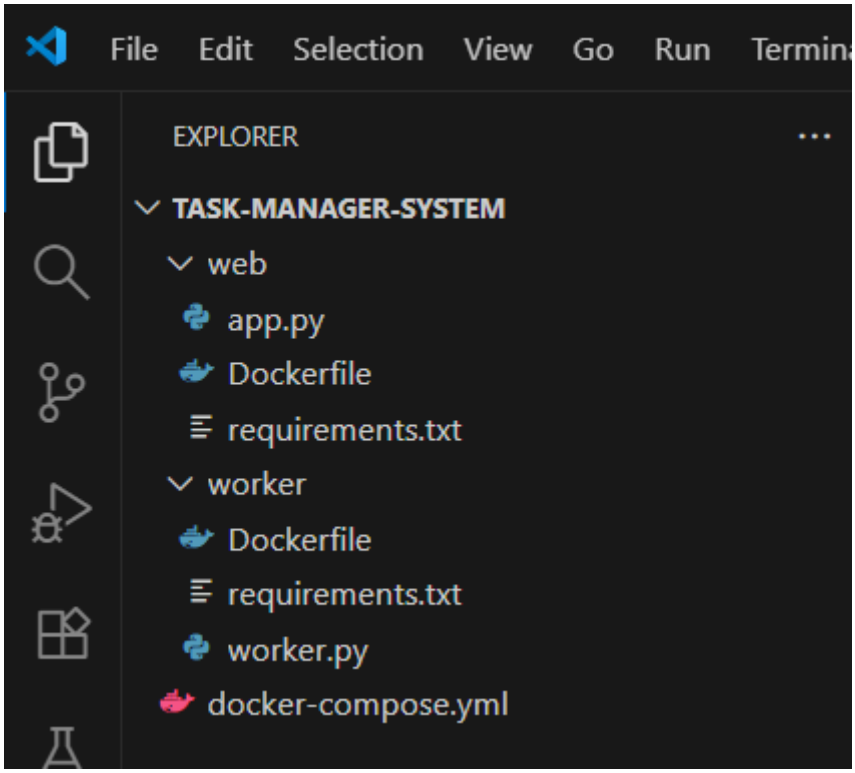
INTEGRACIÓN DE TECNOLOGÍAS Y SERVICIOS INFORMÁTICOS

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 guiaado. 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.

```yaml
version: '3.8'

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

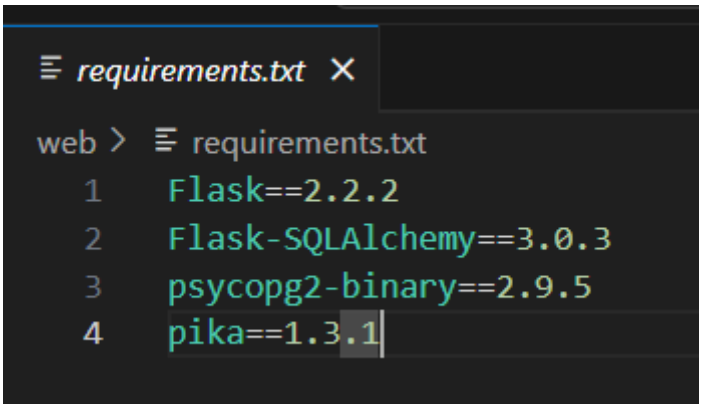
```yaml
    volumes:
      - ./web:/app  # Montaje para desarrollo en vivo
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/taskdb
      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
    depends_on:
      - db
      - mq

  # 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/)

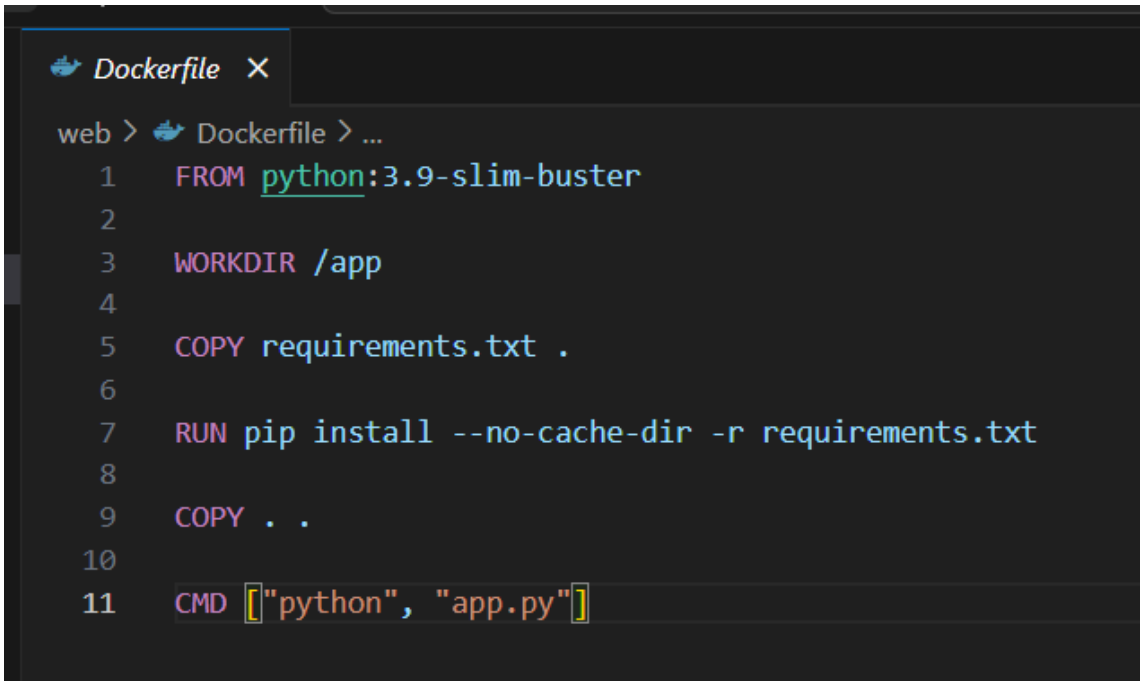
Completemos los archivos creados anteriormente, utilizando el código otorgado en el ejercicio guiado para el servicio web.

```
# 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)
```



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.



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)
```

```python
    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

worker > requirements.txt

```
1  pika==1.3.1
```

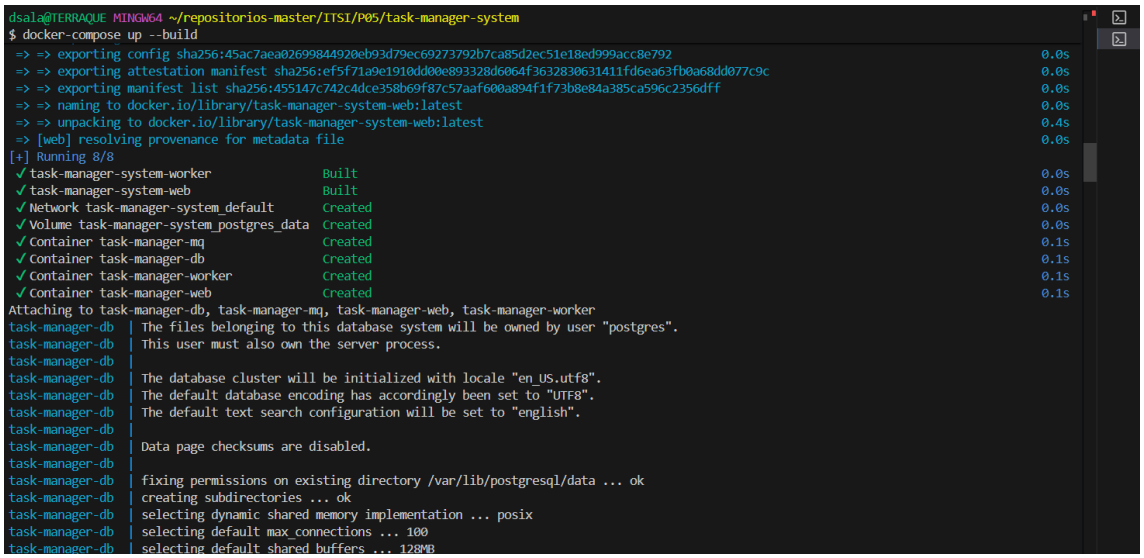
Dockerfile

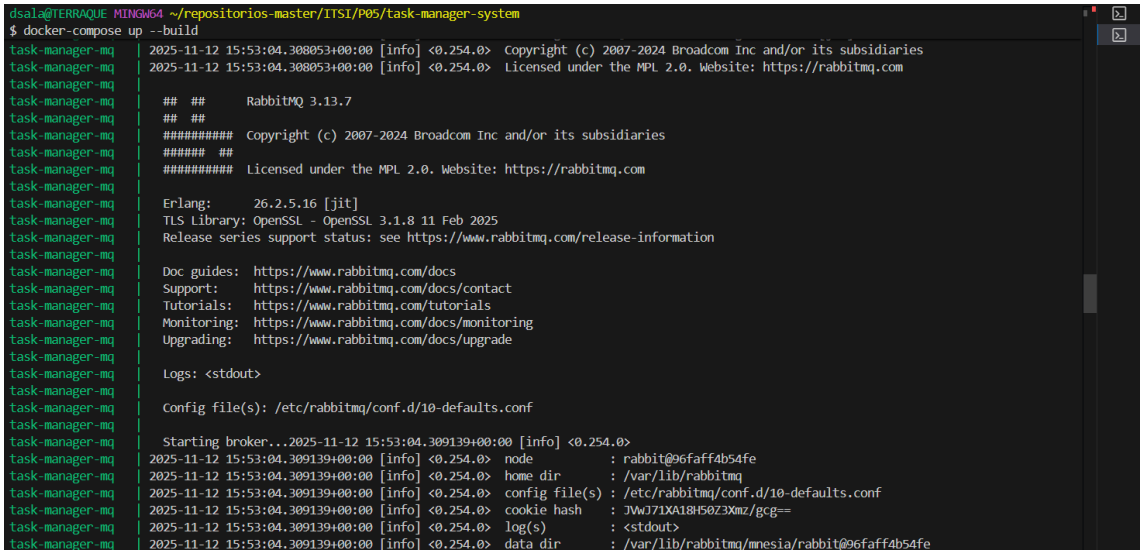
worker > Dockerfile > ...

```
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.



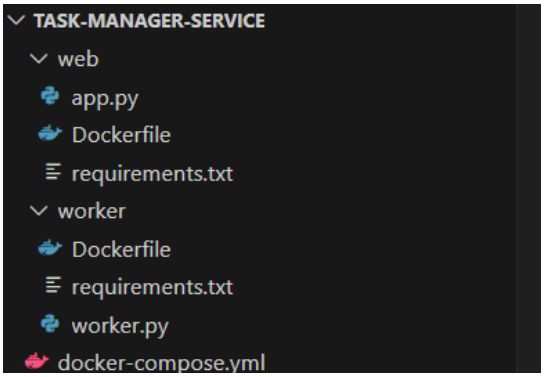


4. Ejercicios Propuestos

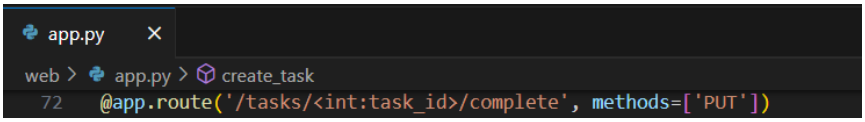
4.1. Ejercicio 1: Actualización de Tareas y Notificación (Dificultad: Baja)

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:

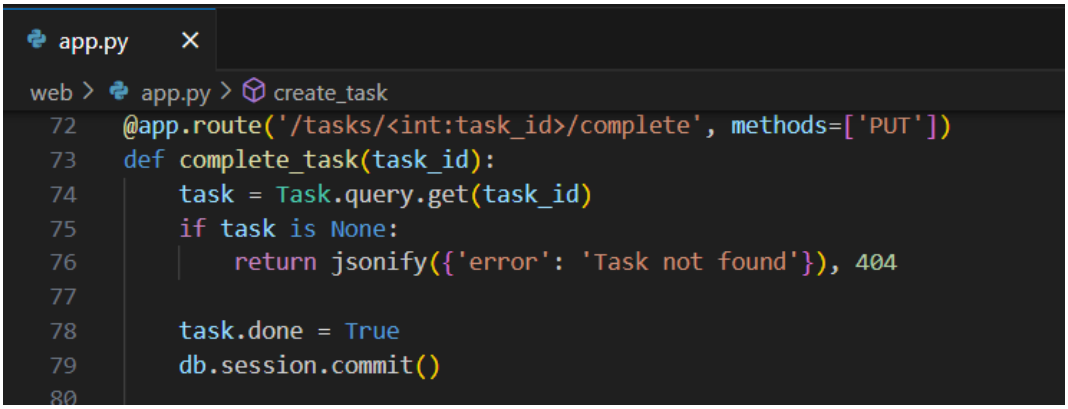


Se crea un nuevo endpoint PUT /tasks/<int:task_id>/complete en app.py.



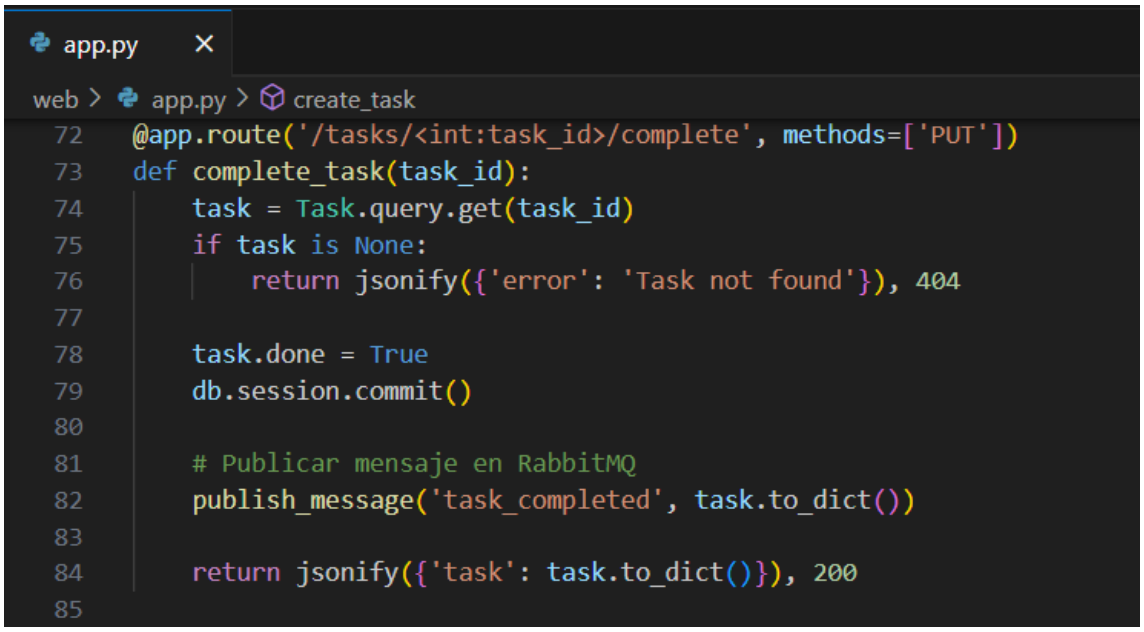
2. *Este endpoint debe encontrar la tarea en la base de datos, cambiar su estado done a true y guardar el cambio.*

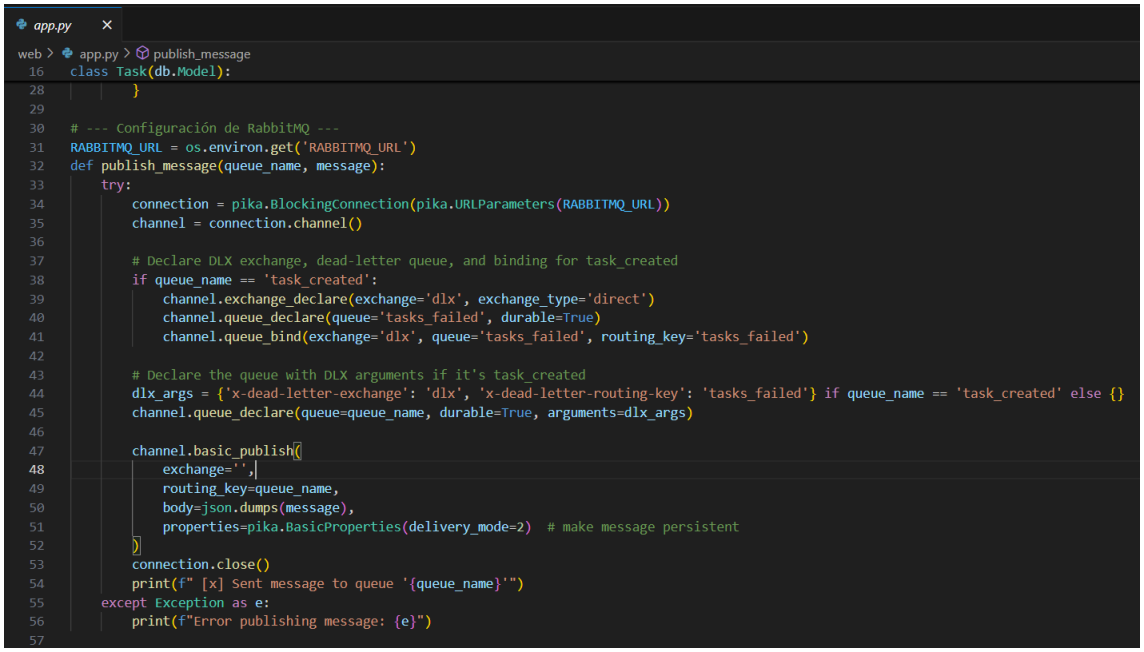
Se añade al endpoint la lógica de la función complete_task, la cual busca la tarea en la base de datos usando Task.query.get(task_id). Si la encuentra, actualiza su estado (task.done = True) y guarda el cambio en la base de datos con db.session.commit().



3. *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.

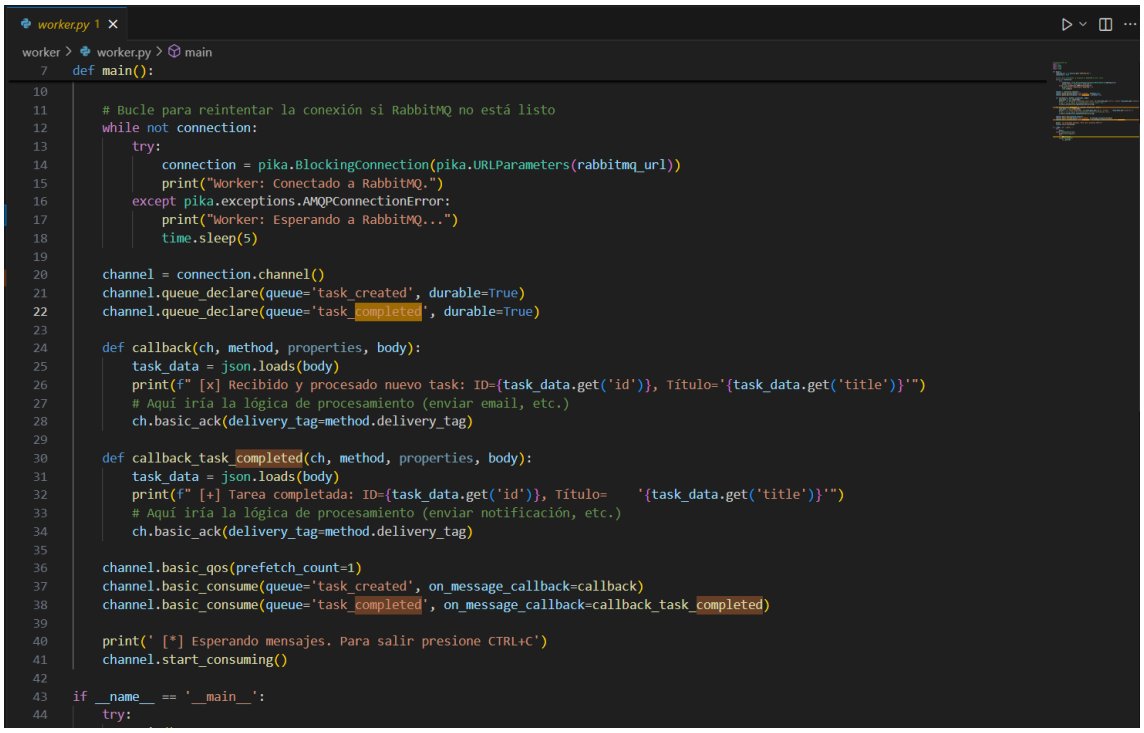




4. *Modifique el worker/worker.py para que también escuche en la cola task_completed y muestre un mensaje de log diferente, como [+] Tarea completada: ID={...}.*

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

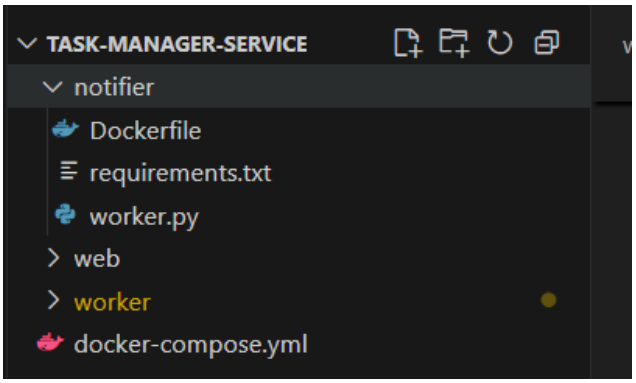
- **Declaración**: Declara la nueva cola task_completed.
- **Nuevo Callback**: Se añade una nueva función callback_task_completed que procesa los mensajes de esta nueva cola, imprimiendo un log: Tarea completada....
- **Consumo Dual**: El worker ahora consume de *dos* colas: task_created (usando callback)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



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

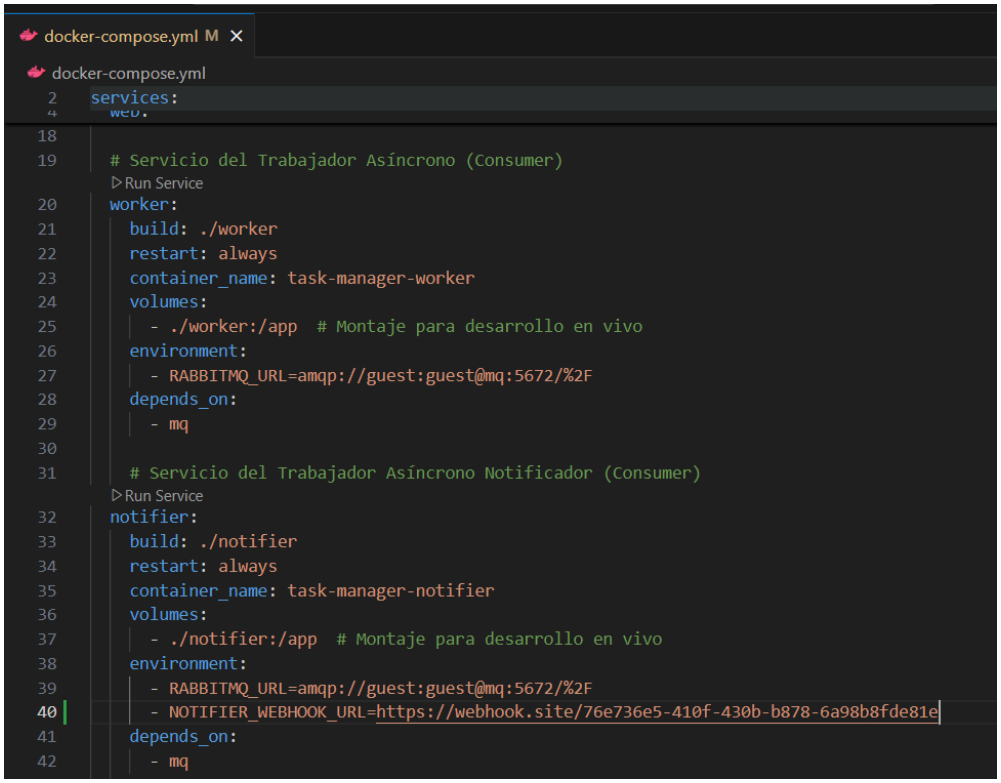
1. *Cree un nuevo directorio notifier/ con su propio worker.py, requirements.txt y Dockerfile.*

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.



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

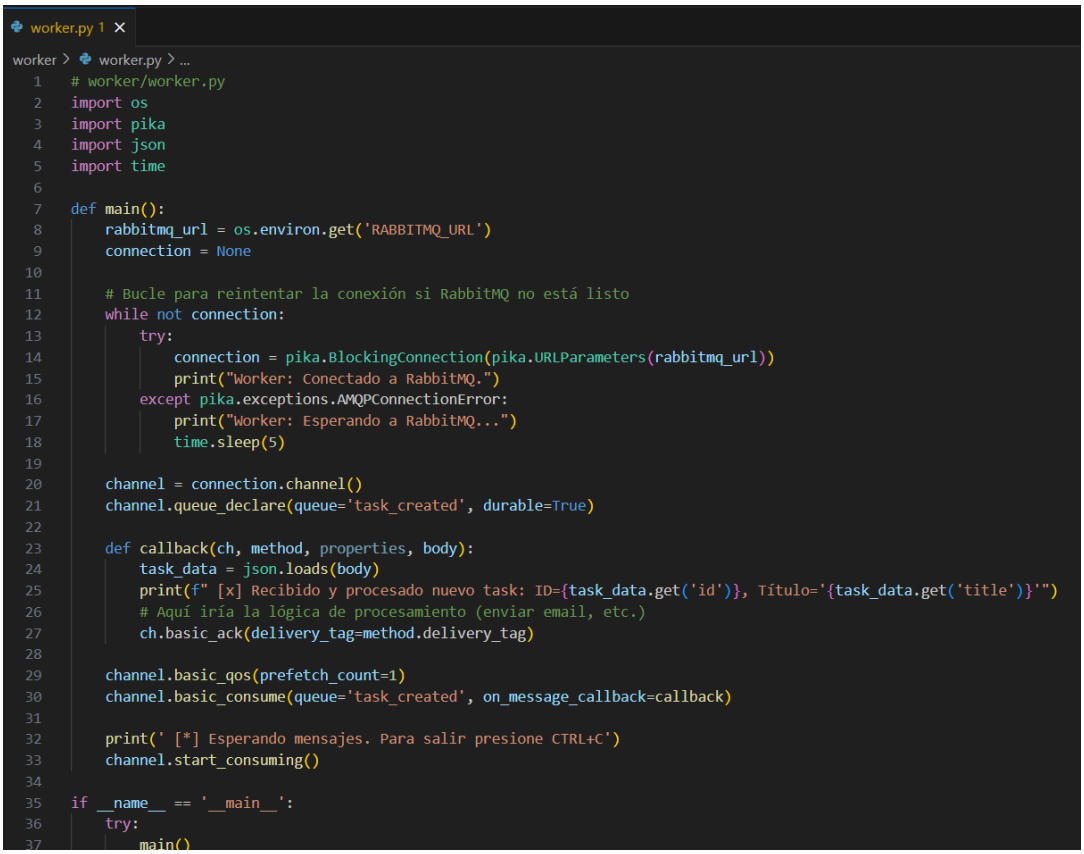
El código para el nuevo servicio notifier se muestra. Inicialmente, este código es una copia del worker modificado del Ejercicio 1, ya que escucha en ambas colas (task_created y task_completed). La intención es que este servicio maneje las notificaciones.

```
# 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.



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.*

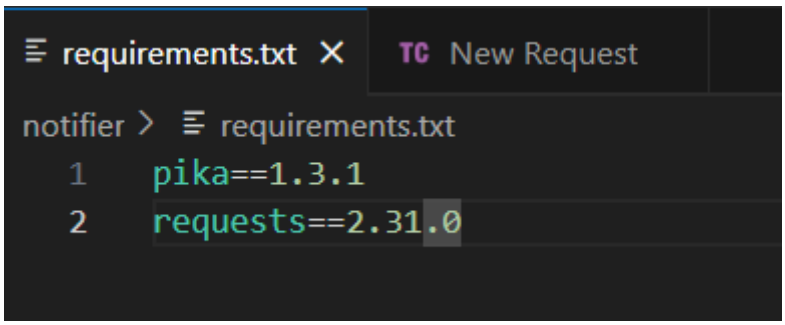
Se añade una variable webhook_url que se lee del entorno (NOTIFIER_WEBHOOK_URL).

```
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.



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.

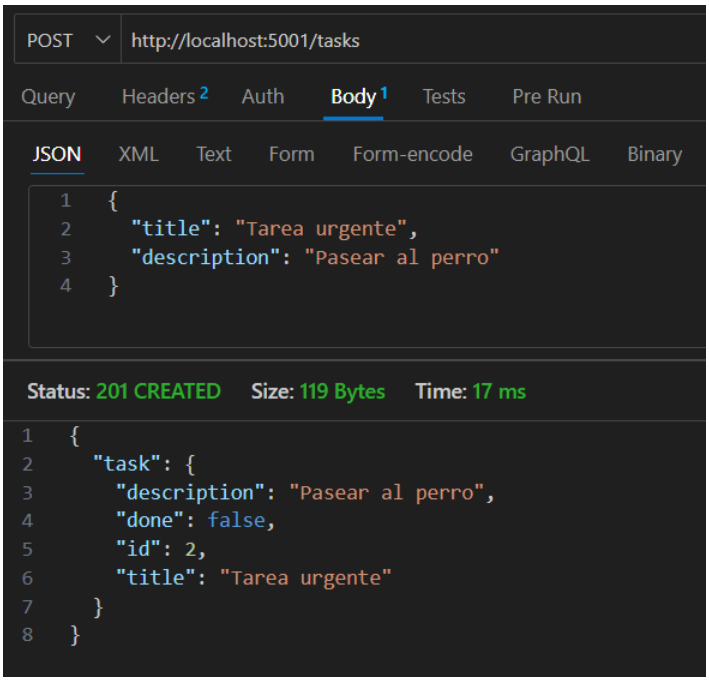
Cuando recibe un mensaje (ya sea de una tarea nueva o una completada), prepara un payload JSON y lo envía a una URL de webhook externa (simulando el envío de un email o una alerta).

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.

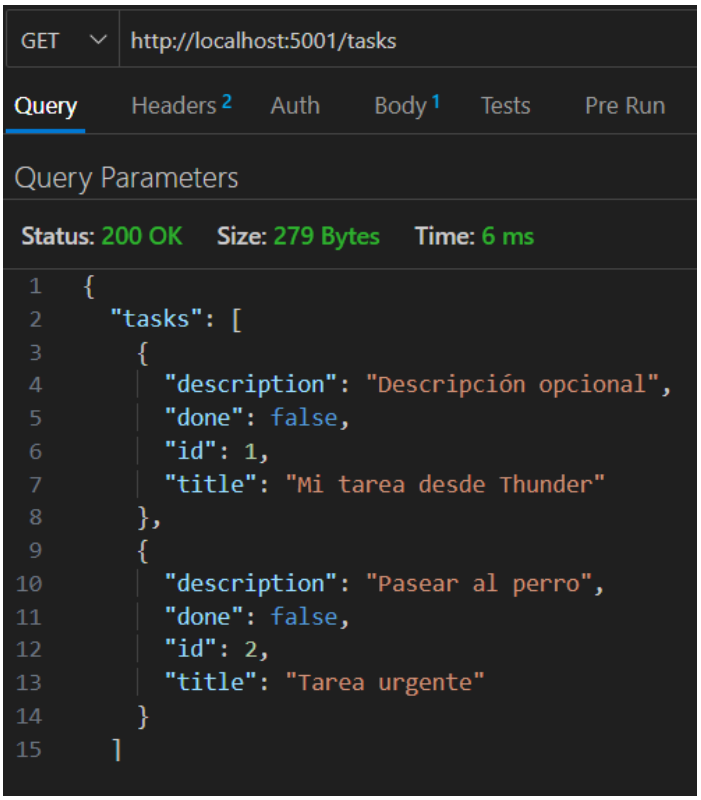
```
# notifier/worker.py
import os
import pika
import json
import time
import requests
import sys

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/xxxxx)
    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)
```

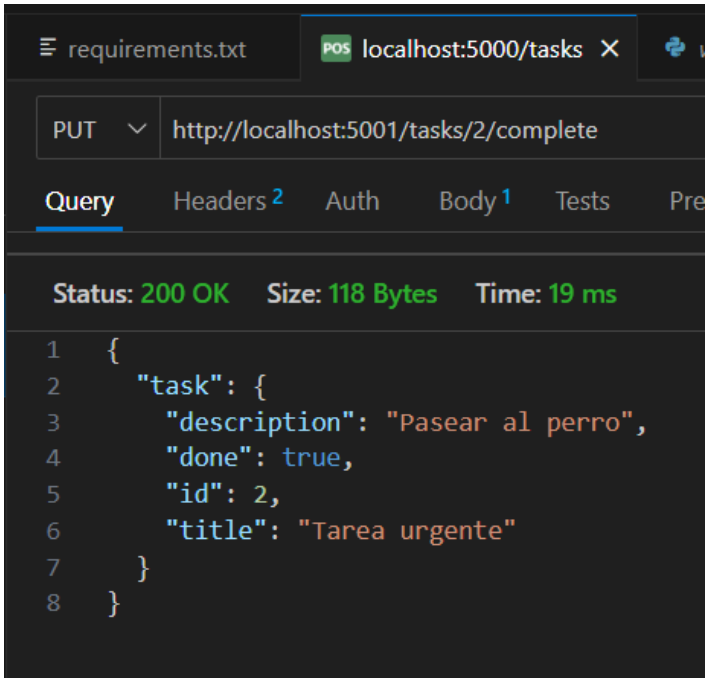
Para estas pruebas básicas de las operaciones POST, GET y PUT se ha utilizado la extensión Thunder Client de Visual Code. En primera instancia, vemos como se utiliza, de forma satisfactoria, la ruta POST /tasks para crear la "Tarea urgente" (ID 2). Responde con 201 CREATED.



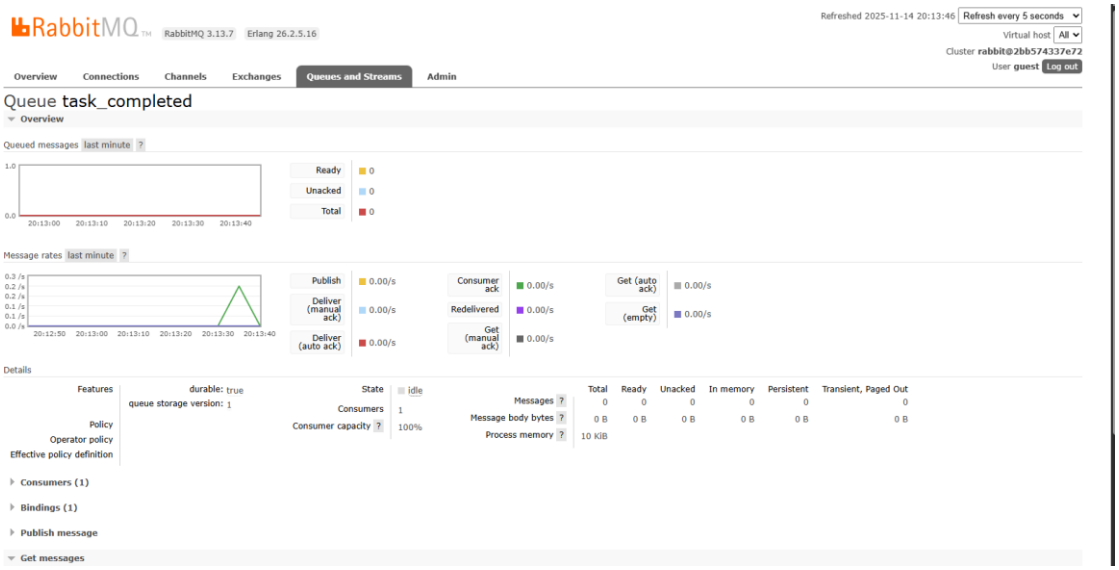
GET /tasks para listar las tareas. Muestra la tarea 1 y la nueva tarea 2.



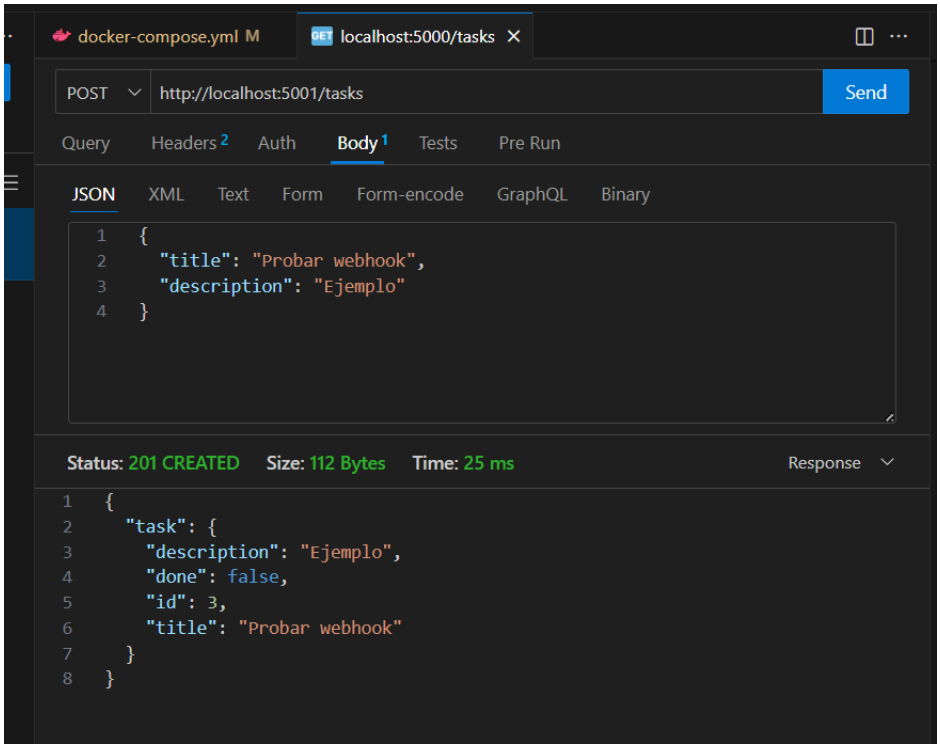
PUT /tasks/2/complete para completar la tarea. Responde con 200 OK y muestra done: true.



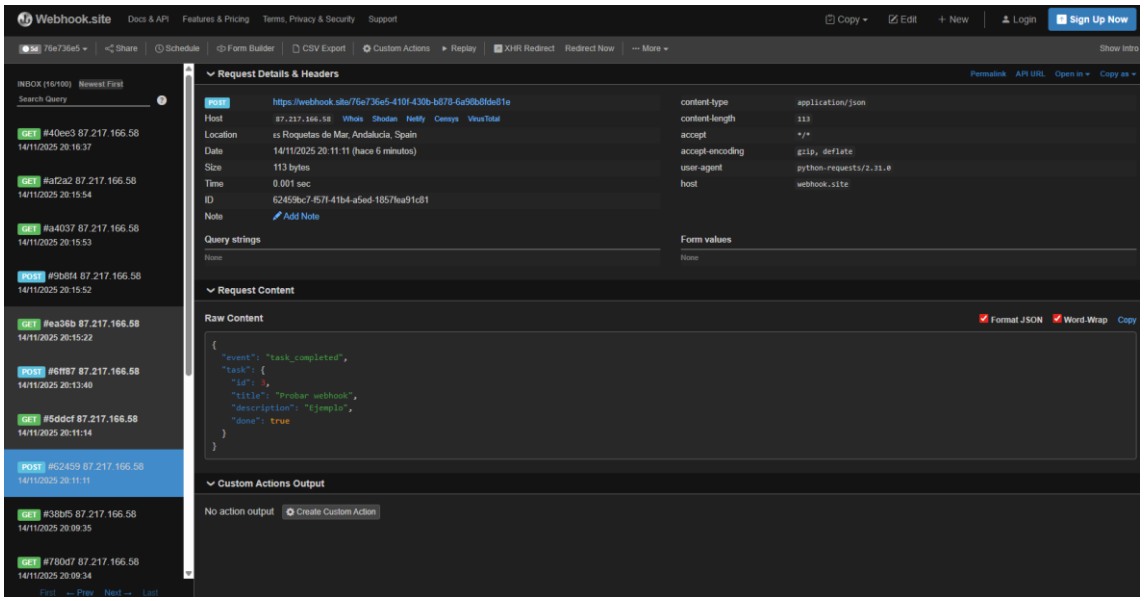
Desde la interfaz de RabbirMQ, podemos observar que el mensaje anterior ha sido recibido por la cola task_completed, al marcar la tarea como completada mediante el PUT.



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__)
```

```python
# --- 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/xxxxx)
    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.

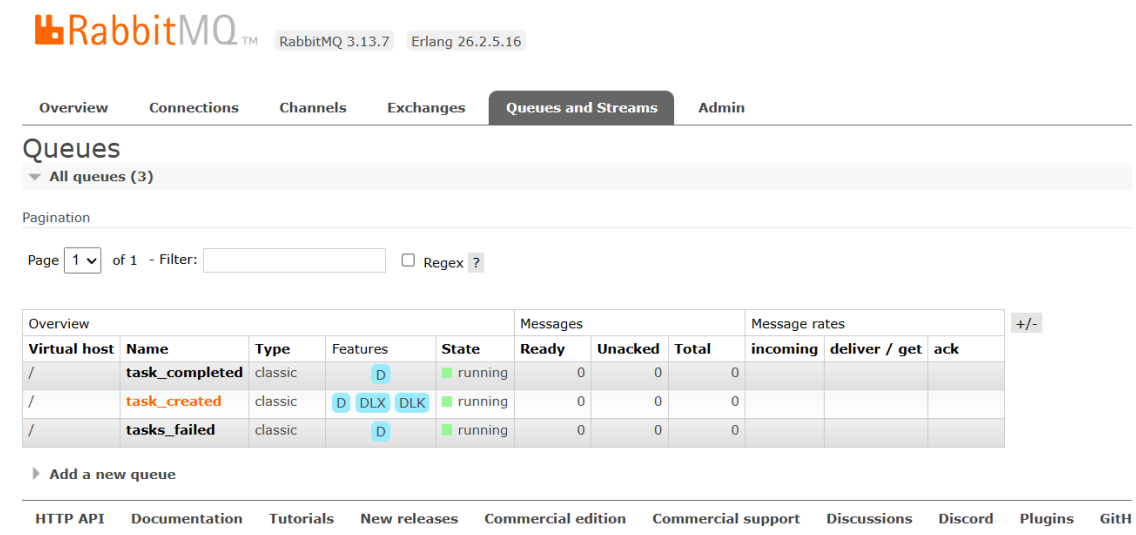
```
def callback(ch, method, properties, body):
    task_data = json.loads(body)

    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.

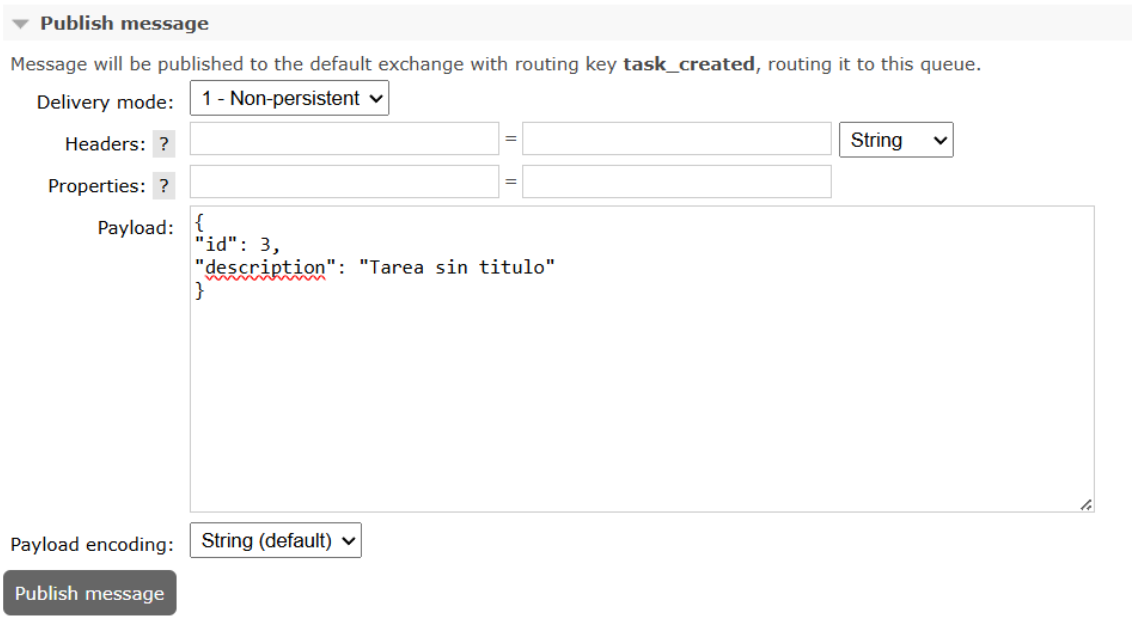
```
def callback(ch, method, properties, body):
    task_data = json.loads(body)

    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.*

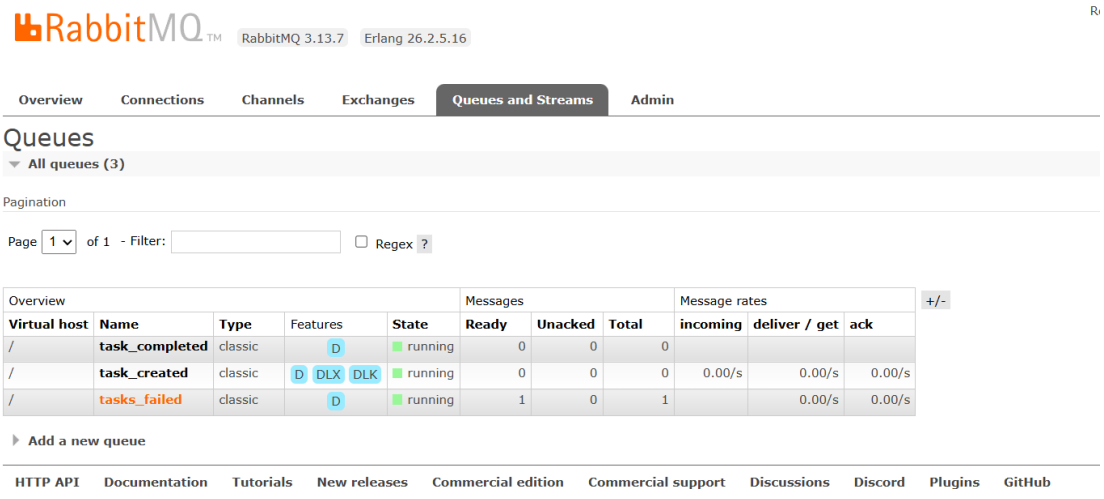
Iniciamos la web local de la interfaz de RabbitMQ. En la pestaña “Queues and Streams” vemos que existen 3 colas: task_completed, task_created (con las etiquetas D de durable y DLX), y tasks_failed. Seleccionamos trask_created para crear una tarea.



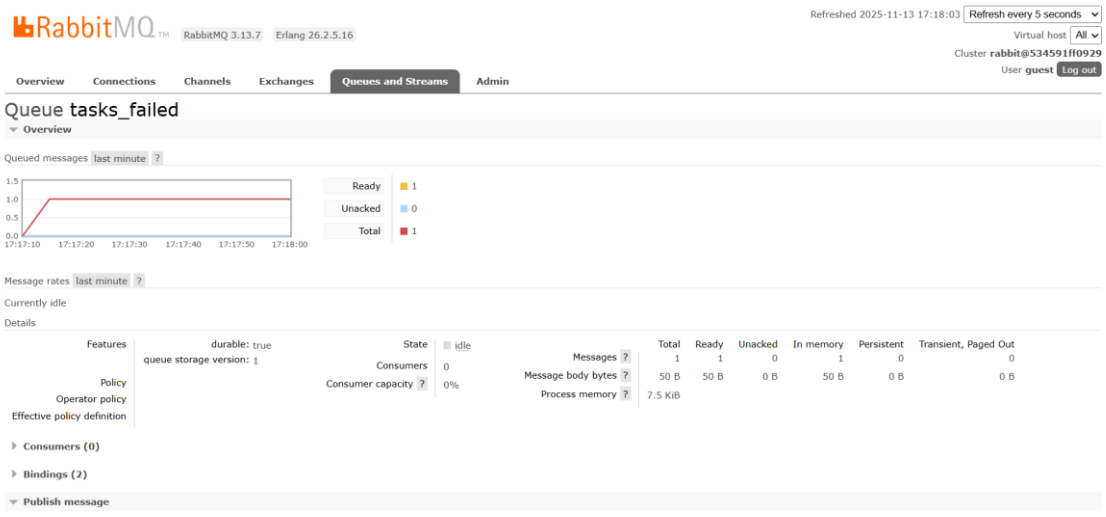
Se publica manualmente un mensaje en task_created que no tiene el campo title:



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



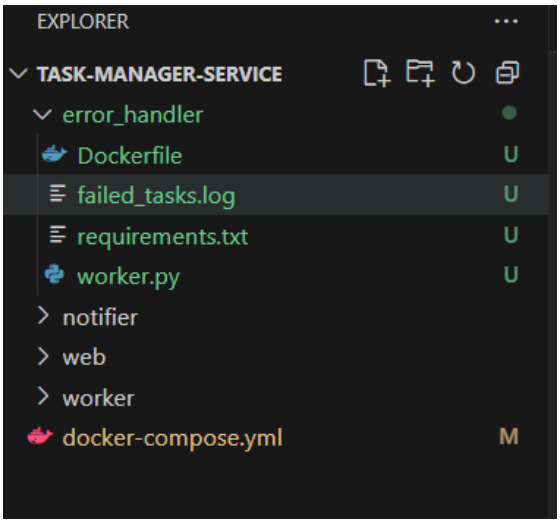
Inmediatamente, la cola tasks_failed muestra un "Total" de 1 mensaje, indicando que el rechazo y el DLX funcionaron.

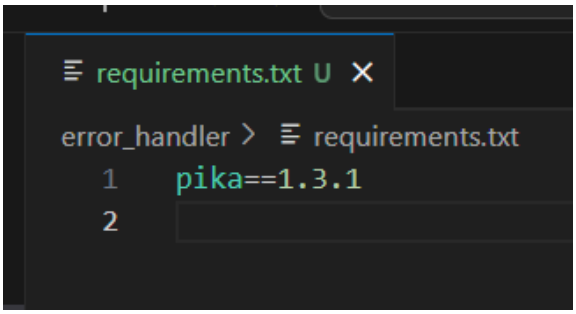


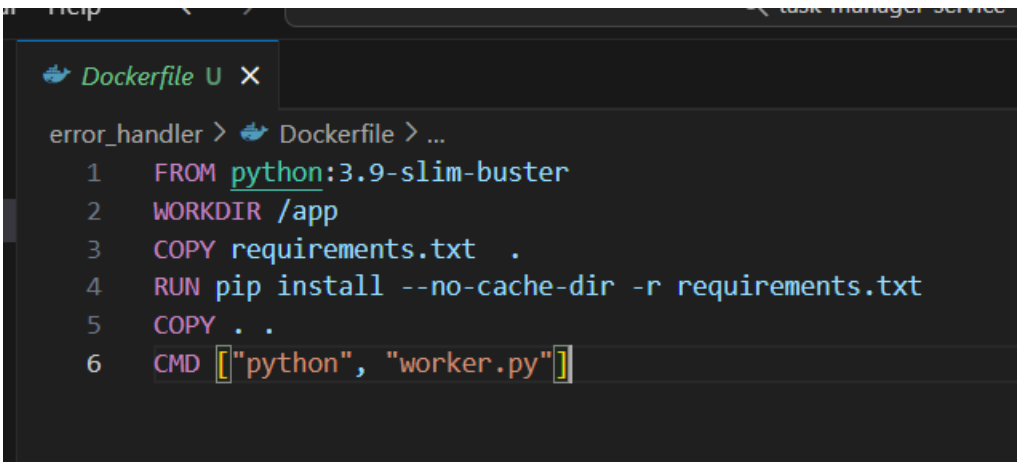
Al ir a "Get Message(s)" en la cola, se puede inspeccionar el mensaje. Verificamos que el payload es el del mensaje anterior.

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.







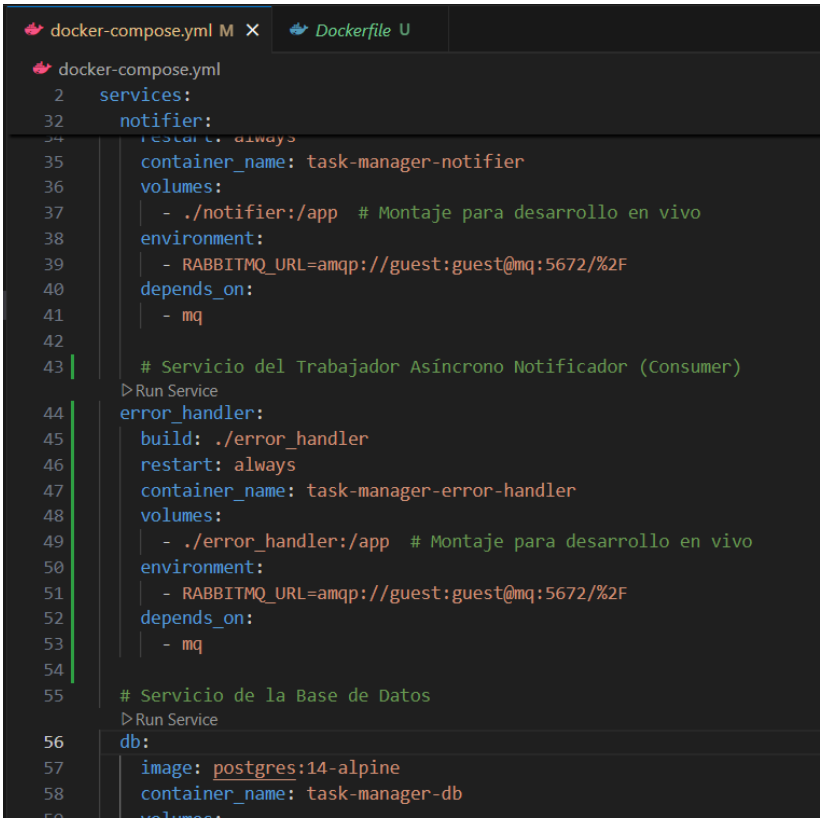
Se muestra el código para este nuevo consumidor.

- **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.

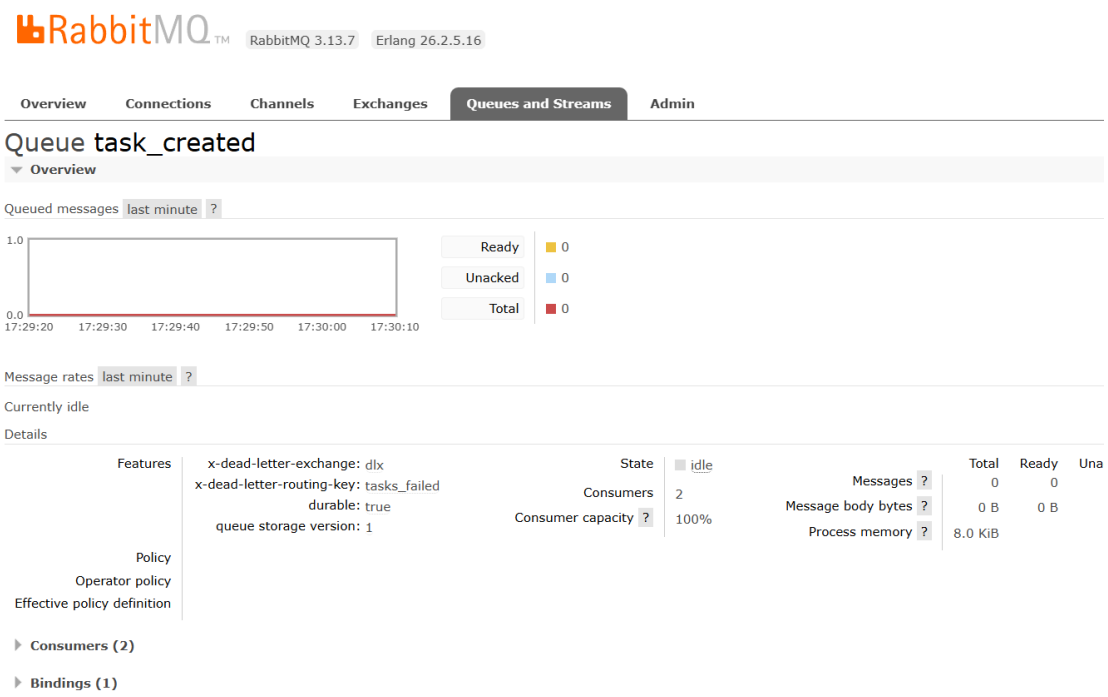
```
# error_handler/worker.py
import os
import pika
import json
import time
import logging
import sys

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'
    )
```

```python
    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)
```

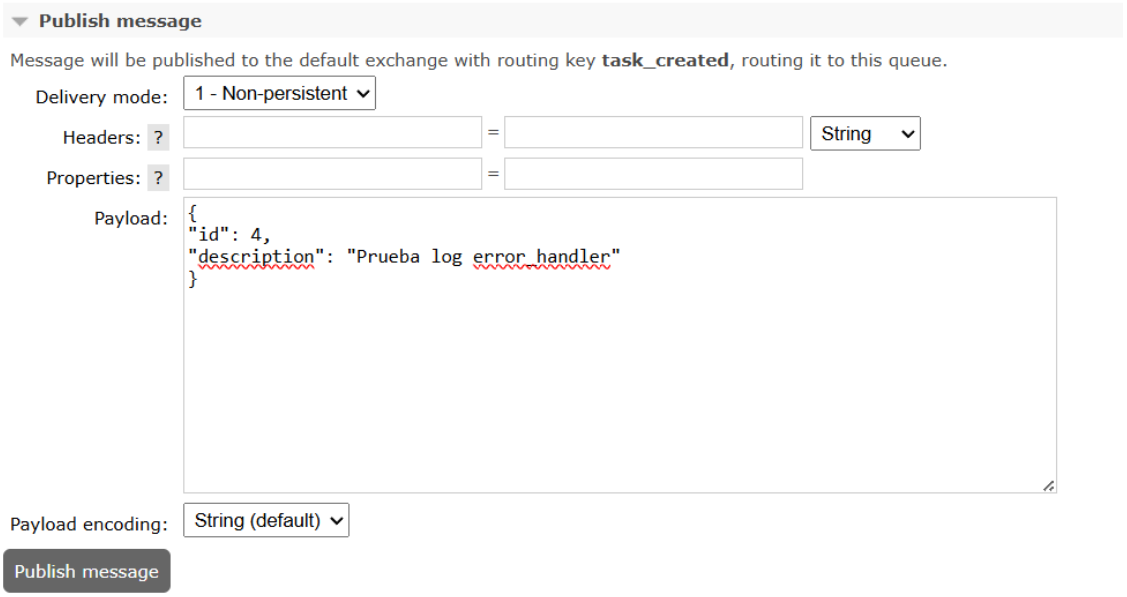
Se añade el nuevo servicio error_handler al docker-compose.yml, configurado de forma similar a los otros workers.



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

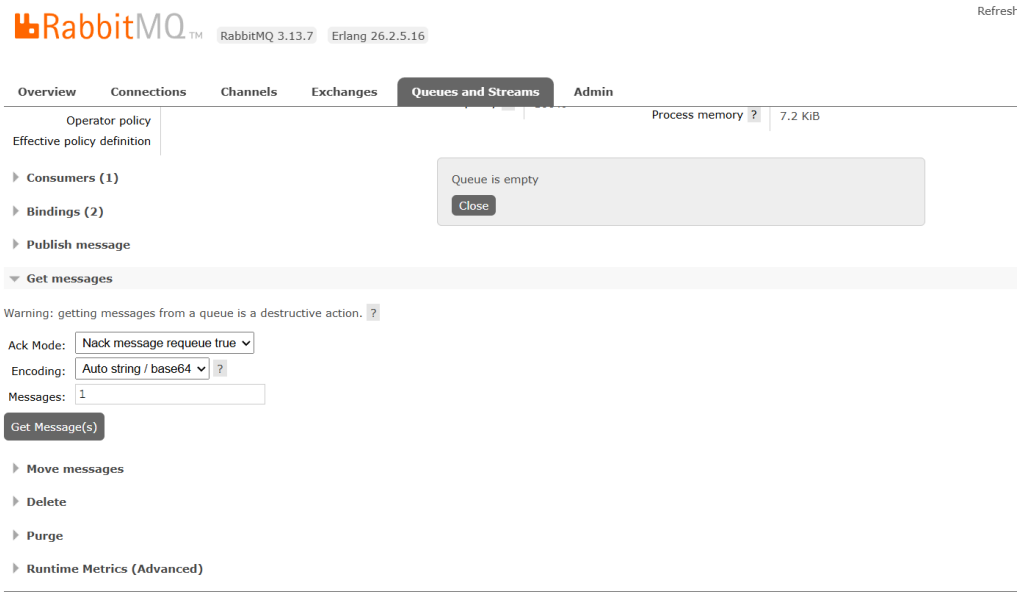


Se publica otro mensaje sin especificar el campo título, en el apartado Publish message.



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.



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