

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

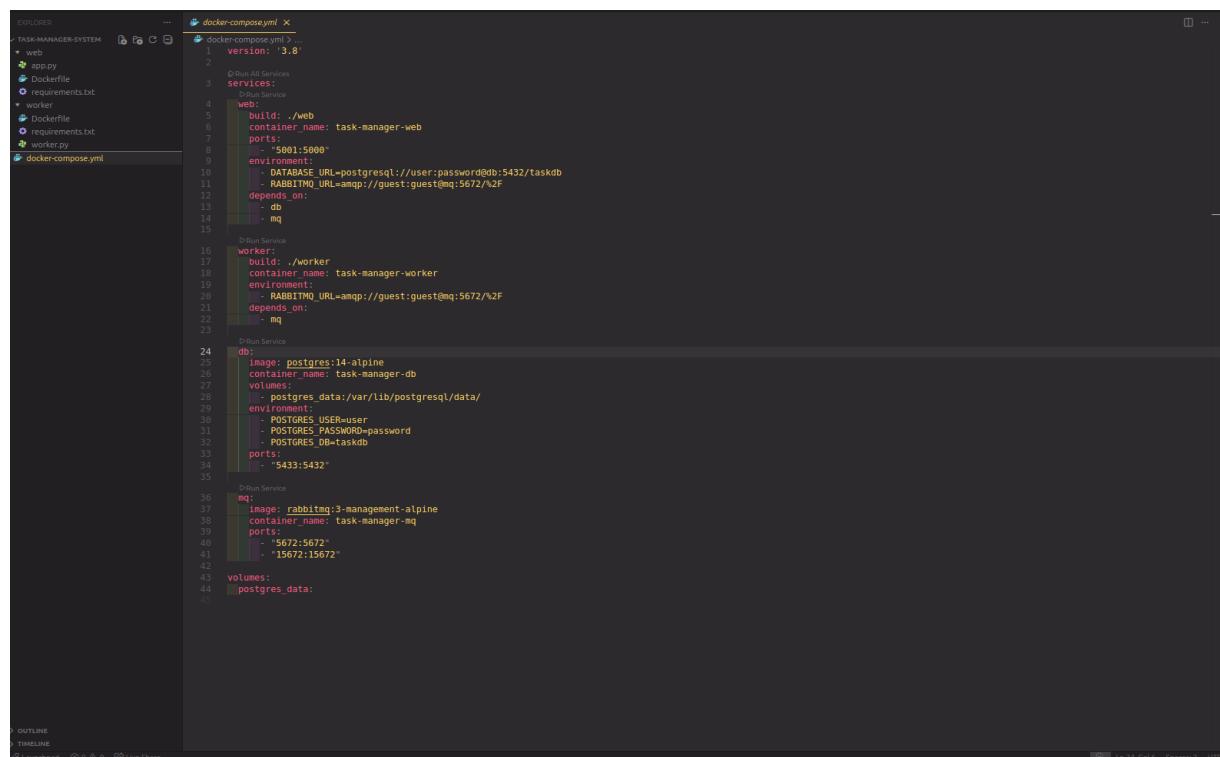
Índice general

1. Ejercicio Guiado 1	2
2. Ejercicio 1	8
3. Ejercicio 2	10
4. Ejercicio 3	14

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 raíz de nuestro proyecto. Este fichero será el centro de control de nuestra aplicación.



```
version: '3.8'
services:
  web:
    build: ./web
    container_name: task-manager-web
    ports:
      - "5001:5000"
    environment:
      - DATABASE_URL=postgres://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:
      - RABBITMQ_URL=amqp://guest:guest@mq:5672/%2F
    depends_on:
      - mq
  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:
```

Una vez creado el fichero, el siguiente paso es crear una 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.

```
...  app.py 1 x
web  app.py 4 tasks.py 5 Task

  # ... Configuración de la Base de Datos ...
10 app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL')
11 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
12 db = SQLAlchemy(app)
13
14 # ... Modelos de la Base de Datos ...
15 class Task(db.Model):
16     id = db.Column(db.Integer, primary_key=True)
17     title = db.Column(db.String(120), nullable=False)
18     description = db.Column(db.String(255), nullable=True)
19     done = db.Column(db.Boolean, default=False)
20
21     def to_dict(self):
22         return {
23             'id': self.id,
24             'title': self.title,
25             'description': self.description,
26             'done': self.done
27         }
28
29 # ... Configuración de RabbitMQ ...
30 RABBITMQ_URL = os.environ.get('RABBITMQ_URL')
31
32 def publish_message(queue_name, message):
33     try:
34         connection = pika.BlockingConnection(pika.URLParameters(RABBITMQ_URL))
35         channel = connection.channel()
36         channel.queue_declare(queue=queue_name, durable=True)
37         channel.basic_publish(
38             exchange='',
39             routing_key=queue_name,
40             body=message,
41             properties=pika.BasicProperties(delivery_mode=2) # persistente
42         )
43     except Exception as e:
44         print(f'Error publishing message: {e}')
45
46 # ... Endpoints de la API ...
47 @app.route('/tasks', methods=['GET'])
48 def get_tasks():
49     tasks = Task.query.all()
50     return jsonify([task.to_dict() for task in tasks])
51
52 @app.route('/tasks', methods=['POST'])
53 def create_task():
54     if not request.json or 'title' not in request.json:
55         return jsonify({'error': 'Bad request: title is required'}), 400
56
57     new_task = Task(
58         title=request.json['title'],
59         description=request.json.get('description', ''),
60     )
61
62     db.session.add(new_task)
63     db.session.commit()
64
65     # Publicar mensaje en RabbitMQ
66     publish_message('task_created', new_task.to_dict())
67
68     return jsonify({'task': new_task.to_dict()}), 201
69
70 # ... puedes añadir más endpoints como GET /tasks/<id>, DELETE, PUT, etc.
71
72 if __name__ == '__main__':
73     with app.app_context():
74         db.create_all()
75     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.

The screenshot shows the Dockerfile editor in Visual Studio Code. The left sidebar displays the project structure:

- EXPLORER
- TASK-MANAGER-SYSTEM
- web
- app
- Dockerfile
- requirements.txt
- worker
- Dockerfile
- requirements.txt
- worker.py
- docker-compose.yml

The right pane shows the Dockerfile content:

```
FROM python:3.9-slim-buster (last pushed 2 years ago)
1 1 FROM python:3.9-slim-buster
2 2 WORKDIR /app
3 3 COPY requirements.txt .
4 4 COPY requirements.txt .
5 5 RUN pip install --no-cache-dir -r requirements.txt
6 6 COPY app .
7 7 COPY app .
8 8 COPY .
9 9 COPY .
10 10 # Copy app code
11 11 # Expose port and start with gunicorn (or flask run for dev)
12 12 EXPOSE 5000
13 13
14 14 CMD ["python", "app.py"]
```

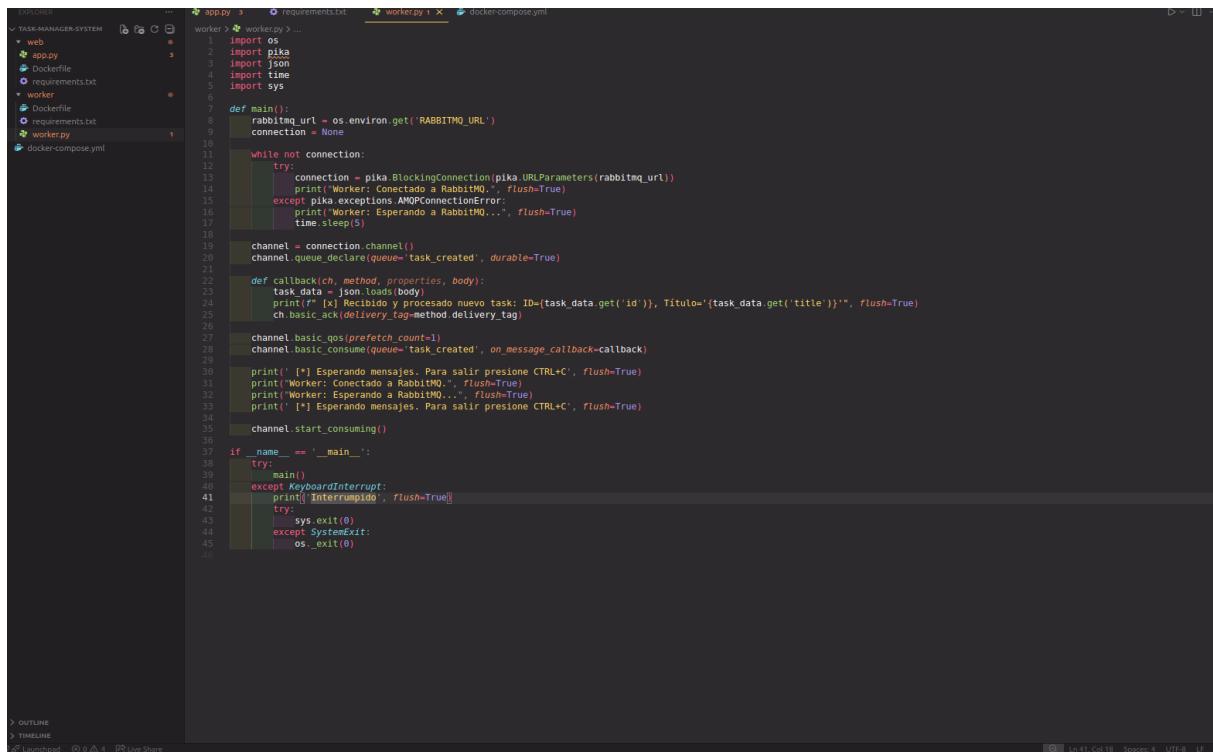
Por último, creamos el archivo requirements.txt donde se definirán las dependencias de Python que se instalarán en el contenedor.



The screenshot shows a code editor interface with a dark theme. On the left, there is a tree view of a project structure named 'TASK-MANAGER-SYSTEM'. Inside 'TASK-MANAGER-SYSTEM', there are several sub-directories like 'web', 'worker', and 'Dockerfile'. In the 'web' directory, there is a file named 'requirements.txt'. The content of this file is:

```
Flask==1.1.1
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 raíz 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.



The screenshot shows a code editor interface with a dark theme. On the left, there is a tree view of a project structure named 'TASK-MANAGER-SYSTEM'. Inside 'TASK-MANAGER-SYSTEM', there are several sub-directories like 'web', 'worker', and 'Dockerfile'. In the 'worker' directory, there is a file named 'worker.py'. The content of this file is:

```
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"[*] Recibido y procesado nuevo task: ID={task_data.get('id')}, Titulo='{task_data.get('title')}'")
        ch.basic_ack(delivery_tag=method.delivery_tag)

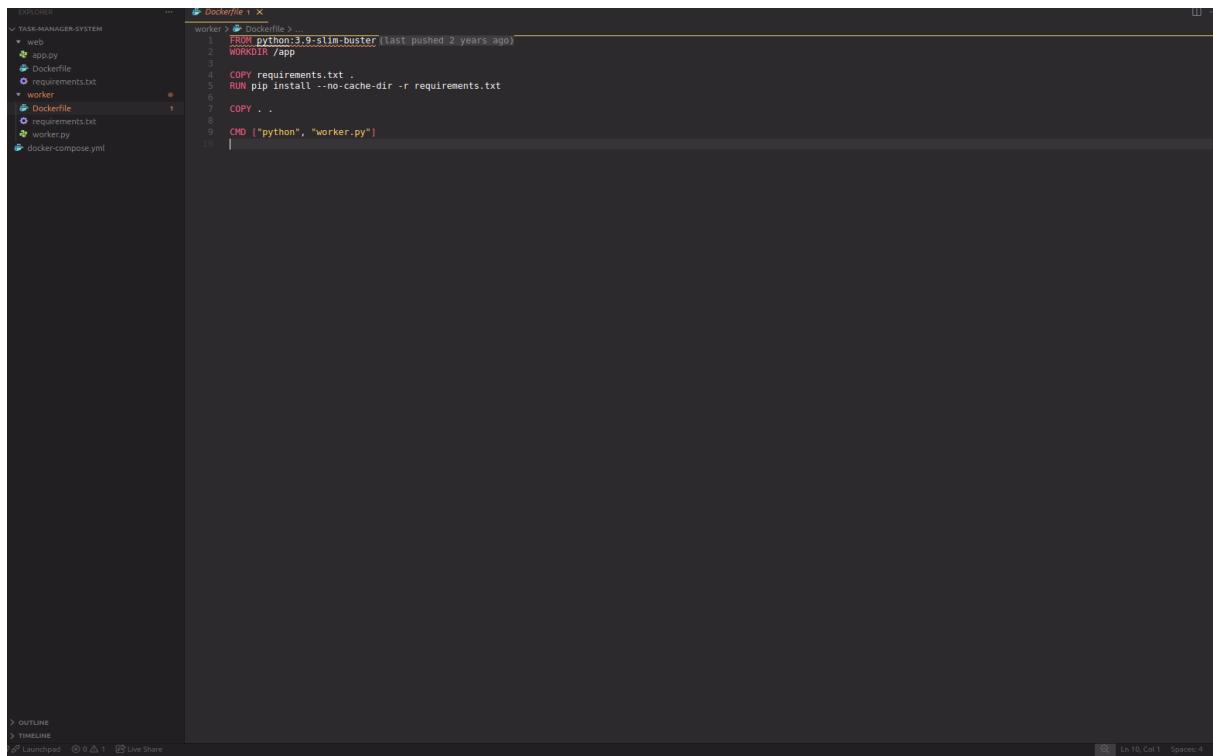
    channel.basic_qos(prefetch_count=1)
    channel.basic_consume(queue='task_created', on_message_callback=callback)

    print("[*] Esperando mensajes. Para salir presione CTRL+C")
    print("Worker: Conectado a RabbitMQ")
    print("Worker: Esperando a RabbitMQ")
    print("[*] Esperando mensajes. Para salir presione CTRL+C")

    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.

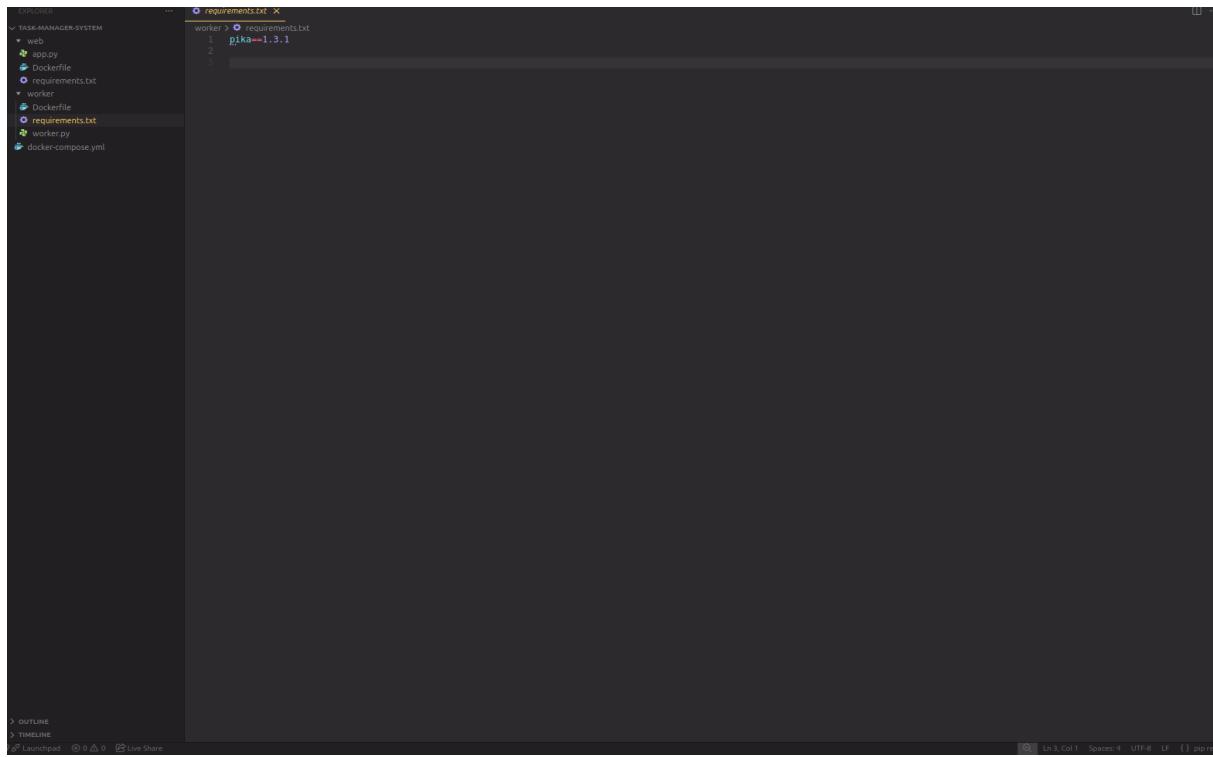


The screenshot shows the VS Code interface with the 'worker' folder selected in the left sidebar. The right pane displays the Dockerfile content:

```
FROM python:3.9-slim-buster (last pushed 2 years ago)
WORKDIR app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY .
CMD ["python", "worker.py"]
```

Below the code editor, the status bar indicates 'Ln 10, Col 1' and 'Spaces: 4'.

Y por último, configuraremos requirements.txt para poner las dependencias.

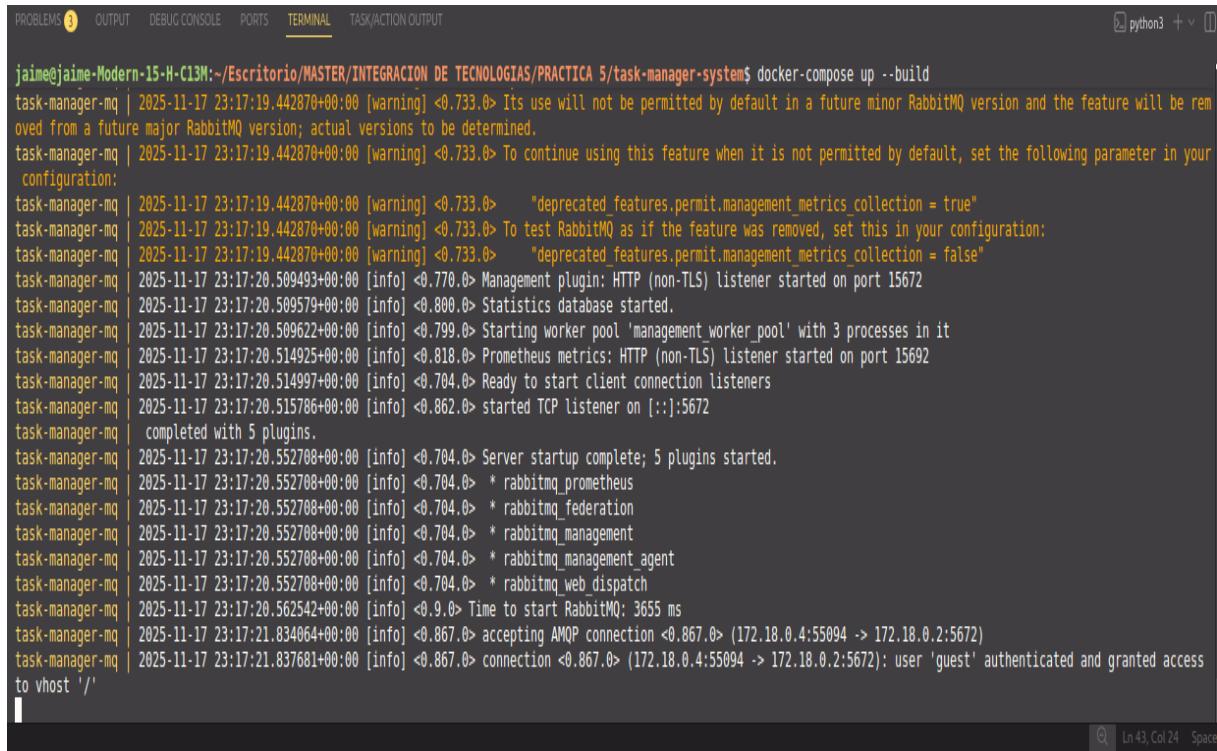


The screenshot shows the VS Code interface with the 'worker' folder selected in the left sidebar. The right pane displays the requirements.txt content:

```
pika==1.3.1
```

Below the code editor, the status bar indicates 'Ln 3, Col 1' and 'Spaces: 4'.

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.

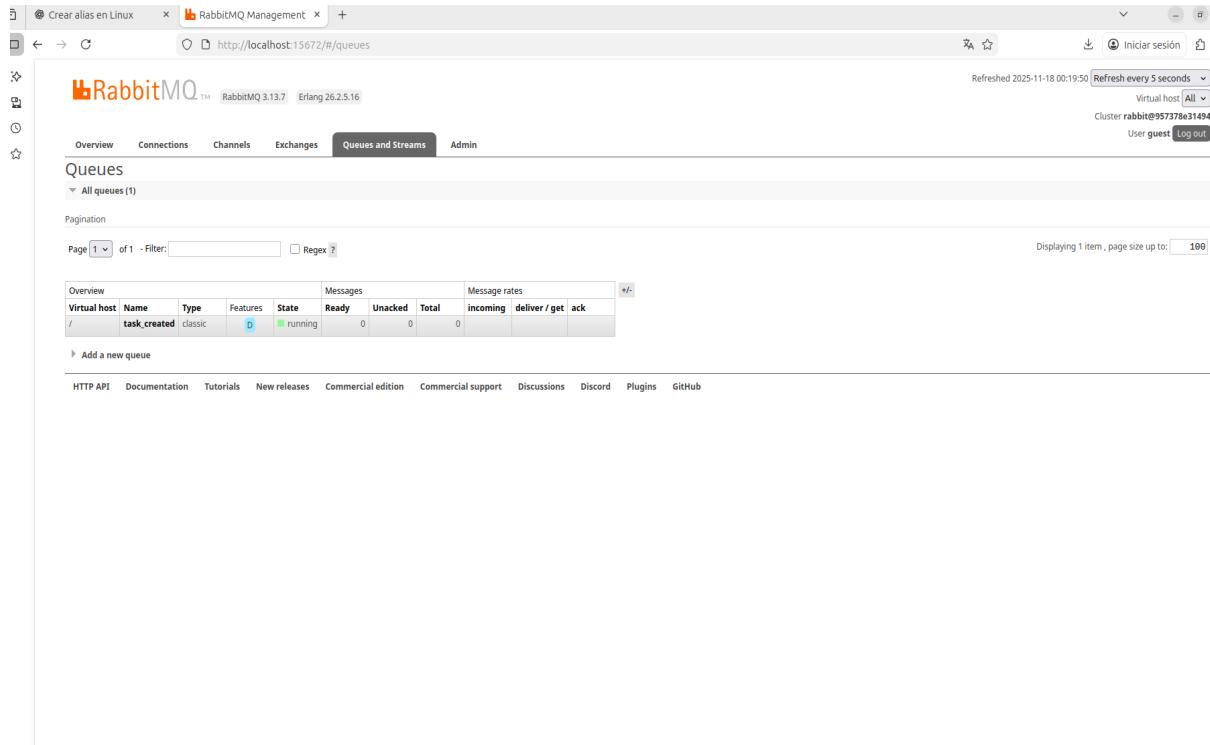


```

jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker-compose up --build
task-manager-mq | 2025-11-17 23:17:19.442870+00:00 [warning] <0.733.0> Its use will not be permitted by default in a future minor RabbitMQ version and the feature will be removed from a future major RabbitMQ version; actual versions to be determined.
task-manager-mq | 2025-11-17 23:17:19.442870+00:00 [warning] <0.733.0> To continue using this feature when it is not permitted by default, set the following parameter in your configuration:
task-manager-mq | 2025-11-17 23:17:19.442870+00:00 [warning] <0.733.0>     "deprecated_features.permit.management_metrics_collection = true"
task-manager-mq | 2025-11-17 23:17:19.442870+00:00 [warning] <0.733.0> To test RabbitMQ as if the feature was removed, set this in your configuration:
task-manager-mq | 2025-11-17 23:17:19.442870+00:00 [warning] <0.733.0>     "deprecated_features.permit.management_metrics_collection = false"
task-manager-mq | 2025-11-17 23:17:20.509493+00:00 [info] <0.770.0> Management plugin: HTTP (non-TLS) listener started on port 15672
task-manager-mq | 2025-11-17 23:17:20.509579+00:00 [info] <0.800.0> Statistics database started.
task-manager-mq | 2025-11-17 23:17:20.509622+00:00 [info] <0.799.0> Starting worker pool 'management_worker_pool' with 3 processes in it
task-manager-mq | 2025-11-17 23:17:20.514925+00:00 [info] <0.818.0> Prometheus metrics: HTTP (non-TLS) listener started on port 15692
task-manager-mq | 2025-11-17 23:17:20.514997+00:00 [info] <0.704.0> Ready to start client connection listeners
task-manager-mq | 2025-11-17 23:17:20.515786+00:00 [info] <0.862.0> started TCP listener on [::]:5672
task-manager-mq | completed with 5 plugins.
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> Server startup complete; 5 plugins started.
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> * rabbitmq_prometheus
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> * rabbitmq_federation
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> * rabbitmq_management
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> * rabbitmq_management_agent
task-manager-mq | 2025-11-17 23:17:20.552708+00:00 [info] <0.704.0> * rabbitmq_web_dispatch
task-manager-mq | 2025-11-17 23:17:20.562542+00:00 [info] <0.9.0> Time to start RabbitMQ: 3655 ms
task-manager-mq | 2025-11-17 23:17:21.834064+00:00 [info] <0.867.0> accepting AMQP connection <0.867.0> (172.18.0.4:55094 -> 172.18.0.2:5672)
task-manager-mq | 2025-11-17 23:17:21.837681+00:00 [info] <0.867.0> connection <0.867.0> (172.18.0.4:55094 -> 172.18.0.2:5672): user 'guest' authenticated and granted access to vhost '/'

```

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



Overview		Messages			Message rates					
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	task_created	classic	D	running	0	0	0			

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.

```

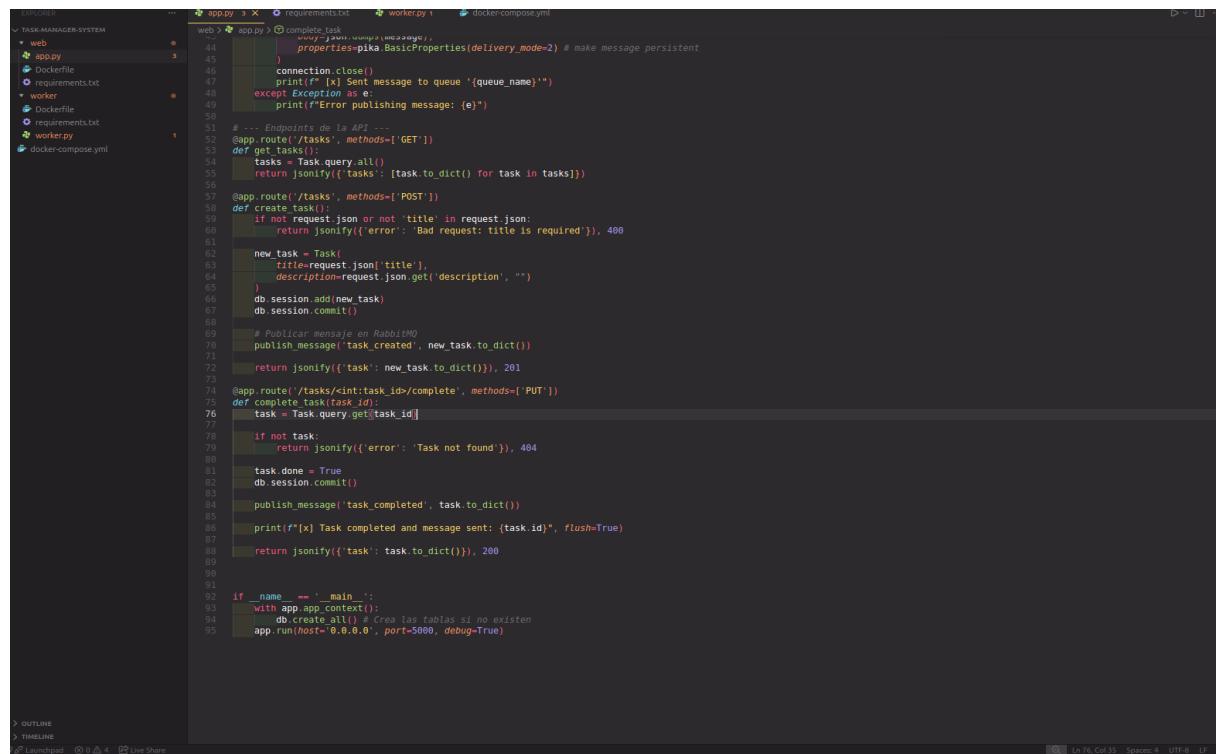
    ...
    app.py | requirements.txt | worker.py | docker-compose.yml
    ...
    worker > worker.py > main
    1 import os
    2 import pika
    3 import json
    4 import time
    5 import sys
    6
    7 def main():
    8     rabbitmq_url = os.environ.get('RABBITMQ_URL')
    9     connection = None
   10
   11     while not connection:
   12         try:
   13             connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
   14             print("Worker: Conectado a RabbitMQ.", flush=True)
   15         except pika.exceptions.AMQPConnectionError:
   16             print("Worker: Esperando a RabbitMQ...", flush=True)
   17             time.sleep(5)
   18
   19     channel = connection.channel()
   20     channel.queue_declare(queue='task_created', durable=True)
   21
   22     def callback(ch, method, properties, body):
   23         task_data = json.loads(body)
   24         print(f"[x] Recibido y procesado nuevo task: ID={task_data.get('id')}, Titulo='{task_data.get('title')}'")
   25         ch.basic_ack(delivery_tag=method.delivery_tag)
   26
   27     channel.basic_qos(prefetch_count=1)
   28     channel.basic_consume(queue='task_created', on_message_callback=callback)
   29
  PROBLEMS 1 OUTPUT DEBUG CONSOLE PORTS TERMINAL TASK ACTION OUTPUT
  ● jaime@jaime-Morden-15-N-CL3M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker compose logs worker
  task-manager-worker | Worker: Conectado a RabbitMQ.
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Conectado a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Esperando a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Esperando a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Conectado a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Conectado a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Esperando a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Esperando a RabbitMQ...
  task-manager-worker | [*] Esperando mensajes. Para salir presione CTRL+C
  task-manager-worker | Worker: Recibido y procesado nuevo task: ID=36, titulo='Prueba Worker'
  ● jaime@jaime-Morden-15-N-CL3M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker compose logs web
  task-manager-web | * Serving Flask app "app"
  task-manager-web | * Debug mode: on
  task-manager-web | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
  task-manager-web | * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
  task-manager-web | * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
  task-manager-web | * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
  task-manager-web | * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
  task-manager-web | * Debugger PIN: 349-410-905
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 00:50:06] "POST /tasks HTTP/1.1" 201 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 00:50:10] "GET /tasks HTTP/1.1" 200 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 00:50:10] "GET /tasks HTTP/1.1" 404 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 00:50:11] "GET /tasks HTTP/1.1" 200 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 00:50:11] "GET /favicon.ico HTTP/1.1" 404 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 01:08:12] "GET /tasks HTTP/1.1" 200 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 01:08:12] "GET /favicon.ico HTTP/1.1" 404 -
  task-manager-web | 172.18.0.1 - - [18/Nov/2025 01:08:20] "POST /tasks HTTP/1.1" 201 -
  
```

The screenshot shows a code editor with a Python file named 'worker.py' open. The code implements a RabbitMQ consumer that listens for messages on a queue named 'task_created'. When a message is received, it prints the task details (ID and title) and then uses basic acknowledgment to reply to the message. Below the code editor is a terminal window showing the logs of two Docker containers: 'worker' and 'web'. The 'worker' container logs show the worker connecting to RabbitMQ and waiting for messages. The 'web' container logs show the Flask application running on port 5000, receiving a POST request to '/tasks' (which creates a task), and several GET requests for tasks and the favicon. The terminal also shows a warning about using the development server in production.

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.



```
app.py 3 X requirements.txt 4 Docker-compose.yml
web > app.py > complete_task
43     connection = pika.BlockingConnection(pika.ConnectionParameters(queue_name))
44     properties = pika.BasicProperties(delivery_mode=2) # make message persistent
45
46     connection.close()
47     print(f" [x] Sent message to queue '{queue_name}'")
48 except Exception as e:
49     print(f"Error publishing message: {e}")
50
51 # --- Endpoints de la API ---
52 @app.route('/tasks', methods=['GET'])
53 def get_tasks():
54     tasks = Task.query.all()
55     return jsonify([{'task': task.to_dict() for task in tasks]})
56
57 @app.route('/tasks', methods=['POST'])
58 def create_task():
59     if not request.json or not 'title' in request.json:
60         return jsonify({'error': 'Bad request: title is required'}), 400
61
62     new_task = Task(
63         title=request.json['title'],
64         description=request.json.get('description', ""))
65
66     db.session.add(new_task)
67     db.session.commit()
68
69     # Publicar mensaje en RabbitMQ
70     publish_message("task_created", new_task.to_dict())
71
72     return jsonify({'task': new_task.to_dict()}), 201
73
74 @app.route('/tasks/<int:task_id>/complete', methods=['PUT'])
75 def complete_task(task_id):
76     task = Task.query.get(task_id)
77
78     if not task:
79         return jsonify({'error': 'Task not found'}), 404
80
81     task.done = True
82     db.session.commit()
83
84     publish_message("task_completed", task.to_dict())
85
86     print(f"[x] Task completed and message sent: {task.id}", flush=True)
87
88     return jsonify({'task': task.to_dict()}), 200
89
90
91 if __name__ == '__main__':
92     with app.app_context():
93         db.create_all() # Crea las tablas si no existen
94     app.run(host='0.0.0.0', port=5000, debug=True)
```

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

```

1 app.py 3 requirements.txt 4 worker.py t 5 docker-compose.yml
worker > worker.py main > on_created
1 import os
2 import pika
3 import json
4 import time
5
6 def main():
7     rabbitmq_url = os.environ.get('RABBITMQ_URL')
8     connection = None
9
10    # Conexión con reintentos
11    while not connection:
12        try:
13            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
14            print("Worker: Conectado a RabbitMQ...", flush=True)
15        except pika.exceptions.AMQPConnectionError:
16            print("Worker: Esperando a RabbitMQ...", flush=True)
17            time.sleep(5)
18
19    channel = connection.channel()
20
21    # Declarar ambas colas
22    channel.queue_declare(queue='task_created', durable=True)
23    channel.queue_declare(queue='task_completed', durable=True)
24
25    # Callback para nuevas tareas
26    def on_created(ch, method, properties, body):
27        task = json.loads(body)
28        print(f"[x] Nueva tarea creada: ID={task['id']}, Título={task['title']}", flush=True)
29        ch.basic_ack(delivery_tag=method.delivery_tag)
30
31    # Callback para tareas completadas
32    def on_completed(ch, method, properties, body):
33        task = json.loads(body)
34        print(f"[+] Tarea completada: ID={task['id']}, Título={task['title']}", flush=True)
35        ch.basic_ack(delivery_tag=method.delivery_tag)
36
37    # Asignar consumidores
38    channel.basic_consume(queue='task_created', on_message_callback=on_created)
39    channel.basic_consume(queue='task_completed', on_message_callback=on_completed)
40
41    print("[*] Worker listo. Escuchando mensajes...", flush=True)
42    channel.start_consuming()
43
44
45 if __name__ == '__main__':
46     try:
47         main()
48     except KeyboardInterrupt:
49         print("Worker detenido", flush=True)
50

```

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

```

1 app.py 3 requirements.txt 4 worker.py t 5 docker-compose.yml
worker > worker.py ...
1 import os
2 import pika
3 import json
4 import time
5
6 def main():
7     rabbitmq_url = os.environ.get('RABBITMQ_URL')
8     connection = None
9
10    # Conexión con reintentos
11    while not connection:
12        try:
13            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
14            print("Worker: Conectado a RabbitMQ...", flush=True)
15        except pika.exceptions.AMQPConnectionError:
16            print("Worker: Esperando a RabbitMQ...", flush=True)
17            time.sleep(5)
18
19    channel = connection.channel()
20
21    # Declarar ambas colas
22    channel.queue_declare(queue='task_created', durable=True)
23    channel.queue_declare(queue='task_completed', durable=True)
24
25    # Callback para nuevas tareas
26    def on_created(ch, method, properties, body):
27        task = json.loads(body)
28        print(f"[x] Nueva tarea creada: ID={task['id']}, Título={task['title']}", flush=True)
29        ch.basic_ack(delivery_tag=method.delivery_tag)
30
31    # Callback para tareas completadas
32    def on_completed(ch, method, properties, body):
33        task = json.loads(body)
34        print(f"[+] Tarea completada: ID={task['id']}, Título={task['title']}", flush=True)
35        ch.basic_ack(delivery_tag=method.delivery_tag)
36
37    # Asignar consumidores
38    channel.basic_consume(queue='task_created', on_message_callback=on_created)
39    channel.basic_consume(queue='task_completed', on_message_callback=on_completed)
40
41    print("[*] Worker listo. Escuchando mensajes...", flush=True)
42    channel.start_consuming()
43
44
45 if __name__ == '__main__':
46     try:
47         main()
48     except KeyboardInterrupt:
49         print("Worker detenido", flush=True)
50

```

jmajome@jmajome-OptiPlex-5090:~/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...

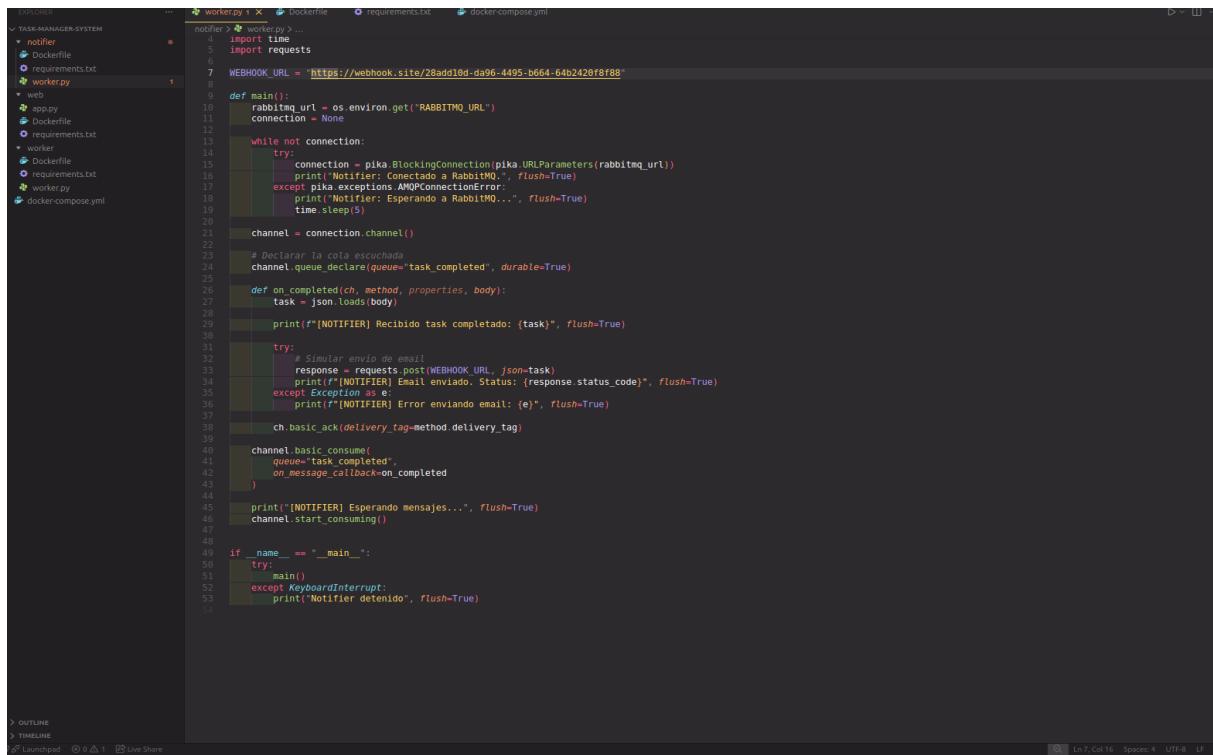
task-manager-worker | [x] Nueva tarea creada: ID=69, Título='Prueba final.'

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 desarrollará la lógica para enviar las notificaciones.



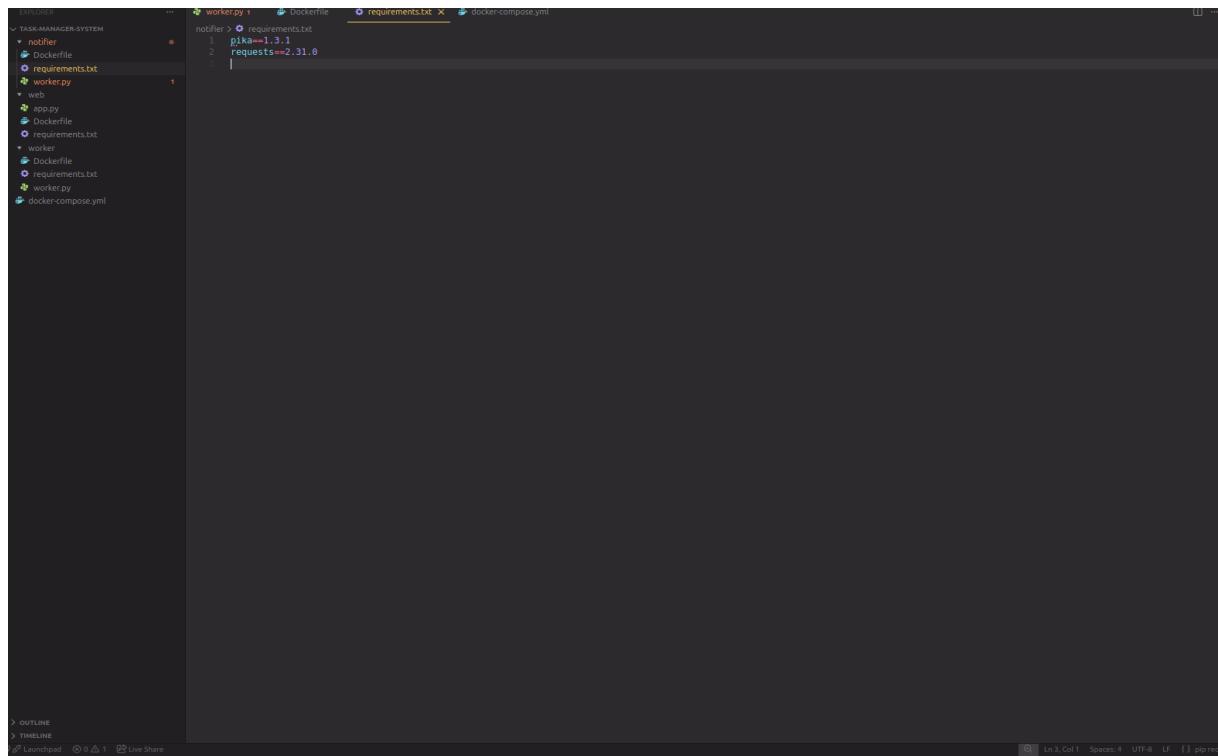
The screenshot shows a code editor with the following file structure:

- TASKMANAGER-SYSTEM
- notifier
- Dockerfile
- requirements.txt
- worker.py
- web
- Dockerfile
- requirements.txt
- worker.py
- docker-compose.yml

The worker.py file contains the following Python code:

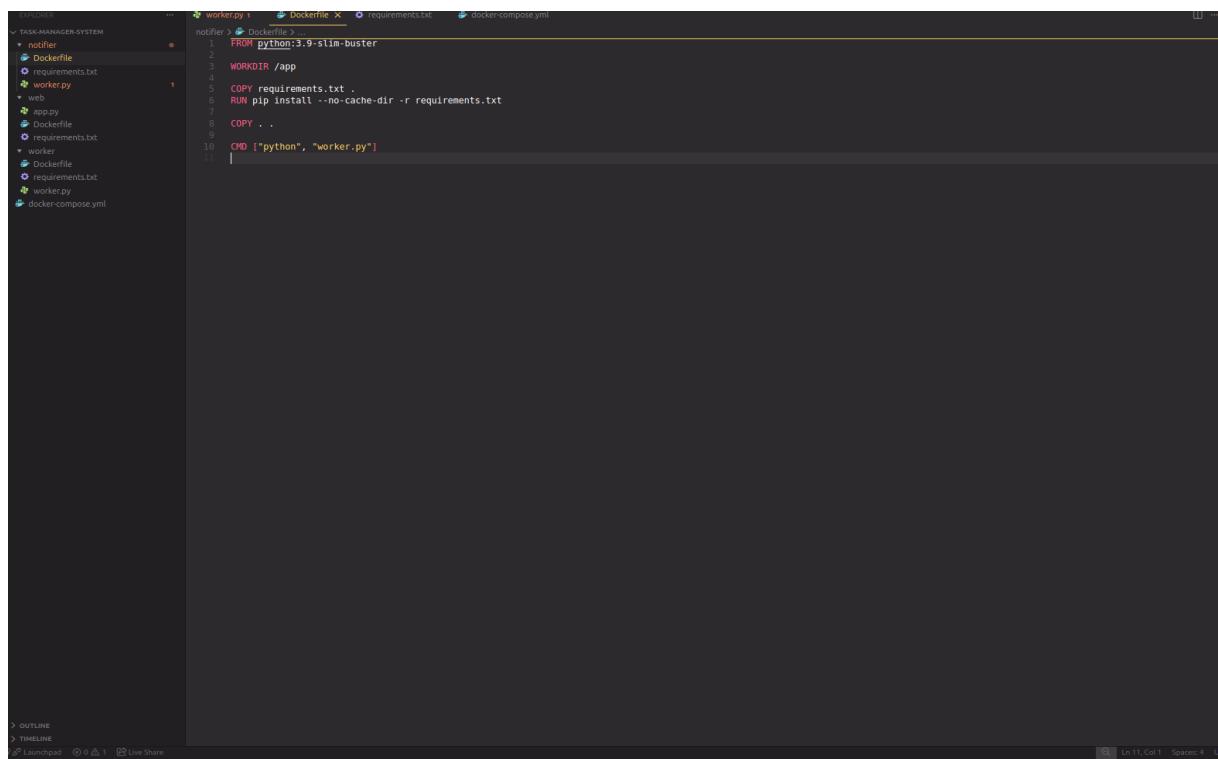
```
notifier > worker.py ...
import time
import requests
WEBHOOK_URL = 'https://webhook.site/28add10d-da06-4495-b664-64b2420f8f88'
def main():
    rabbitmq_url = os.environ.get("RABBITMQ_URL")
    connection = None
    while not connection:
        try:
            connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
        except pika.exceptions.AMQPConnectionError:
            print("[NOTIFIER] Conectado a RabbitMQ...", flush=True)
            time.sleep(5)
    channel = connection.channel()
    # Declarar la cola escuchada
    channel.queue_declare(queue="task_completed", durable=True)
    def on_completed(ch, method, properties, body):
        task = json.loads(body)
        print("[NOTIFIER] Recibido task completado: {task}", flush=True)
        try:
            response = requests.post(WEBHOOK_URL, json=task)
            print("[NOTIFIER] Email enviado. Status: {response.status_code}", flush=True)
        except Exception as e:
            print("[NOTIFIER] Error enviando email: {e}", flush=True)
        ch.basic_ack(delivery_tag=method.delivery_tag)
    channel.basic_consume(
        queue="task_completed",
        on_message_callback=on_completed
    )
    print("[NOTIFIER] Esperando mensajes...", flush=True)
    channel.start_consuming()
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("[Notifier detenido]", flush=True)
```

En requirements.txt se pondrán las dependencias de python.



```
pika==1.3.1
requests==2.31.0
```

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



```
FROM python:3.9-slim-buster
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "worker.py"]
```

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

```

version: '3'
services:
  web:
    build: ./web
    container_name: task-manager-web
    ports:
      - "5001:5000"
    environment:
      - POSTGRES_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:
      - db
      - 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.

```

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))
        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):
        print(f"[*] Nueva tarea creada: ID={task['id']}, Titulo='{task['title']}'")
        ch.basic_ack(delivery_tag=method.delivery_tag)

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

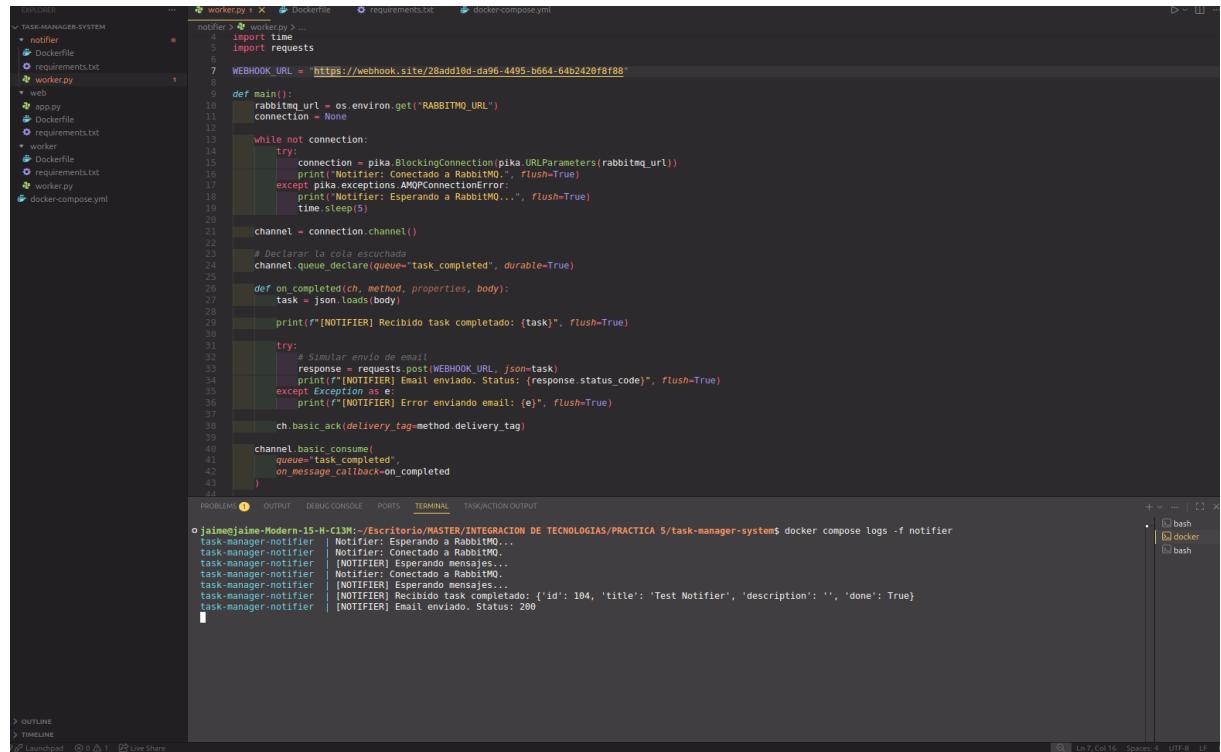
    # Asignar consumidores
    channel.basic_consume(queue='task_created', on_message_callback=on_created)

    print(" [*] Worker listo. Escuchando mensajes...")
    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.



```
notifier > worker.py ...
import time
import requests
WEBHOOK_URL = 'https://webhook.site/28add10d-da96-4495-b664-64b2420f8f88'
rabitmq_url = os.environ.get('RABBITMQ_URL')
connection = None
while not connection:
    try:
        connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
        print('Notifier: Conectado a RabbitMQ...', flush=True)
    except pika.exceptions.AMQPConnectionError:
        print('Notifier: Esperando a RabbitMQ...', flush=True)
        time.sleep(5)
channel = connection.channel()
# Declarar la cola escuchada
channel.queue_declare(queue='task_completed', durable=True)
def on_completed(ch, method, properties, body):
    task = json.loads(body)
    print(f'[NOTIFIER] Recibido task completado: {task}', flush=True)
    try:
        # Simular envio de email
        response = requests.post(WEBHOOK_URL, json=task)
        print(f'[NOTIFIER] Email enviado. Status: {response.status_code}', flush=True)
    except Exception as e:
        print(f'[NOTIFIER] Error enviando email: {e}', flush=True)
    ch.basic_ack(delivery_tag=method.delivery_tag)
channel.basic_consume(
    queue='task_completed',
    on_message_callback=on_completed
)
d.d
```

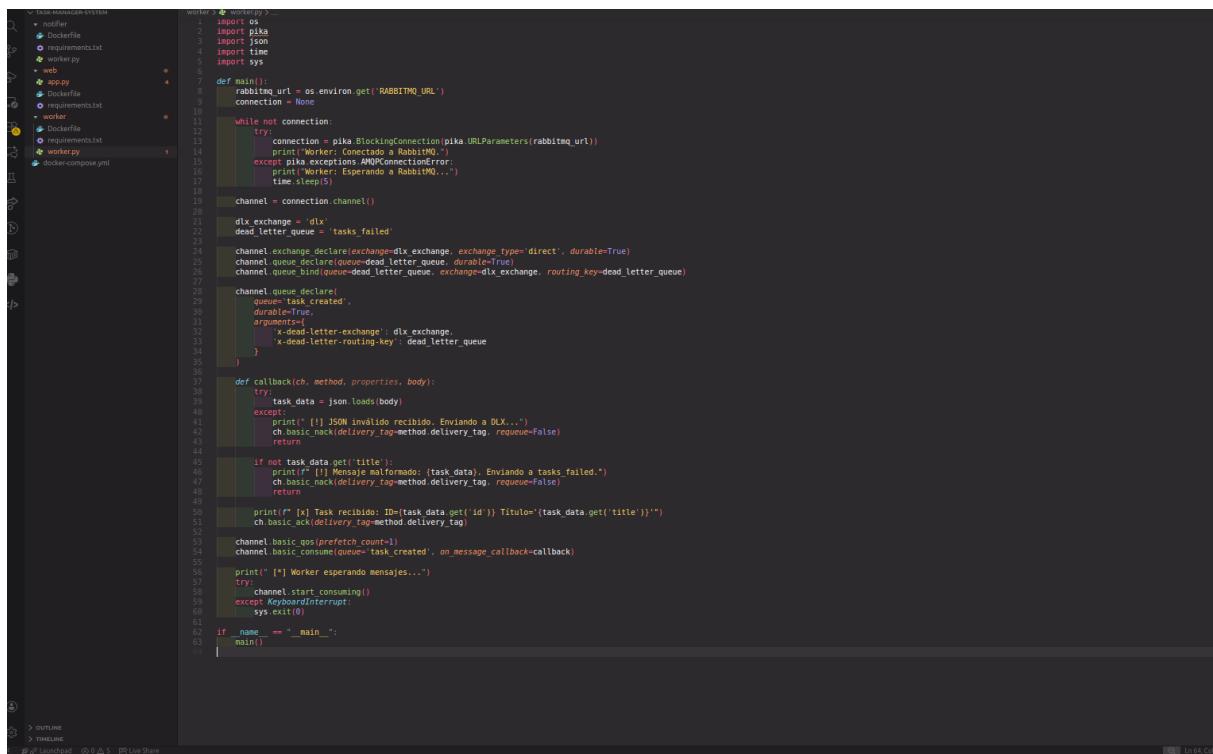
jaimojaime-MacBook-Pro-15-N-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system\$ docker compose logs -f notifier

```
task-manager-notifier | Notifier: Esperando a RabbitMQ...
task-manager-notifier | Notifier: Conectado a RabbitMQ...
task-manager-notifier | [NOTIFIER] Esperando mensajes...
task-manager-notifier | [NOTIFIER] Esperando a RabbitMQ...
task-manager-notifier | [NOTIFIER] Esperando mensajes...
task-manager-notifier | [NOTIFIER] Recibido task completado: {'id': 104, 'title': 'Test Notifier', 'description': '', 'done': True}
task-manager-notifier | [NOTIFIER] Email enviado. Status: 200
```

Capítulo 4

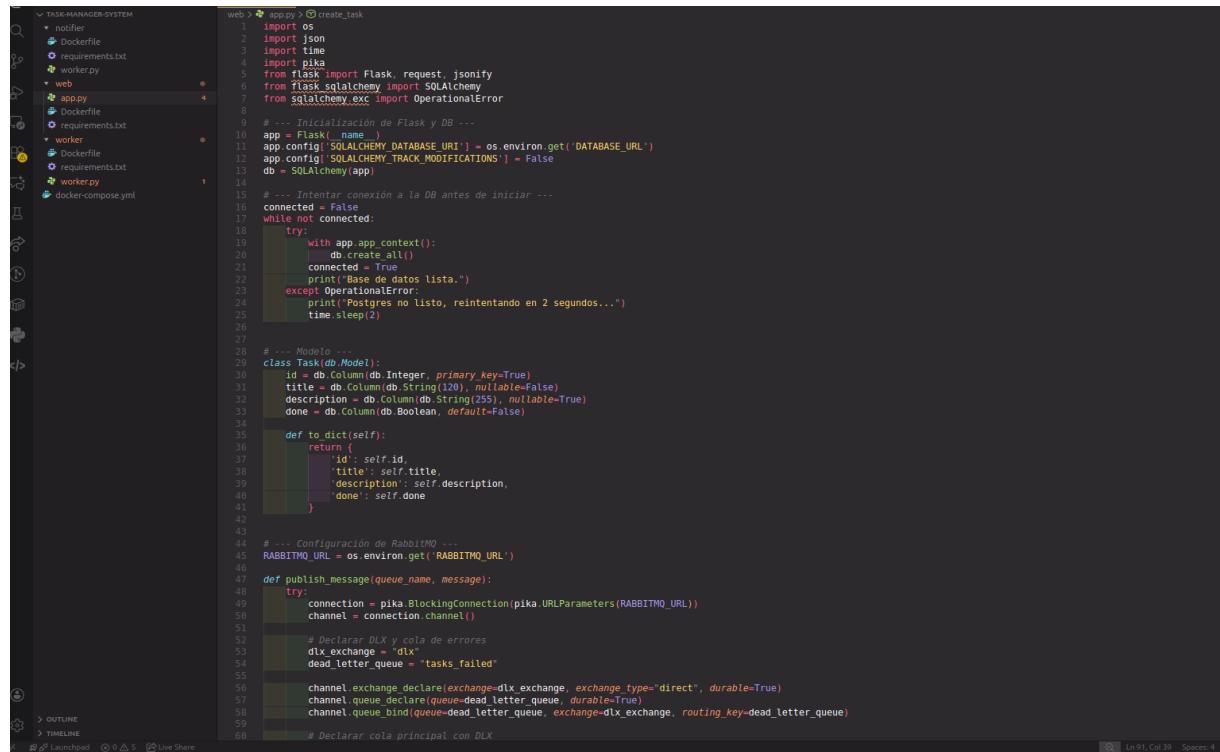
Ejercicio 3

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.



```
worker.py
1  #!/usr/bin/python3
2
3  import os
4  import pika
5  import json
6  import time
7  import sys
8
9  def main():
10    rabbitmq_url = os.environ.get('RABBITMQ_URL')
11    connection = None
12
13    while not connection:
14      try:
15        connection = pika.BlockingConnection(pika.URLParameters(rabbitmq_url))
16      except pika.exceptions.AMQPConnectionError:
17        print("[-] No se pudo conectar a RabbitMQ...")
18        time.sleep(5)
19
20    channel = connection.channel()
21
22    dlx_exchange = dlx
23    dead_letter_queue = 'tasks_failed'
24
25    channel.exchange_declare(exchange=dlx_exchange, exchange_type='direct', durable=True)
26    channel.queue_declare(queue=dead_letter_queue, durable=True)
27    channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)
28
29    channel.queue_declare(queue='tasks_created',
30                          durable=True,
31                          arguments={
32                            'x-dead-letter-exchange': dlx_exchange,
33                            'x-dead-letter-routing-key': dead_letter_queue
34                          })
35
36
37  def callback(ch, method, properties, body):
38    try:
39      task_data = json.loads(body)
40    except:
41      print("[-] JSON invalido recibido. Envioando a DLX...")
42      ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
43      return
44
45    if not task_data.get('title'):
46      print("[-] Mensaje malformado: (task_data). Envioando a tasks_failed...")
47      ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
48      return
49
50    print("[+] Task recibido: Id=(task_data.get('id')) Titulo=(task_data.get('title'))")
51    ch.basic_ack(method.delivery_tag)
52
53    channel.basic_qos(prefetch=1)
54    channel.basic_consume(queue='task_created', on_message_callback=callback)
55
56    print("[+] Worker esperando mensajes...")
57    try:
58      channel.start_consuming()
59    except KeyboardInterrupt:
60      sys.exit(0)
61
62  if __name__ == '__main__':
63    main()
```

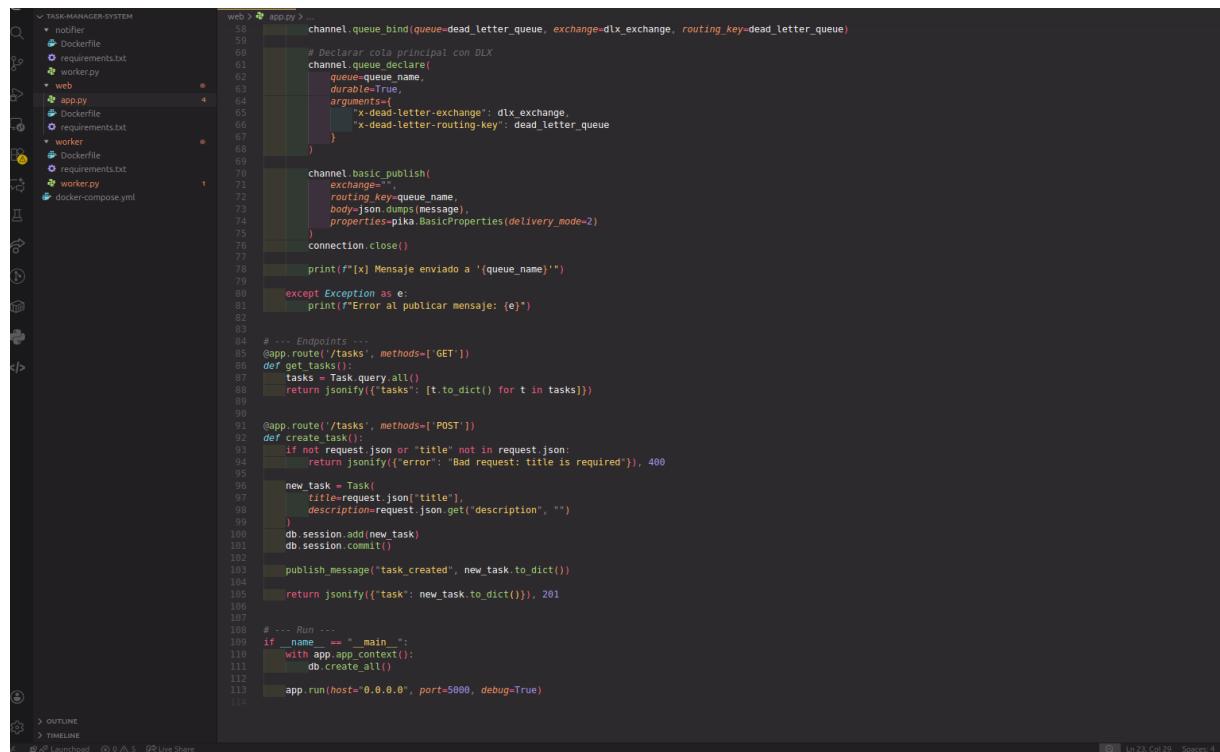
A continuación se ha modificado app.py para mandar cualquier mensaje rechazado por worker de la cola task-created a task-failed.



```

web > app.py > create_task
1 import os
2 import json
3 import time
4 import pika
5 from flask import Flask, request, jsonify
6 from flask_sqlalchemy import SQLAlchemy
7 from sqlalchemy.exc import OperationalError
8
9 # --- Inicialización de Flask y DB ---
10 app = Flask( name )
11 app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL')
12 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
13 db = SQLAlchemy(app)
14
15 # --- Intentar conexión a la DB antes de iniciar ---
16 connected = False
17 while not connected:
18     try:
19         with app.app_context():
20             db.create_all()
21             connected = True
22             print("Base de datos lista.")
23     except OperationalError:
24         print("Postgres no listo, reintentando en 2 segundos...")
25         time.sleep(2)
26
27 # -- Modelos --
28 class Task(db.Model):
29     id = db.Column(db.Integer, primary_key=True)
30     title = db.Column(db.String(120), nullable=False)
31     description = db.Column(db.String(255), nullable=True)
32     done = db.Column(db.Boolean, default=False)
33
34     def to_dict(self):
35         return {
36             'id': self.id,
37             'title': self.title,
38             'description': self.description,
39             'done': self.done
40         }
41
42
43 # --- Configuración de RabbitMQ ---
44 RABBITMQ_URL = os.environ.get('RABBITMQ_URL')
45
46 def publish_message(queue_name, message):
47     try:
48         connection = pika.BlockingConnection(pika.URLParameters(RABBITMQ_URL))
49         channel = connection.channel()
50
51         # Declarar DLX y cola de errores
52         dlx_exchange = "dlx"
53         dead_letter_queue = "tasks_failed"
54
55         channel.exchange_declare(exchange=dlx_exchange, exchange_type="direct", durable=True)
56         channel.queue_declare(queue=queue_name, durable=True)
57         channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)
58
59         # Declarar cola principal con DLX
60
61     except Exception as e:
62         print(f"[x] Mensaje enviado a '{queue_name}'")
63
64     except Exception as e:
65         print(f"Error al publicar mensaje: {e}")
66
67
68     # --- Endpoints ---
69 @app.route('/tasks', methods=['GET'])
70 def get_tasks():
71     tasks = Task.query.all()
72     return jsonify({'tasks': [t.to_dict() for t in tasks]})
73
74
75 @app.route('/tasks', methods=['POST'])
76 def create_task():
77     if not request.json or "title" not in request.json:
78         return jsonify({'error': 'Bad request: title is required'}), 400
79
80     new_task = Task(
81         title=request.json["title"],
82         description=request.json.get("description", ""))
83
84     db.session.add(new_task)
85     db.session.commit()
86
87     publish_message("task_created", new_task.to_dict())
88
89     return jsonify({'task': new_task.to_dict()}), 201
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114

```



```

web > app.py > ...
58         channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)
59
60         # Declarar cola principal con DLX
61         channel.queue_declare(
62             queue=queue_name,
63             durable=True,
64             arguments={
65                 "x-dead-letter-exchange": dlx_exchange,
66                 "x-dead-letter-routing-key": dead_letter_queue
67             }
68         )
69
70         channel.basic_publish(
71             exchange="",
72             routing_key=queue_name,
73             body=json.dumps(message),
74             properties=pika.BasicProperties(delivery_mode=2)
75         )
76
77         connection.close()
78
79         print(f"[x] Mensaje enviado a '{queue_name}'")
80
81     except Exception as e:
82         print(f"Error al publicar mensaje: {e}")
83
84
85     # --- Endpoints ---
86 @app.route('/tasks', methods=['GET'])
87 def get_tasks():
88     tasks = Task.query.all()
89     return jsonify({'tasks': [t.to_dict() for t in tasks]})
90
91
92 @app.route('/tasks', methods=['POST'])
93 def create_task():
94     if not request.json or "title" not in request.json:
95         return jsonify({'error': 'Bad request: title is required'}), 400
96
97     new_task = Task(
98         title=request.json["title"],
99         description=request.json.get("description", ""))
100
101     db.session.add(new_task)
102     db.session.commit()
103
104     publish_message("task_created", new_task.to_dict())
105
106     return jsonify({'task': new_task.to_dict()}), 201
107
108
109
110
111
112
113
114

```

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

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":100, "description":"sin title"}

Payload encoding: String (default)

Publish message

Get messages
Move messages
Delete
Purge
Runtime Metrics (Advanced)

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.

Virtual host	Name	Type	Features	State	Messages			Message rates		
					Ready	Unacked	Total	incoming	deliver / get	ack
/	task_completed	classic	D	running	0	0	0	0.20/s	0.20/s	0.00/s
/	task_created	classic	D DLX DLK	running	0	0	0			
/	tasks_failed	classic	D	running	2	0	2			

Add new queue