Memoria Práctica 5

Jaime Parra Jiménez

INTEGRACIÓN DE TECNOLOGÍAS Y SERVICIOS INFORMÁTICOS

Universidad de Almería

Correo: jpj451@inlumine.ual.es

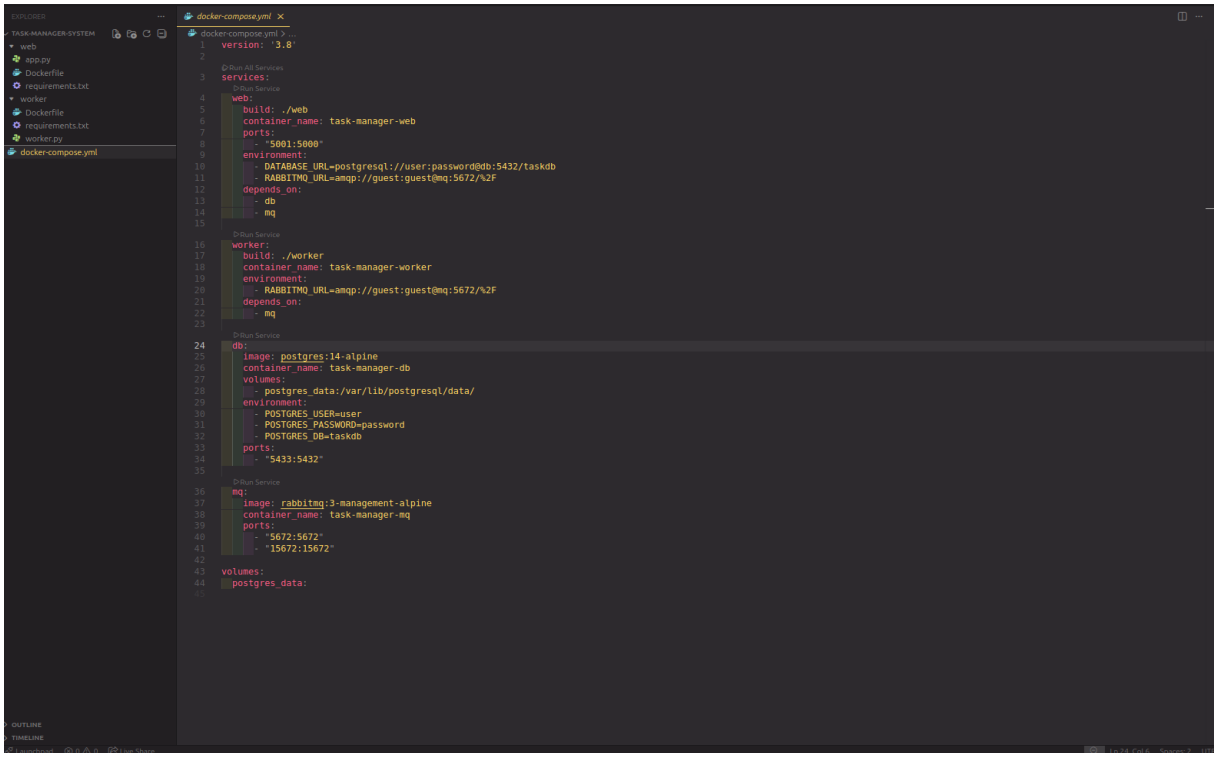
19 de noviembre de 2025

Índice general

Capítulo 1

Ejercicio Guiado 1

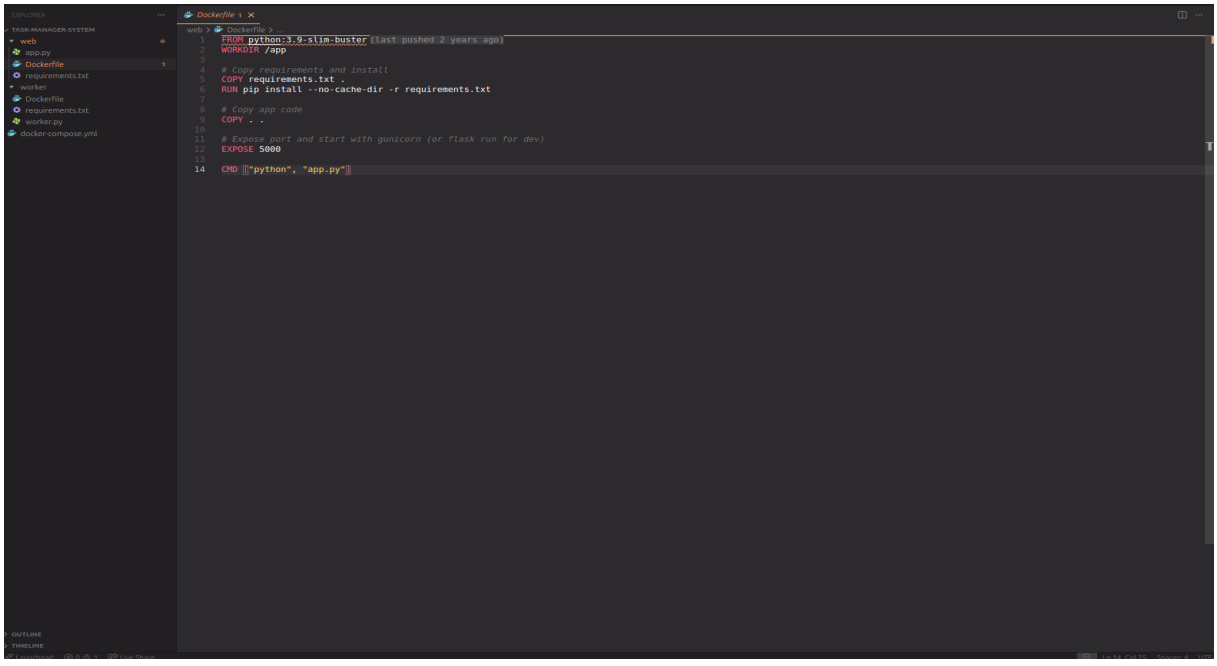
El objetivo del ejercicio guiado es transformar un microservicio único en una aplicación de tres componentes: una API web, una base de datos PostgreSQL y un trabajador asíncrono, todos orquestados por Docker Compose. Para ello será necesario crear varios ficheros. Primero, se crea el fichero docker-compose.yml en la raiz de nuestro proyecto. Este fichero será el centro de control de nuestra aplicación.



Una vez creado el fichero, el siguiente paso es crer un carpeta llamada web. Dentro de ella tendremos los ficheros requirements.txt, Dockerfile y app.py. Primero configuraremos app.py, que es el archivo principal de la aplicación.

```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()
        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)  # persistente
        )
        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 ---
# GET /tasks - obtener todas las tareas
@app.route('/tasks', methods=['GET'])
def get_tasks():
    tasks = Task.query.all()
    return jsonify({'tasks': [task.to_dict() for task in tasks]})

# POST /tasks - crear una nueva tarea
@app.route('/tasks', methods=['POST'])
def create_task():
    if not request.json or 'title' not 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

#... puedes añadir más endpoints como GET /tasks/<id>, DELETE, PUT, etc.

if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(host='0.0.0.0', port=5000, debug=True)
```

A continuación, creamos el Dockerfile que es el encargado de crear la imagen de Docker del contenedor.

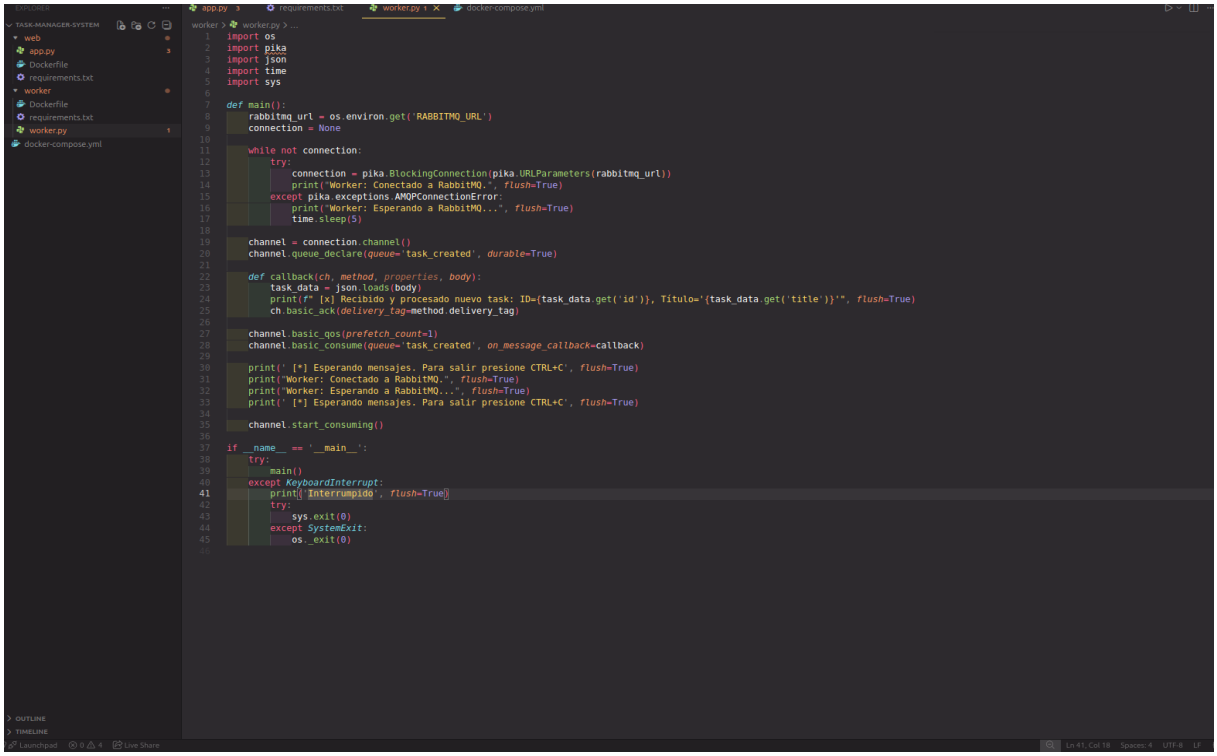


```
FROM python:3.9-slim-buster (last pushed 2 years ago)
WORKDIR /app

# Copy requirements and install
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy app code
COPY . .

# Expose port and start with gunicorn (or flask run for dev)
EXPOSE 5000

CMD ["python", "app.py"]
```

Por último, creamos el archivo requirements.txt donde se definiran las dependencias de Python que se instalaran en el contenedor.

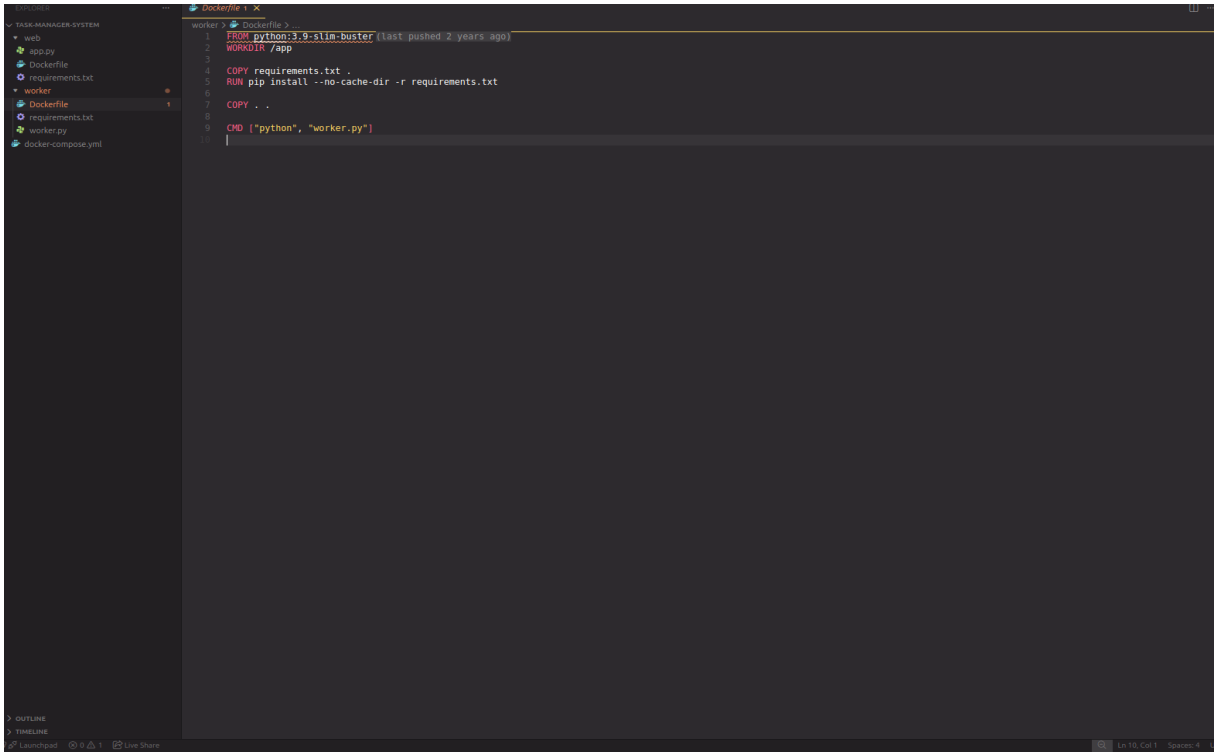
```
Flask==2.2.2
Flask-SQLAlchemy==3.0.3
psycopg2-binary==2.9.5
pika==1.3.1
Werkzeug==2.2.2
```

Una vez creada la web es paso de crear el worker, nuestro consumidor de mensajes. Para ello creamos en la raiz del proyecto una carpeta worker, la cual tendrá los mismos ficheros que web. Primero configuraremos worker.py, donde se conectará a RabbitMQ, escuchará mensajes en la cola y los procesará.

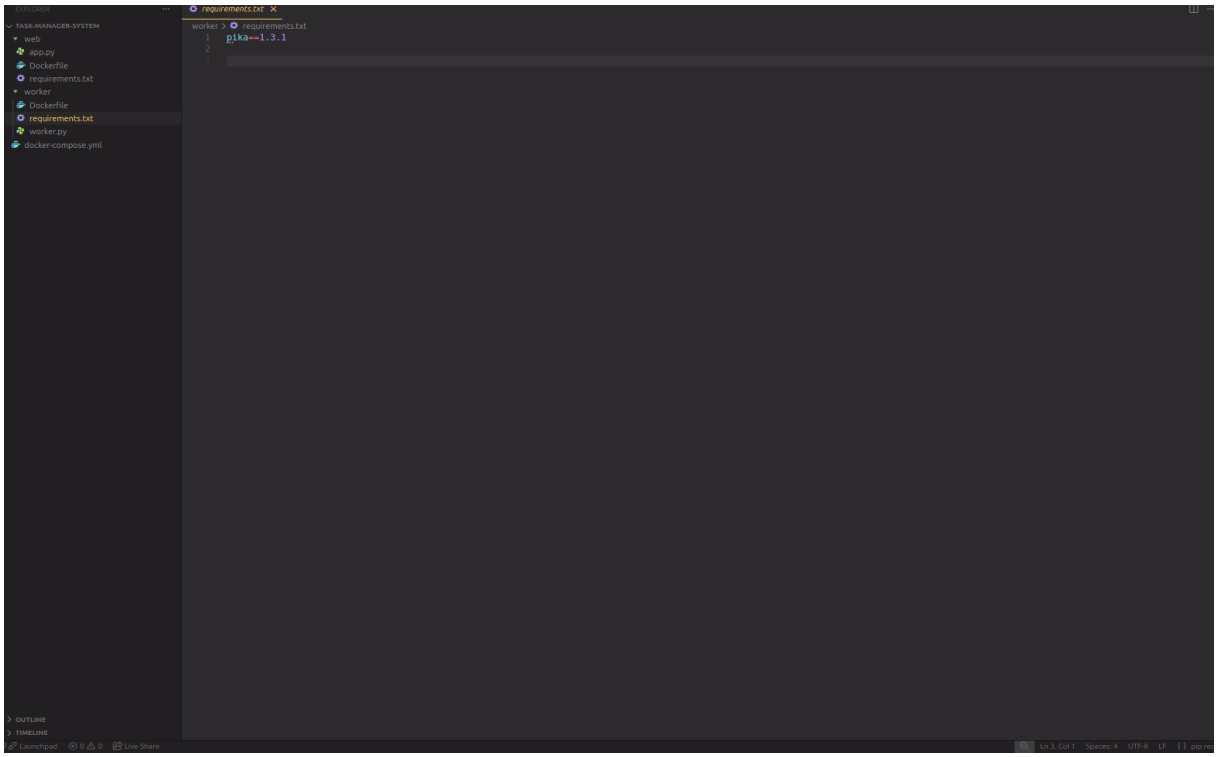


```
import os
import pika
import json
import time
import sys

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

    while not connection:
        try:
            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.", flush=True)
        except pika.exceptions.AMQPConnectionError:
            print('Worker: Esperando a RabbitMQ...', flush=True)
            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')}, Titulo='{task_data.get('title')}'", flush=True)
        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', flush=True)
    print("Worker: Conectado a RabbitMQ.", flush=True)
    print("Worker: Esperando a RabbitMQ...", flush=True)
    print(' [*] Esperando mensajes. Para salir presione CTRL+C', flush=True)

    channel.start_consuming()

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

El dockerfile será igual que el de web.

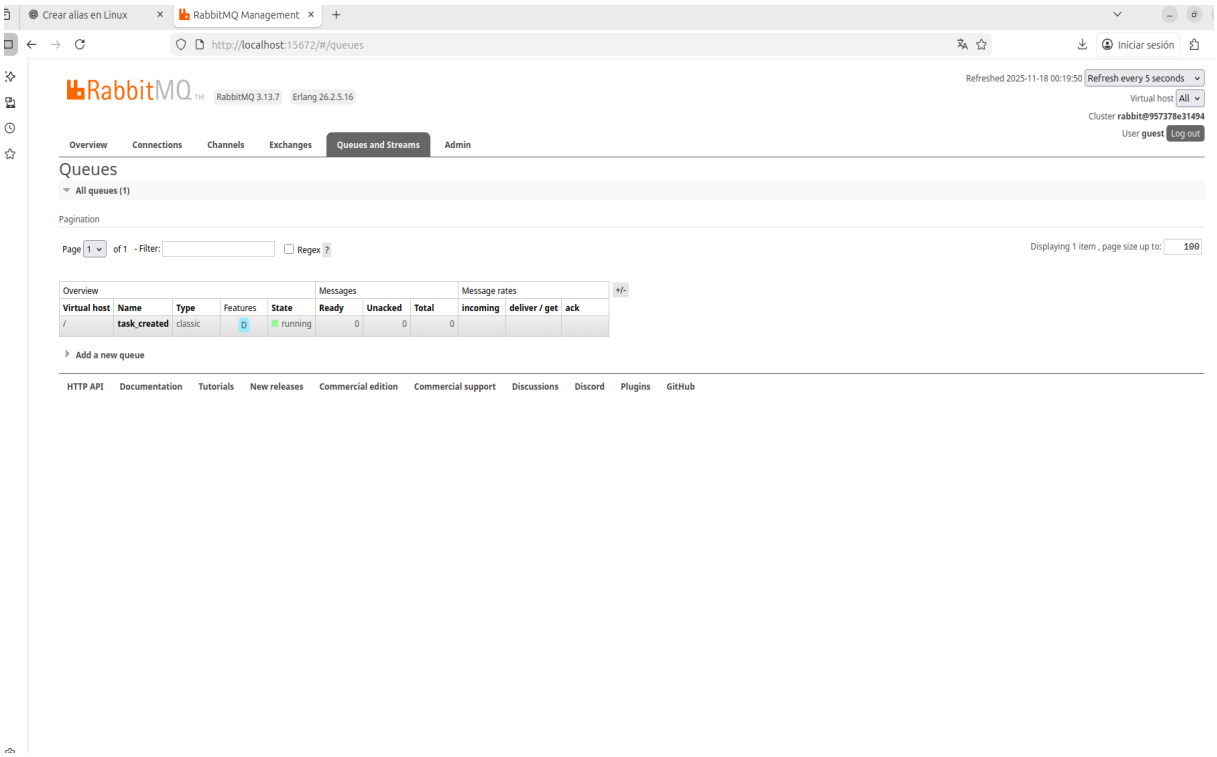


Y por útimo, configuramos requirements.txt para poner las dependencias.

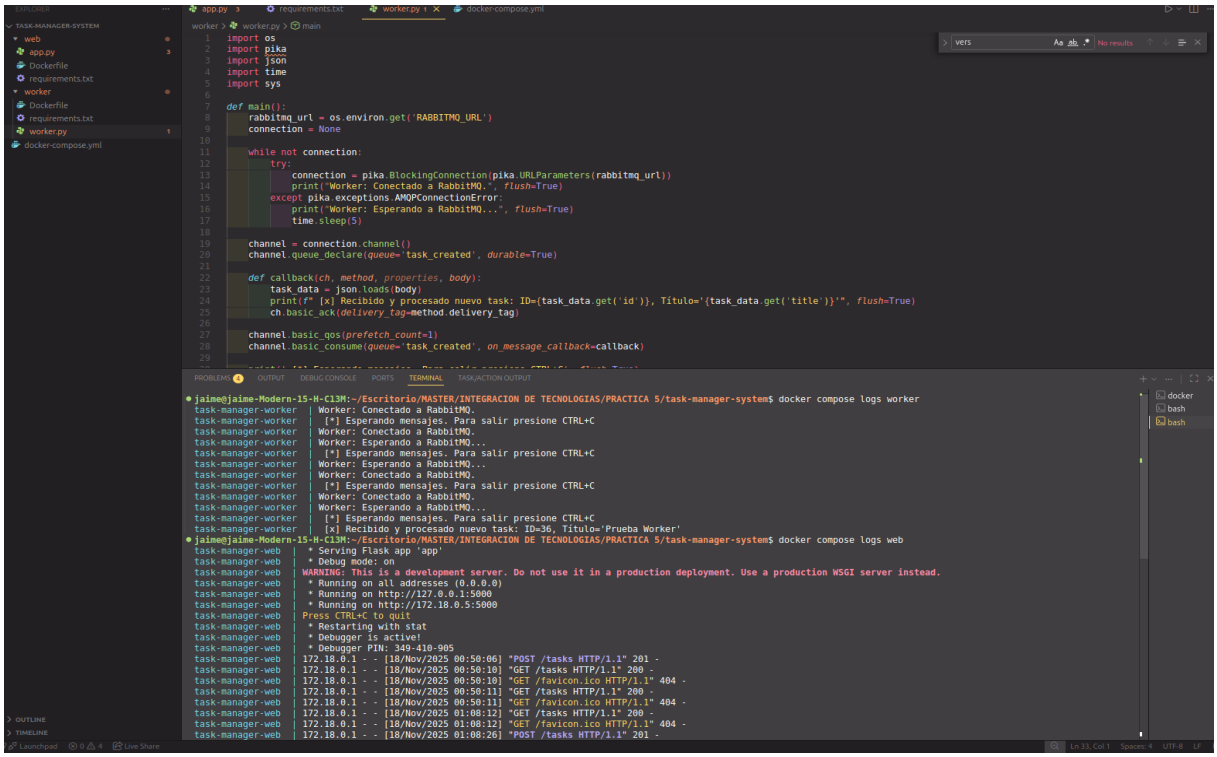


Ya configurados todos los archivos es hora de levantar la aplicación. Para ello debemos poner el siguiente comando docker-compose up –build en la terminal desde la raiz del proyecto.

También comprobaremos RabbitMQ desde el navegador para ver la cola creada.



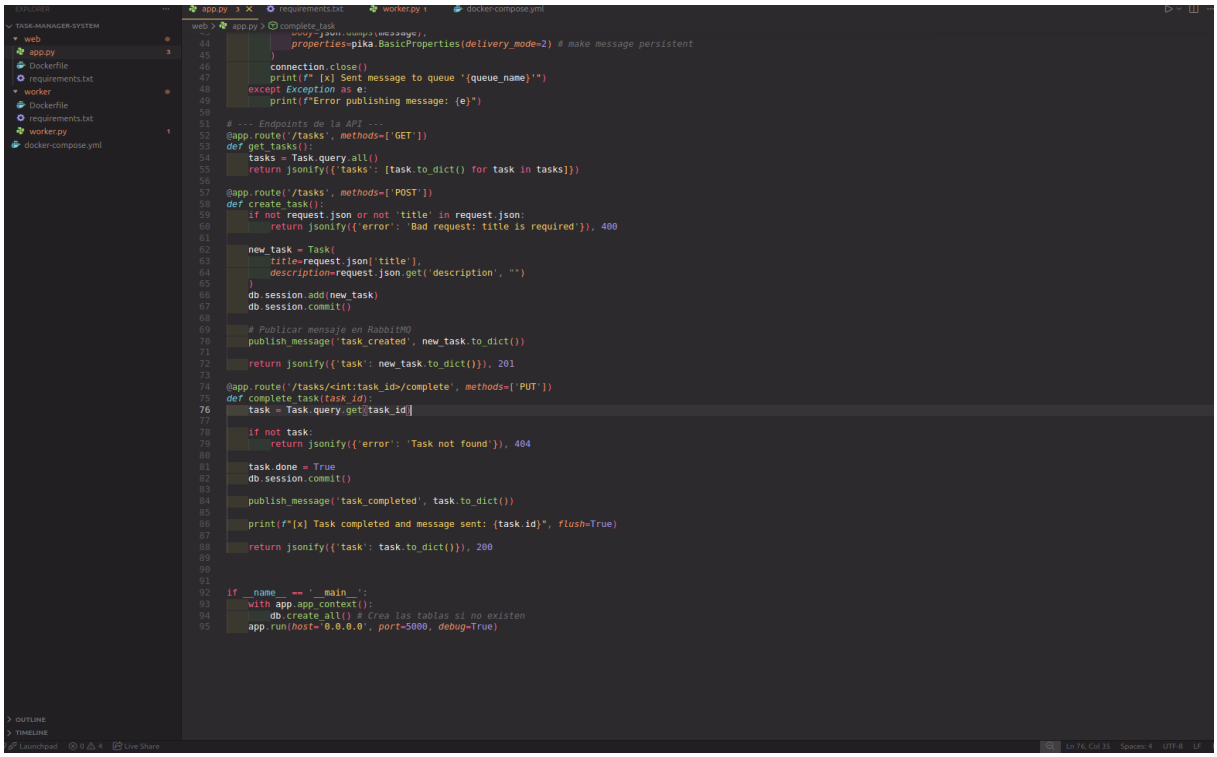
Por último, probaremos el flujo completo haciendo un curl para crear una nueva tarea en la API, si es correcta en el log de worker se podrá ver el mensaje de que se ha recibido la nueva tarea.



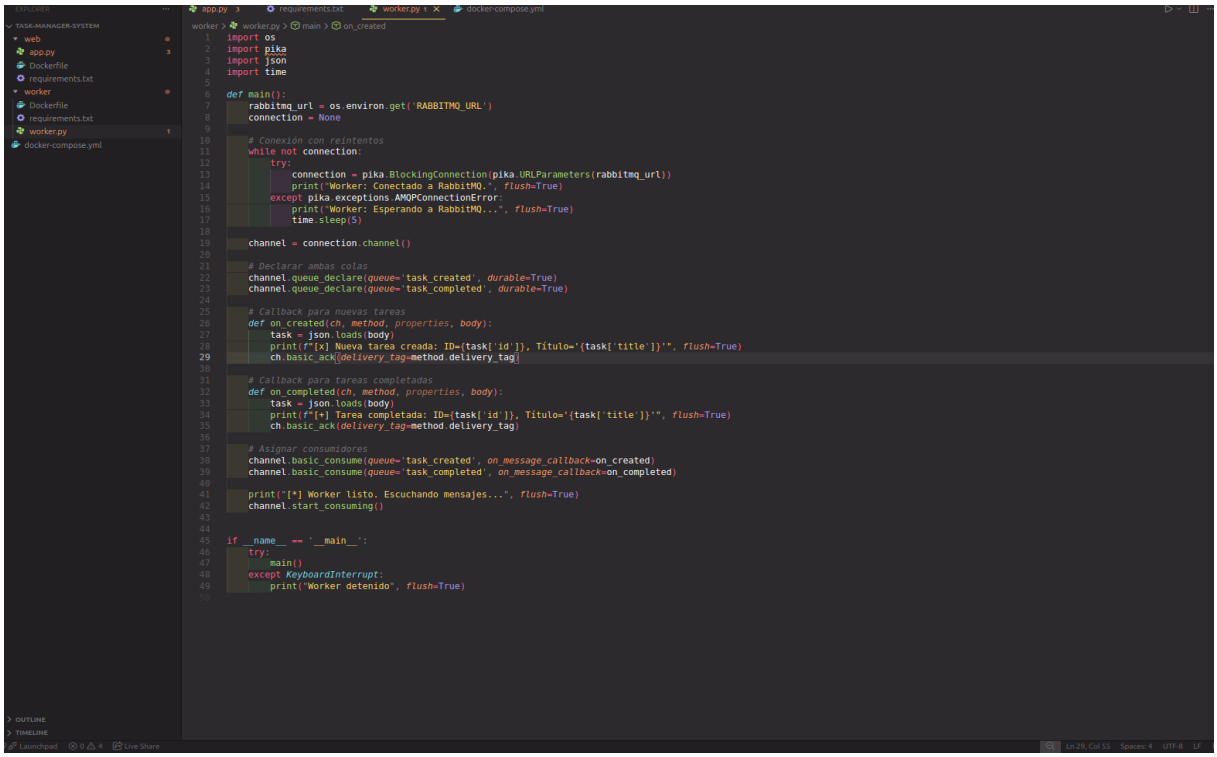
Capítulo 2

Ejercicio 1

El objetivo del ejercicio 1 es implementar la funcionalidad para marcar una tarea como completada y notificarlo a través de la cola de mensajes. Lo primero que se debe hacer es añadir un endpoint PUT en app.py para poder marcar una tarea como completada.

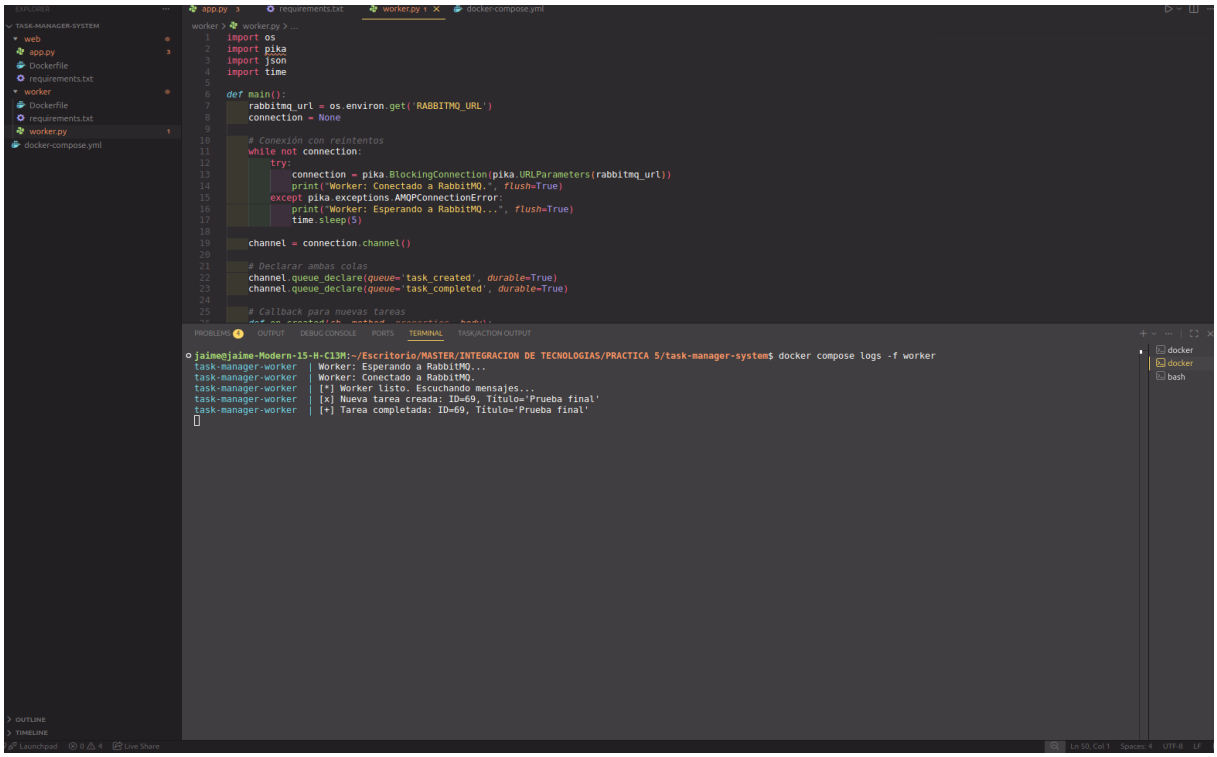


Además, será encesario modificar worker.py para crear una nueva cola llamda task-completed donde se publicará un mensaje con los datos de la tarea actualizada.



```python
import os
import pika
import json
import time

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

    # Conexión con reintentos
    while not connection:
        try:
            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.", flush=True)
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...", flush=True)
            time.sleep(5)

    channel = connection.channel()

    # Declarar ambas colas
    channel.queue_declare(queue='task_created', durable=True)
    channel.queue_declare(queue='task_completed', durable=True)

    # Callback para nuevas tareas
    def on_created(ch, method, properties, body):
        task = json.loads(body)
        print(f"[x] Nueva tarea creada: ID={task['id']}, Título='{task['title']}'", flush=True)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    # Callback para tareas completadas
    def on_completed(ch, method, properties, body):
        task = json.loads(body)
        print(f"[+] Tarea completada: ID={task['id']}, Título='{task['title']}'", flush=True)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    # Asignar consumidores
    channel.basic_consume(queue='task_created', on_message_callback=on_created)
    channel.basic_consume(queue='task_completed', on_message_callback=on_completed)

    print("[*] Worker listo. Escuchando mensajes...", flush=True)
    channel.start_consuming()


if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Worker detenido', flush=True)
```

Una vez modificado worker.py para poder ver el mensaje de la cola creada se podrá ver por terminal el mensaje.

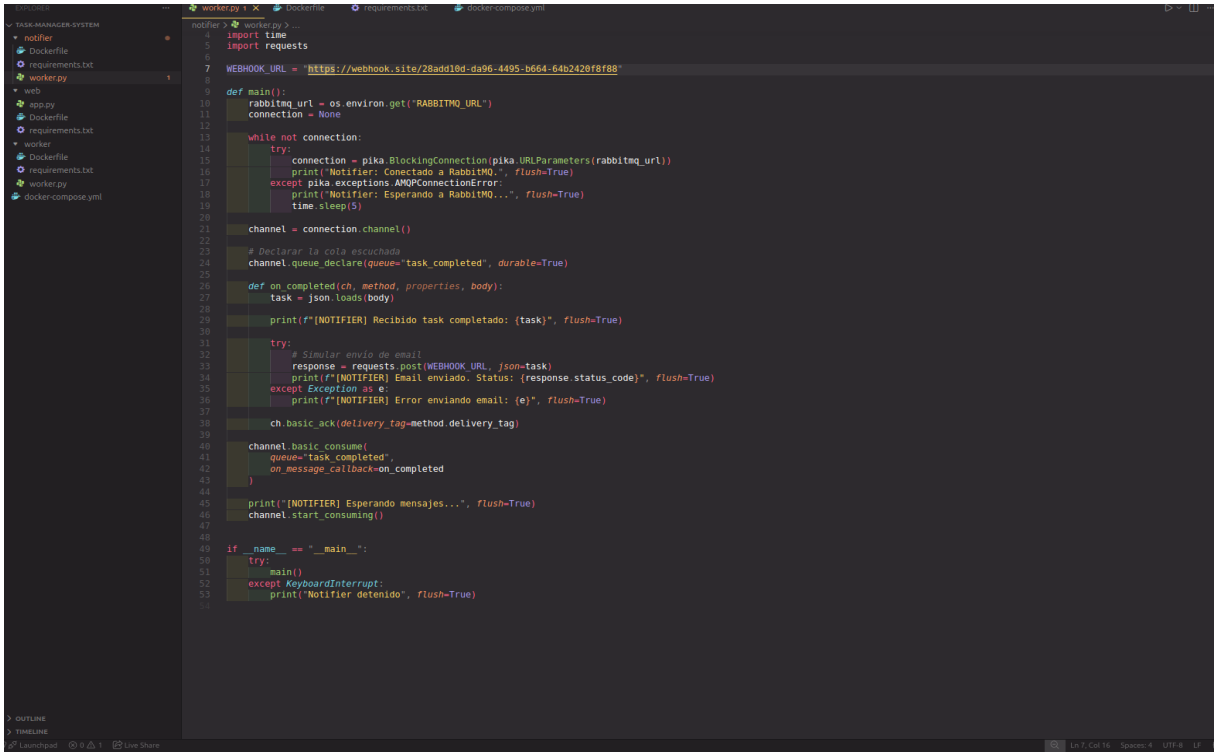


```
jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker compose logs -f worker
task-manager-worker  | Worker: Esperando a RabbitMQ...
task-manager-worker  | Worker: Conectado a RabbitMQ.
task-manager-worker  | [*] Worker listo. Escuchando mensajes...
task-manager-worker  | [x] Nueva tarea creada: ID=69, Título='Prueba final'
task-manager-worker  | [+] Tarea completada: ID=69, Título='Prueba final'
```

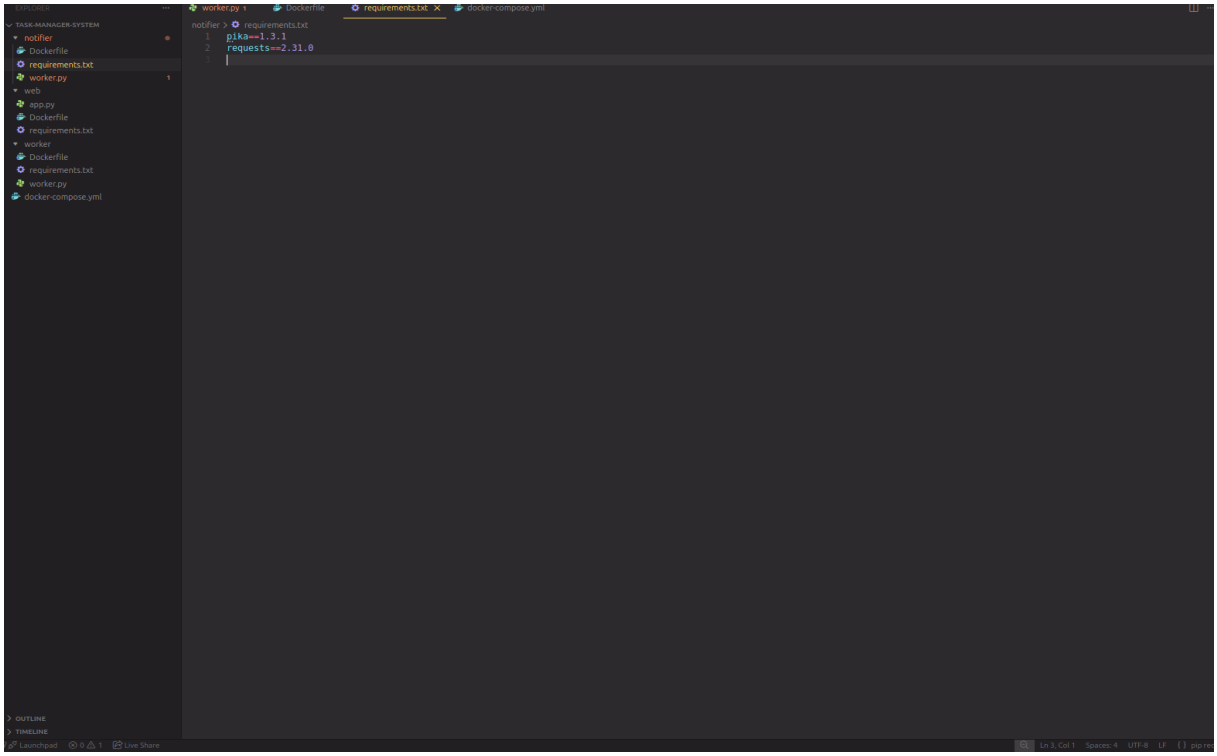
Capítulo 3

Ejercicio 2

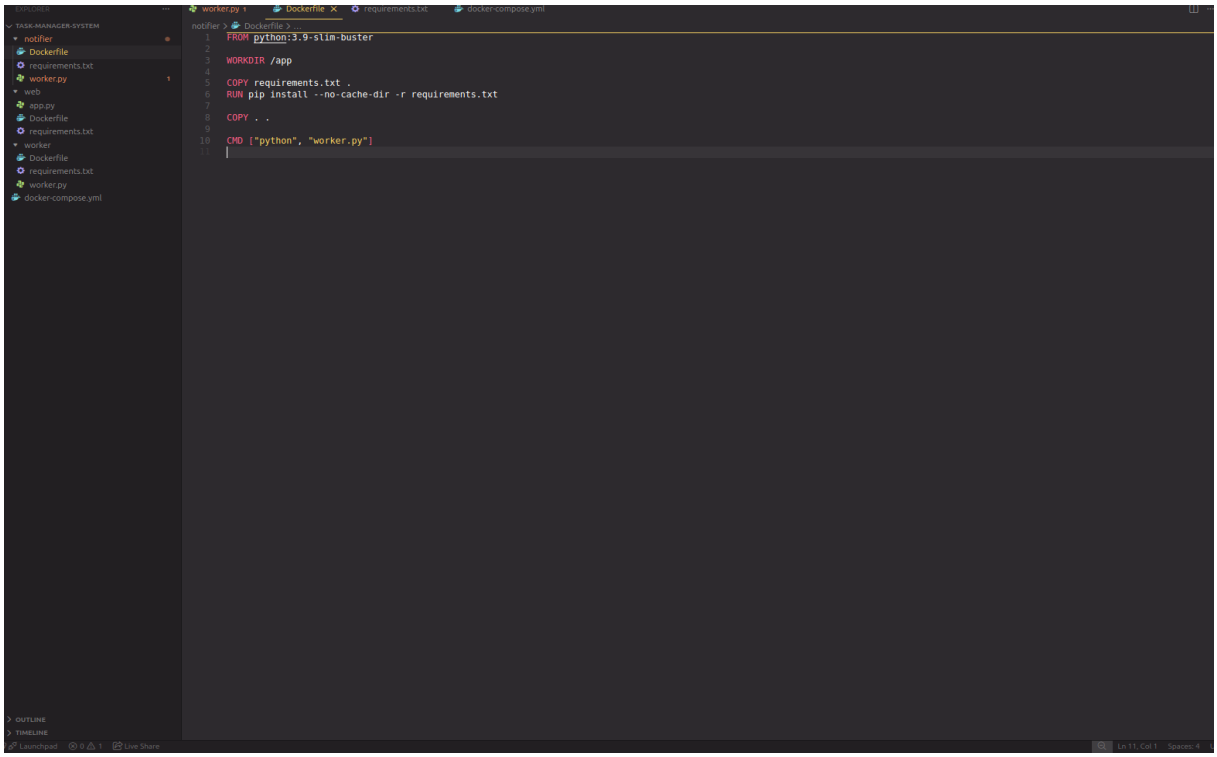
El objetivo del ejercicio 2 es crear un tercer servicio dedicado exclusivamente a enviar notificaciones, desacoplando aún más la lógica.Para ello, lo primero es crear un nuevo directorio llamado notifier con su propio worker.py, requirements.txt y Dockerfile. En worker.py se desarrolará la lógica para enviar las notificaciones.



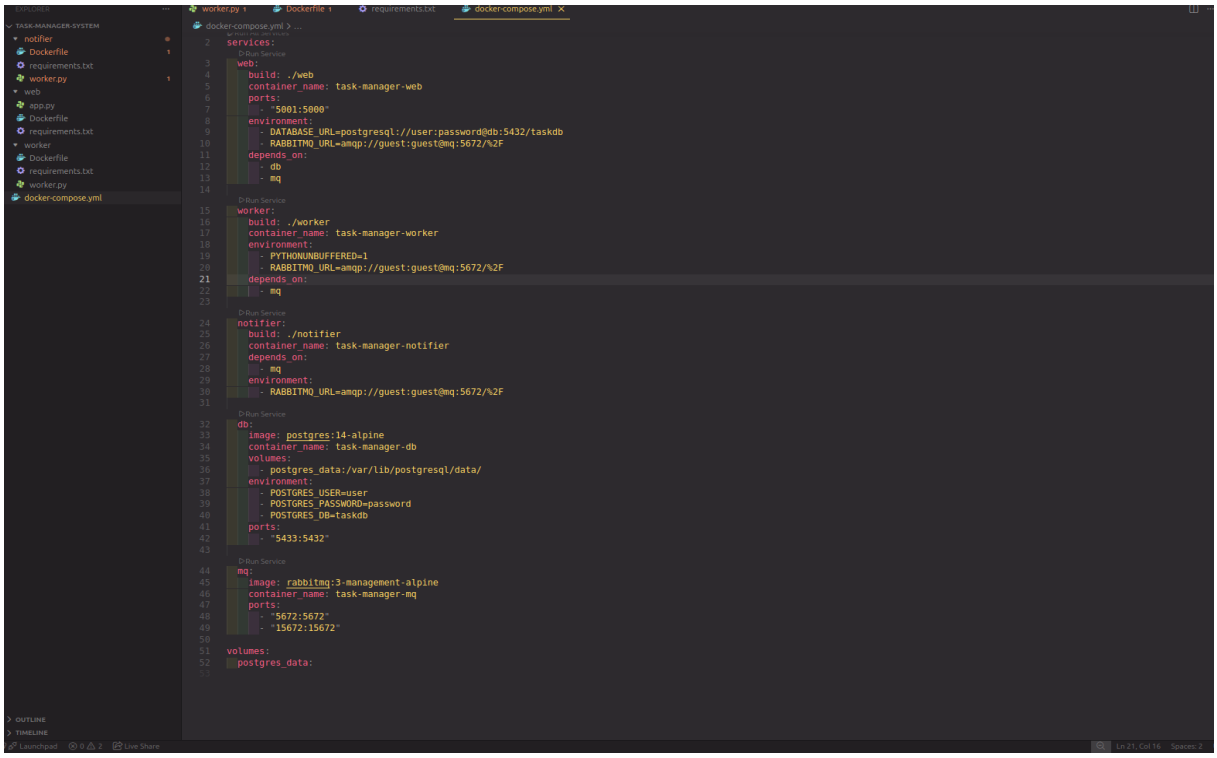
En requirements.txt se pondrán las dependencias de python.



Por último, el dockerfile será el encargado de crear la imagen del docker.

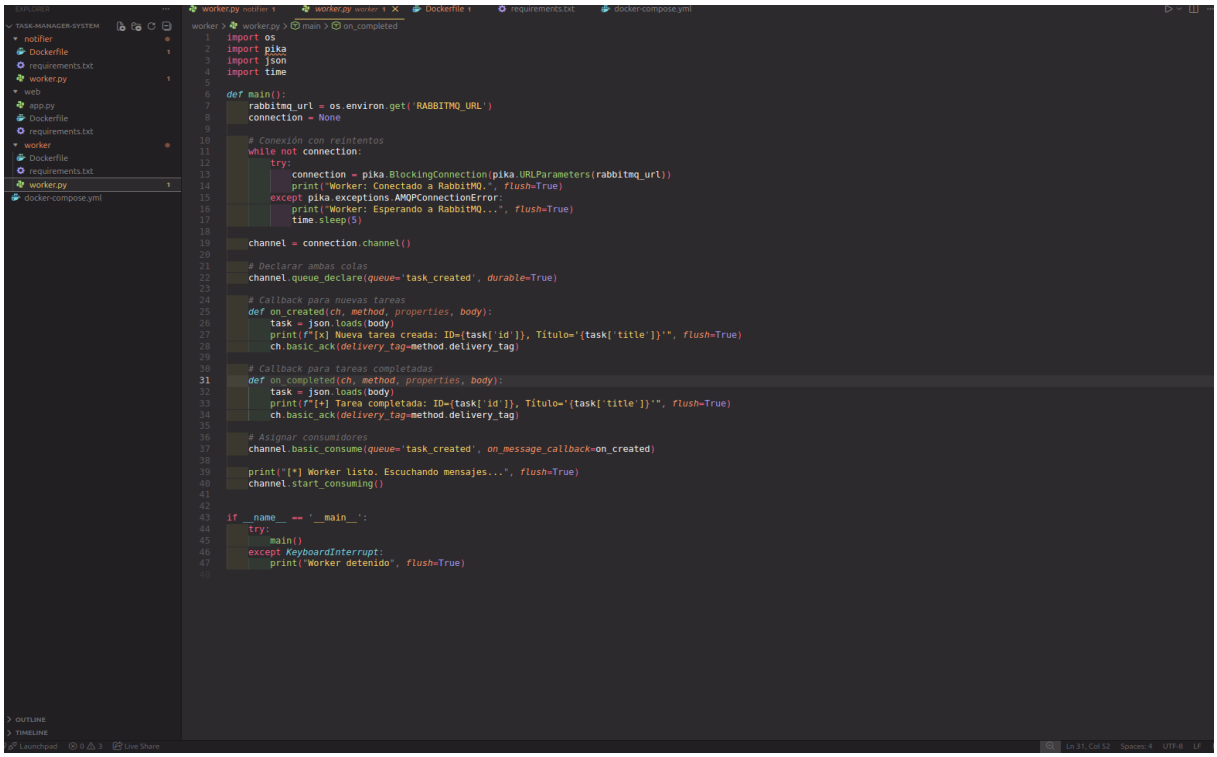


Una vez configurado el notifier es necesario añadir a docker-compose.yml el nuevo servicio.



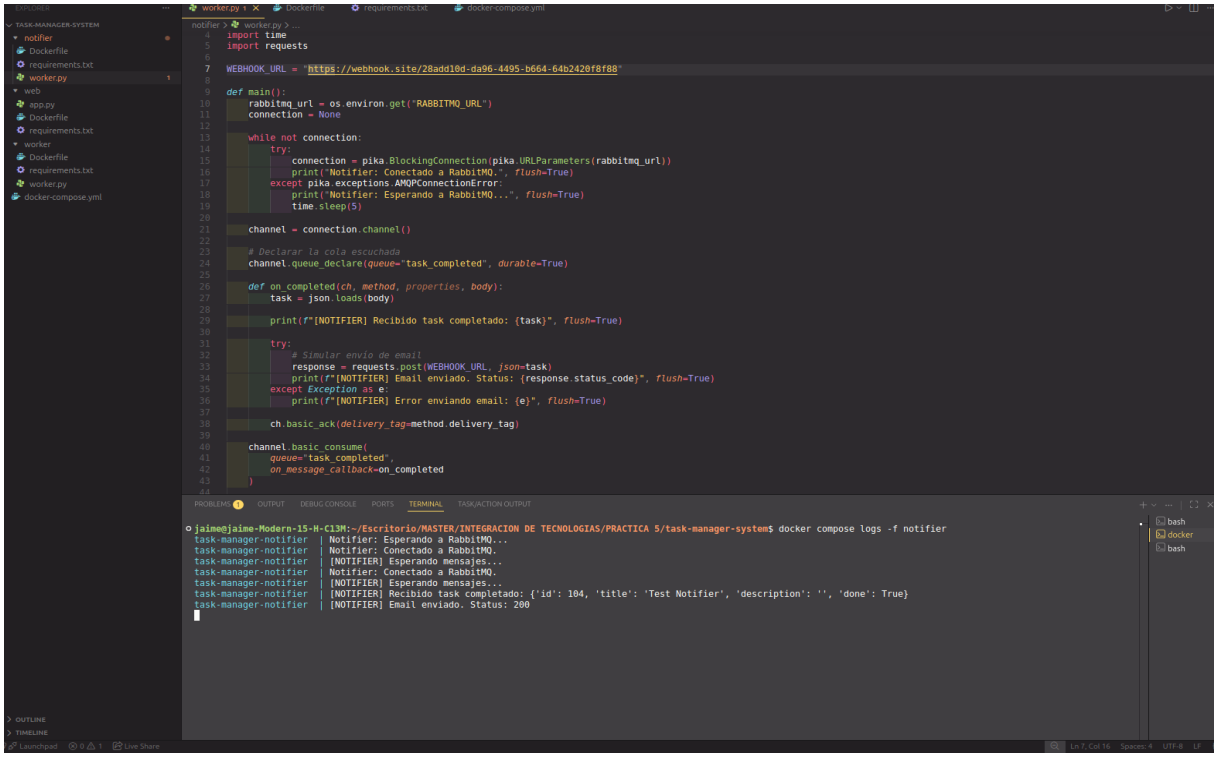
```yaml
services:
  web:
    build: ./web
    container_name: task-manager-web
    ports:
      - "5001:5000"
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/taskdb
      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
    depends_on:
      - db
      - mq

  worker:
    build: ./worker
    container_name: task-manager-worker
    environment:
      - PYTHONUNBUFFERED=1
      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
    depends_on:
      - mq

  notifier:
    build: ./notifier
    container_name: task-manager-notifier
    depends_on:
      - mq
    environment:
      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F

  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"

  mq:
    image: rabbitmq:3-management-alpine
    container_name: task-manager-mq
    ports:
      - "5672:5672"
      - "15672:15672"

volumes:
  postgres_data:
```

El siguiente paso es modificar el worker.py de worker para que no escuche la cola task-completed.



```python
import os
import pika
import json
import time

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

    # Conexión con reintentos
    while not connection:
        try:
            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
            print("Worker: Conectado a RabbitMQ.", flush=True)
        except pika.exceptions.AMQPConnectionError:
            print("Worker: Esperando a RabbitMQ...", flush=True)
            time.sleep(5)

    channel = connection.channel()

    # Declarar ambas colas
    channel.queue_declare(queue='task_created', durable=True)

    # Callback para nuevas tareas
    def on_created(ch, method, properties, body):
        task = json.loads(body)
        print(f"[x] Nueva tarea creada: ID={task['id']}, Título='{task['title']}'", flush=True)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    # Callback para tareas completadas
    def on_completed(ch, method, properties, body):
        task = json.loads(body)
        print(f"[+] Tarea completada: ID={task['id']}, Título='{task['title']}'", flush=True)
        ch.basic_ack(delivery_tag=method.delivery_tag)

    # Asignar consumidores
    channel.basic_consume(queue='task_created', on_message_callback=on_created)

    print('[*] Worker listo. Escuchando mensajes...', flush=True)
    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Worker detenido', flush=True)
```

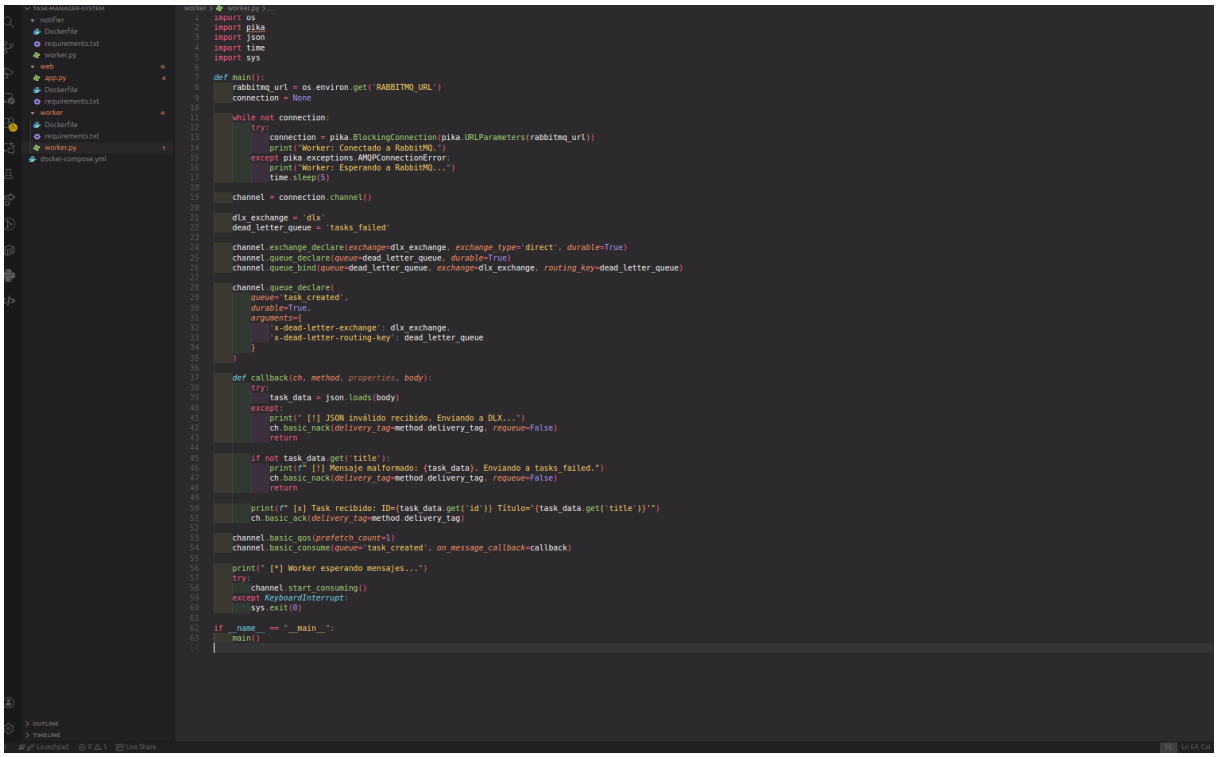
Una vez realizados todos estos cambios, podemos ver como notifier es el encargado de consumir la cola de mensajes.



Capítulo 4

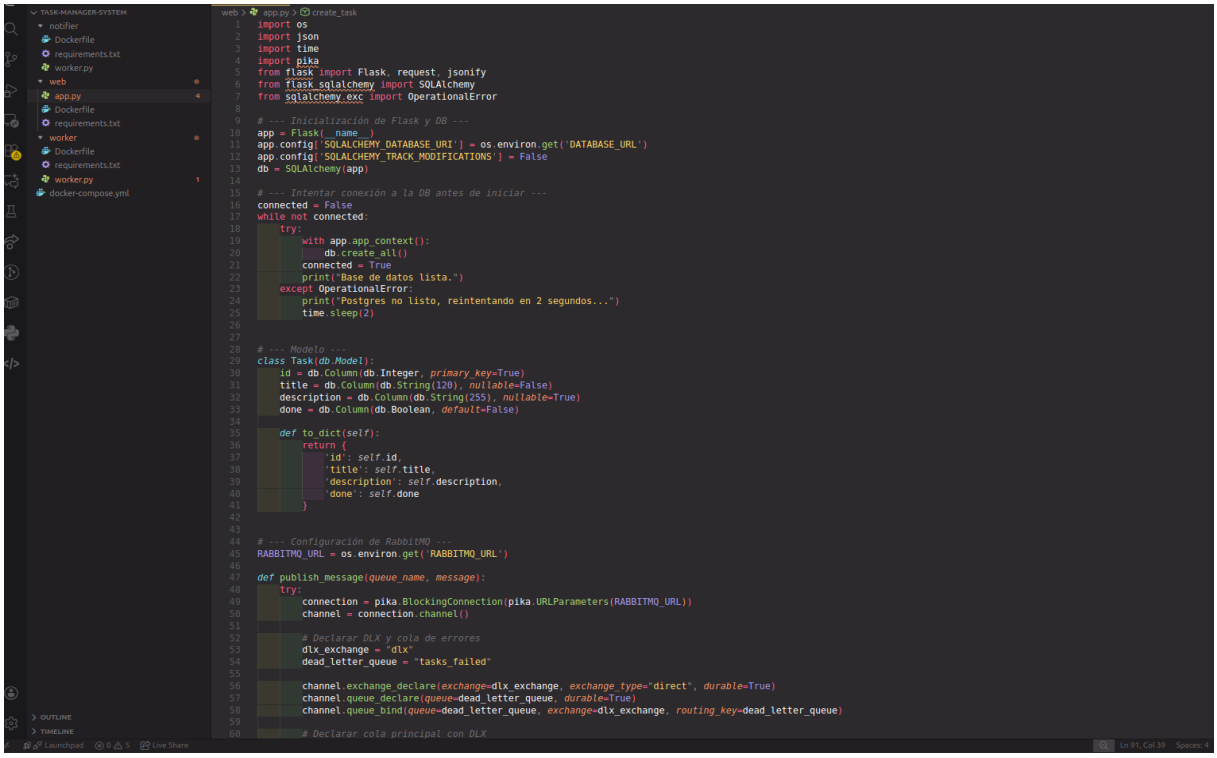
Ejercicio 3

El objetivo del ejercicio 3 es aumentar la resiliencia del sistema de mensajería. Para ello, es necesario modificar los archivos app.py y worker.py. El primer paso ha sido adaptar worker.py para que al recibir un mensaje sin título rechace el mensaje sin enconlarlo.

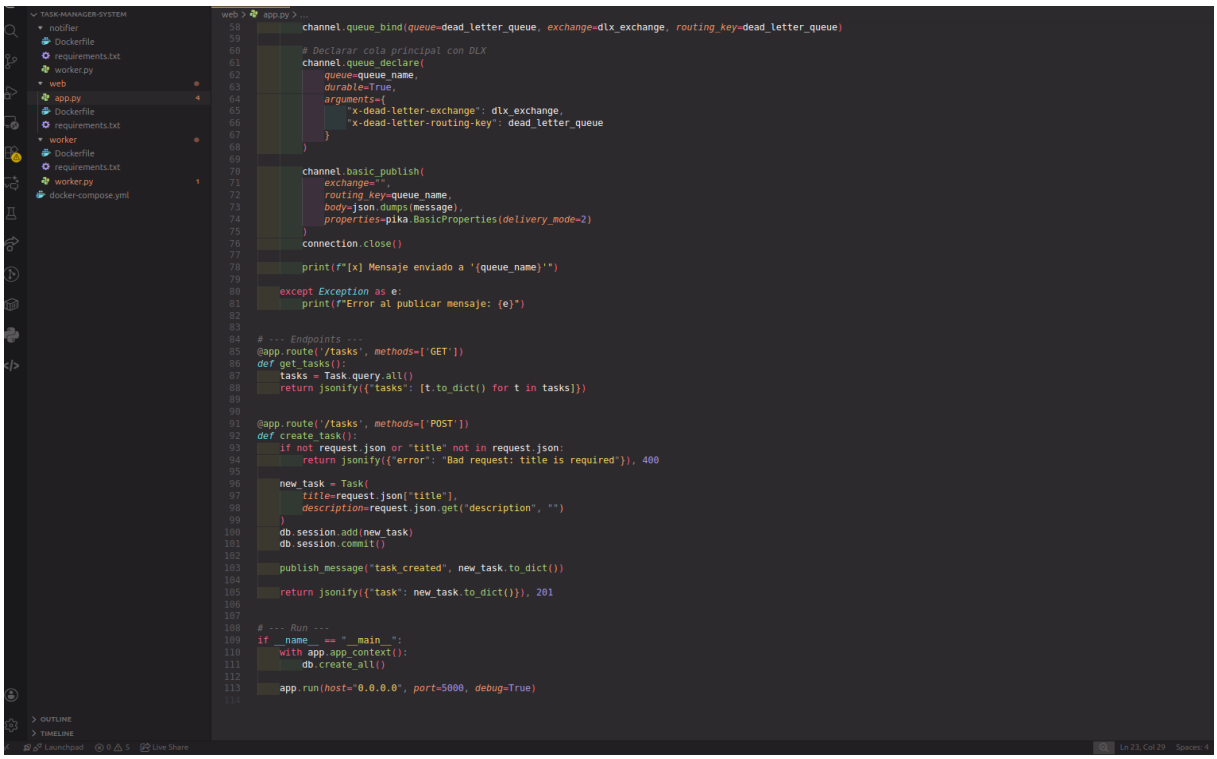


```
import os
import pika
import json
import time
import sys

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

    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()

    dlx_exchange = 'dlx'
    dead_letter_queue = 'tasks_failed'

    channel.exchange_declare(exchange=dlx_exchange, exchange_type='direct', durable=True)
    channel.queue_declare(queue=dead_letter_queue, durable=True)
    channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)

    channel.queue_declare(
        queue='task_created',
        durable=True,
        arguments={
            'x-dead-letter-exchange': dlx_exchange,
            'x-dead-letter-routing-key': dead_letter_queue
        }
    )

    def callback(ch, method, properties, body):
        try:
            task_data = json.loads(body)
        except:
            print(" [!] JSON inválido recibido. Enviando a DLX...")
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
            return

        if not task_data.get('title'):
            print(f" [!] Mensaje malformado: {task_data}. Enviando a tasks_failed.")
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
            return

        print(f" [x] Task recibido: ID={task_data.get('id')} Título='{task_data.get('title')}'")
        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(" [*] Worker esperando mensajes...")
    try:
        channel.start_consuming()
    except KeyboardInterrupt:
        sys.exit(0)

if __name__ == "__main__":
    main()
```

A continuación se ha modificado app.py para mandar cualquier mensaje rechazado por worker de la cola task-created a task-failed.

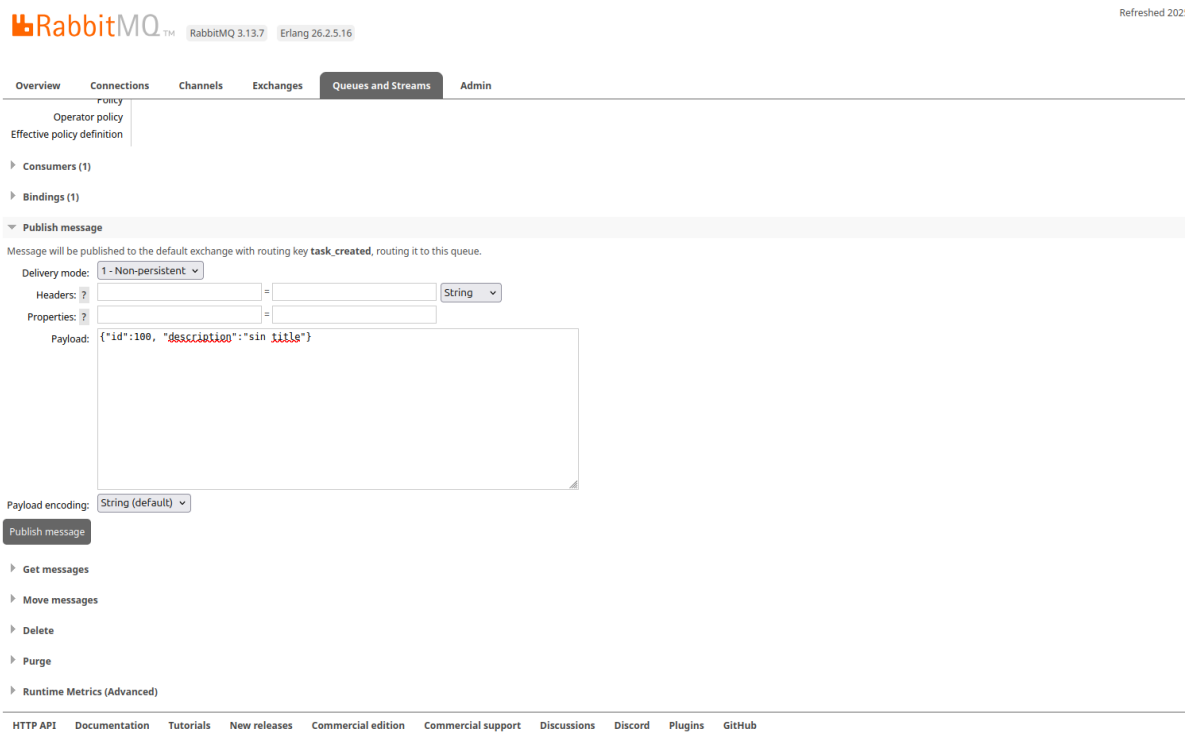


```python
import os
import json
import time
import pika
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.exc import OperationalError

# --- Inicialización de Flask y DB ---
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

# --- Intentar conexión a la DB antes de iniciar ---
connected = False
while not connected:
    try:
        with app.app_context():
            db.create_all()
        connected = True
        print("Base de datos lista.")
    except OperationalError:
        print("Postgres no listo, reintentando en 2 segundos...")
        time.sleep(2)


# --- Modelo ---
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()

        # Declarar DLX y cola de errores
        dlx_exchange = "dlx"
        dead_letter_queue = "tasks_failed"

        channel.exchange_declare(exchange=dlx_exchange, exchange_type="direct", durable=True)
        channel.queue_declare(queue=dead_letter_queue, durable=True)
        channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)

        # Declarar cola principal con DLX
```



```python
        channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)

        # Declarar cola principal con DLX
        channel.queue_declare(
            queue=queue_name,
            durable=True,
            arguments={
                "x-dead-letter-exchange": dlx_exchange,
                "x-dead-letter-routing-key": dead_letter_queue
            }
        )

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

        print(f"[x] Mensaje enviado a '{queue_name}'")

    except Exception as e:
        print(f"Error al publicar mensaje: {e}")


# --- Endpoints ---
@app.route('/tasks', methods=['GET'])
def get_tasks():
    tasks = Task.query.all()
    return jsonify({"tasks": [t.to_dict() for t in tasks]})


@app.route('/tasks', methods=['POST'])
def create_task():
    if not request.json or "title" not 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()

    publish_message("task_created", new_task.to_dict())

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


# --- Run ---
if __name__ == "__main__":
    with app.app_context():
        db.create_all()

    app.run(host="0.0.0.0", port=5000, debug=True)
```

Para comprobar que el funcionamiento es correcto, desde Rabbit introducirems una tarea sin título.



Al no tener título la tarea se deb guardar en la cola task-failed, como vemos a continuación es lo esperado por lo que el funcionamiento sería correcto.