

Memoria Práctica 6

Jaime Parra Jiménez

INTEGRACIÓN DE TECNOLOGÍAS Y SERVICIOS INFORMÁTICOS

Universidad de Almería

Correo: jpj451@inlumine.ual.es

27 de noviembre de 2025

Índice general

1. Ejercicio Guiado 1	2
2. Ejercicio 1	7
3. Ejercicio 2	9
4. Ejercicio 3	11

Capítulo 1

Ejercicio Guiado 1

El objetivo del ejercicio guiado es conectar n8n el sistema de la práctica 5. n8n leerá datos de la base de datos PostgreSQL, y luego actuará como productor y consumidor de mensajes de RabbitMQ. El primer paso será preparar el entorno de microservicios. Para ello se debe iniciar la pila de servicios en segundo plano usando el comando docker-compose up -d –build.

```
worker> worker> 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.")
 15         except pika.exceptions.AMQPConnectionError:
 16             print("Worker: Esperando a RabbitMQ...")
 17             time.sleep(5)
 18
 19     channel = connection.channel()
 20
 21     dlx_exchange = 'dlx'
 22     dead_letter_queue = 'tasks_failed'
 23
 24     channel.exchange_declare(exchange=dlx_exchange, exchange_type='direct', durable=True)
 25     channel.queue_declare(queue=dead_letter_queue, durable=True)
 26     channel.queue_bind(queue=dead_letter_queue, exchange=dlx_exchange, routing_key=dead_letter_queue)
 27
 28     channel.queue_declare(
 29         queue='task_created',
 30         durable=True,
 31         arguments={
 32             'x-dead-letter-exchange': dlx_exchange,
 33
 34             PROBLEMS 0 OUTPUT DEBUG CONSOLE PORTS TERMINAL TASK/ACTION OUTPUT
jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker compose up -d --build + bash
=> exporting layers
=> writing image sha256:a9579536238a1598ab91448ec86b5e5a5afe07b21f5d29e031406235263da80
=> naming to docker.io/library/task-manager-system-worker
=> [notifier] resolving provenance for metadata file
=> [web] resolving provenance for metadata file
=> [worker] resolving provenance for metadata file
[+] Running 8/8
 ✓ task-manager-system-web           Built
 ✓ task-manager-system-worker         Built
 ✓ task-manager-system-notifier      Built
 ✓ Container task-manager-mq        Started
Container 56fe3920031f task-manager-db    Started
Container task-manager-web          Started
Container task-manager-worker       Started
Container task-manager-notifier    Started
[+] Running 0/0
jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$
```

A continuación, es necesario conectar n8n a la red de microservicios. Para conectarlo se usará el siguiente comando.

The screenshot shows a terminal window with the following logs:

```

jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker ps
CONTAINER ID        IMAGE               COMMAND             STATUS              PORTS
5206b0058d0c        task-manager-system-notifier   "python worker.py"   39 minutes ago      Up 12 minutes
ee5b2d366b40        task-manager-system-worker     "python worker.py"   39 minutes ago      Up 12 minutes
56fe3920            ager-worker
56fe3920            postgres:14-alpine           "docker-entrypoint.s" 39 minutes ago      Up 12 minutes    0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp
031f task-manager-db
dad9fd75235c        rabbitmq:3-management-alpine  "docker-entrypoint.s" 39 minutes ago      Up 12 minutes    4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, [::]:5672->5672/tcp, 15671/tcp, 15691-15692/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp, [::]:15672->15672/tcp
235e task-manager-mq
jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ docker network connect task-man manager-system default n8n
jaime@jaime-Modern-15-H-C13M:~/Escritorio/MASTER/INTEGRACION DE TECNOLOGIAS/PRACTICA 5/task-manager-system$ 

```

Below the terminal, there is a code editor window showing a Python file named `worker.py`:

```

# worker.py
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()

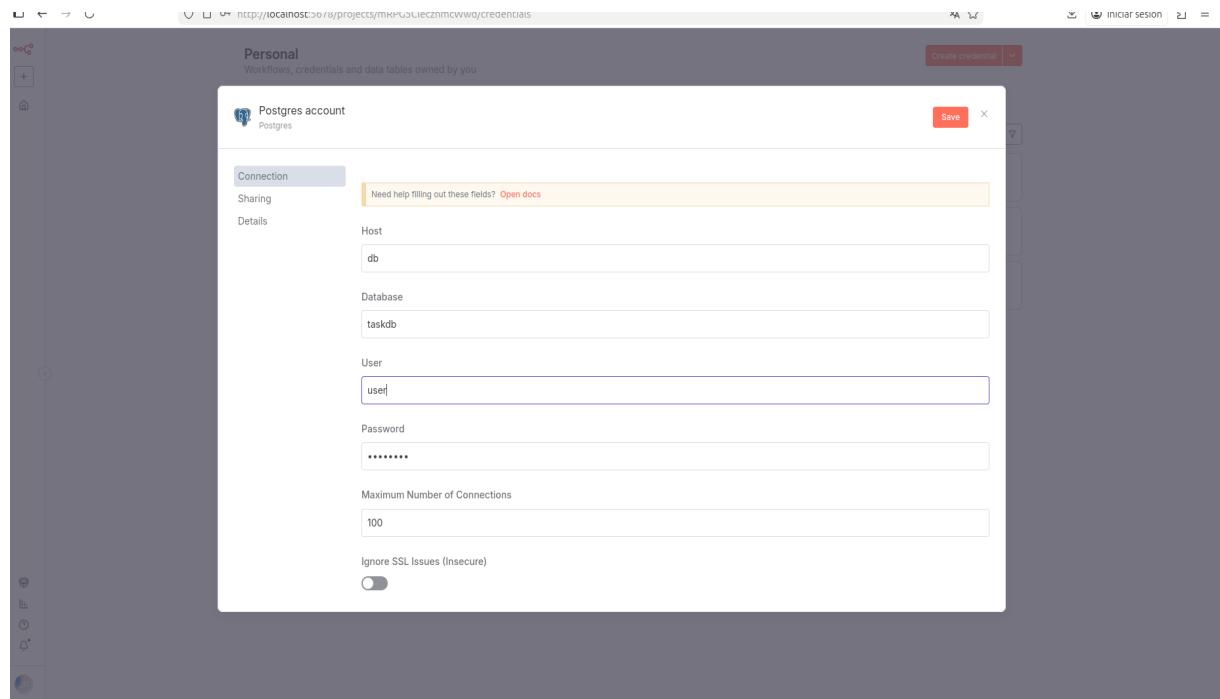
    dlx_exchange = dlx
    dead_letter_queue = 'tasks_failed'

    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)

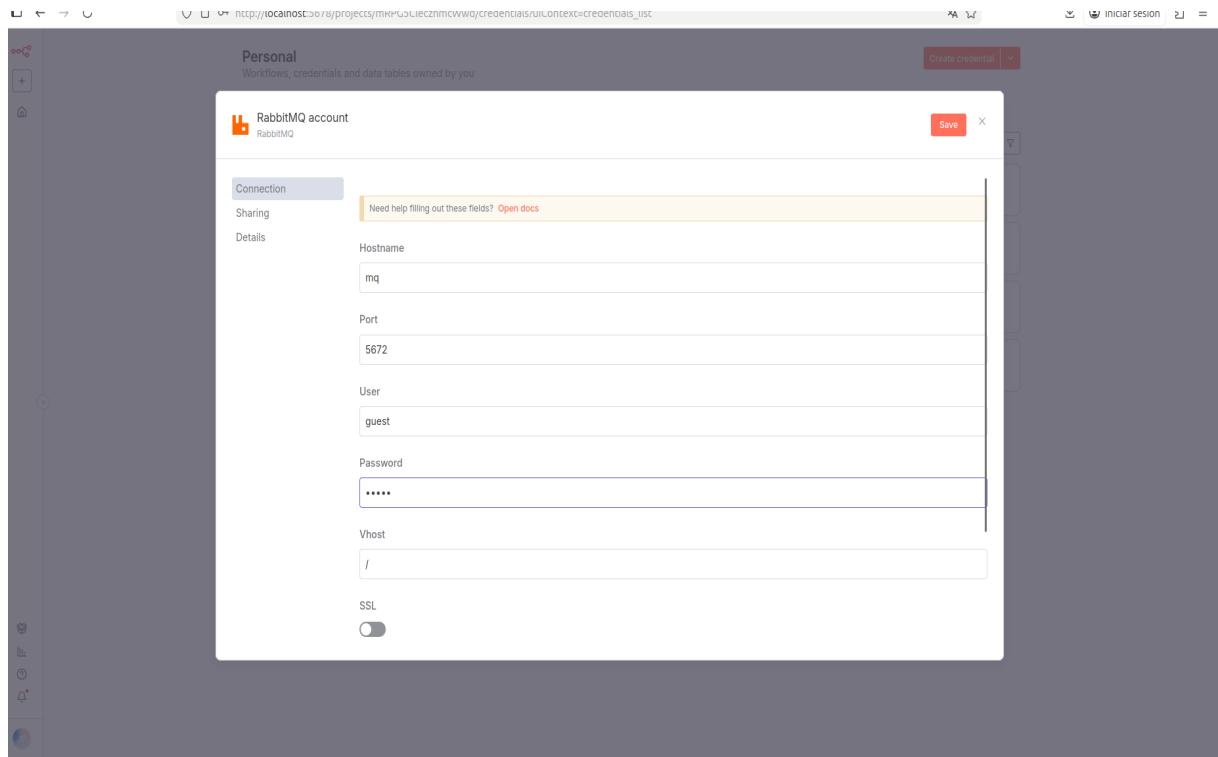
    channel.queue_declare(
        queue='task_created',
        durable=True,
        arguments={
            'x-dead-letter-exchange': dlx_exchange,
            'x-dead-letter-routing-key': dead_letter_queue
        }
    )

```

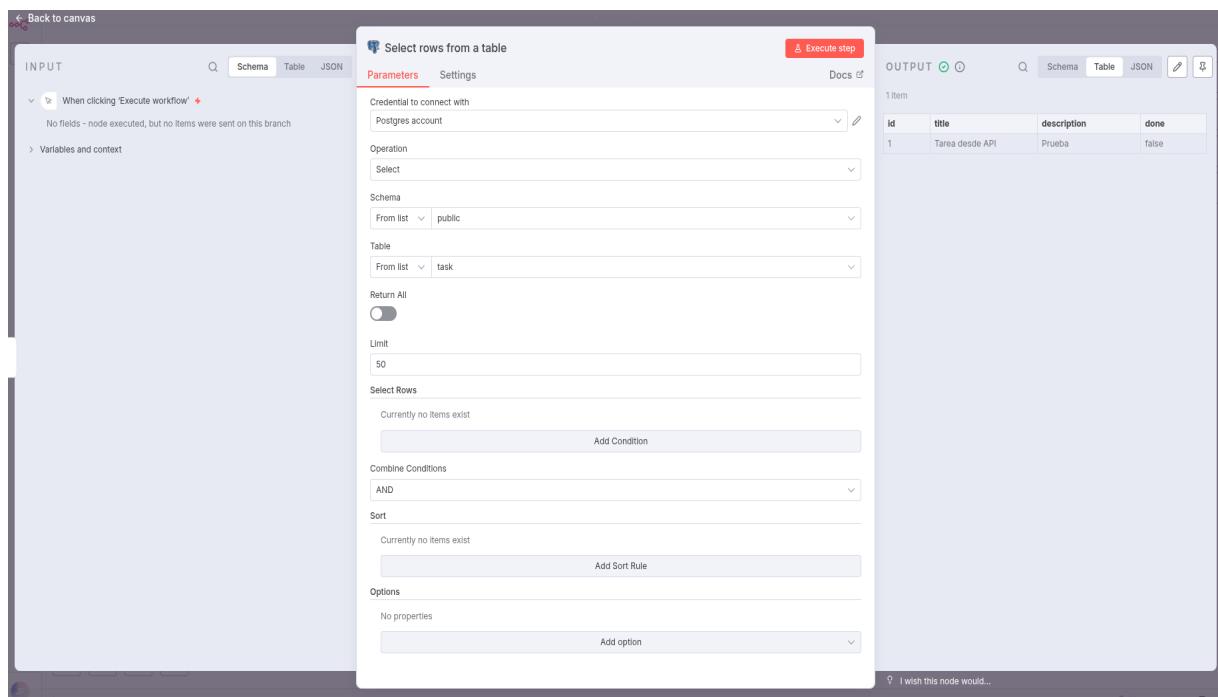
Una vez conectado n8n a la red es necesario configurar las credenciales en n8n. La primera credencial es PostgreSQL. Le asignaremos el host, database, user, password y port.



La segunda credencial es RabbitMQ donde se le asignará el host, user, password y port.



El siguiente paso es crear un flujo que lee todas las tareas de la base de datos. Para ello, se añade un nodo Manual Trigger y a continuación un nodo PostgreSQL. Se debe seleccionar la tabla task, la misma que en la práctica 5, y la operación de select.



Además, también se crearán dos flujos para demostrar el patrón asíncrono. El primer flujo será productor, que recibirá una petición web y enviará la tarea a RabbitMQ. El nombre del flujo es Productor de Tareas y comenzará con un nodo Webhook Trigger.

headers	params	query	body	webhookUrl	executionMode
host : localhost:5678 user-agent : curl/8.5.0 accept : */* content-type : application/json content-length : 73	{empty object}	{empty object}	title : Tarea enviada por Webhook n8n description : A cola RabbitMQ	http://localhost:5678/webhook-test/crear_tarea	test

A continuación se añade un nodo RabbitMQ para que use la cola task-created y guardar los datos en esta cola.

headers	params	query	body	webhookUrl	executionMode
			success true		

Para crear el flujo consumidor, llamado Consumidor de Tareas, se pondrá un RabbitMQ Trigger como primer nodo para que escuche la cola task-created.

The screenshot shows the n8n interface with a 'RabbitMQ Trigger' node selected. The 'Parameters' tab is active, displaying the following configuration:

- Credential to connect with: RabbitMQ account
- Queue / Topic: task_created
- Options: No properties

Below the node configuration, there is a note: "Once you've finished building your workflow, **activate** it to have it also listen continuously (you just won't see those executions here). [More info](#)".

The 'OUTPUT' panel on the right shows the schema of the data received from the trigger:

fields	properties	content
consumerTag : amq.ctag-cCJ7UTT5MbFXU3KTxOCemQ deliveryTag : 1 redelivered : false exchange : empty routingKey : task_created	headers : {empty object}	{}

Por último, se añadirá un nodo Edit Fields(Set) para simular un procesamiento. Se añadirá un campo con el nombre de procesado-por y el valor de n8n-consumidor.

The screenshot shows the n8n interface with an 'Edit Fields' node selected. The 'Parameters' tab is active, displaying the following configuration:

- Mode: Manual Mapping
- Fields to Set:
 - procesado_por : String
n8n_consumidor

The 'INPUT' panel on the left shows the data received from the previous RabbitMQ Trigger node:

- fields:
 - consumerTag : amq.ctag-cCJ7UTT5MbFXU3KTxOCemQ
 - deliveryTag : 1
 - redelivered : false
 - exchange :
 - routingKey : task_created
- properties
- headers
- content : {}

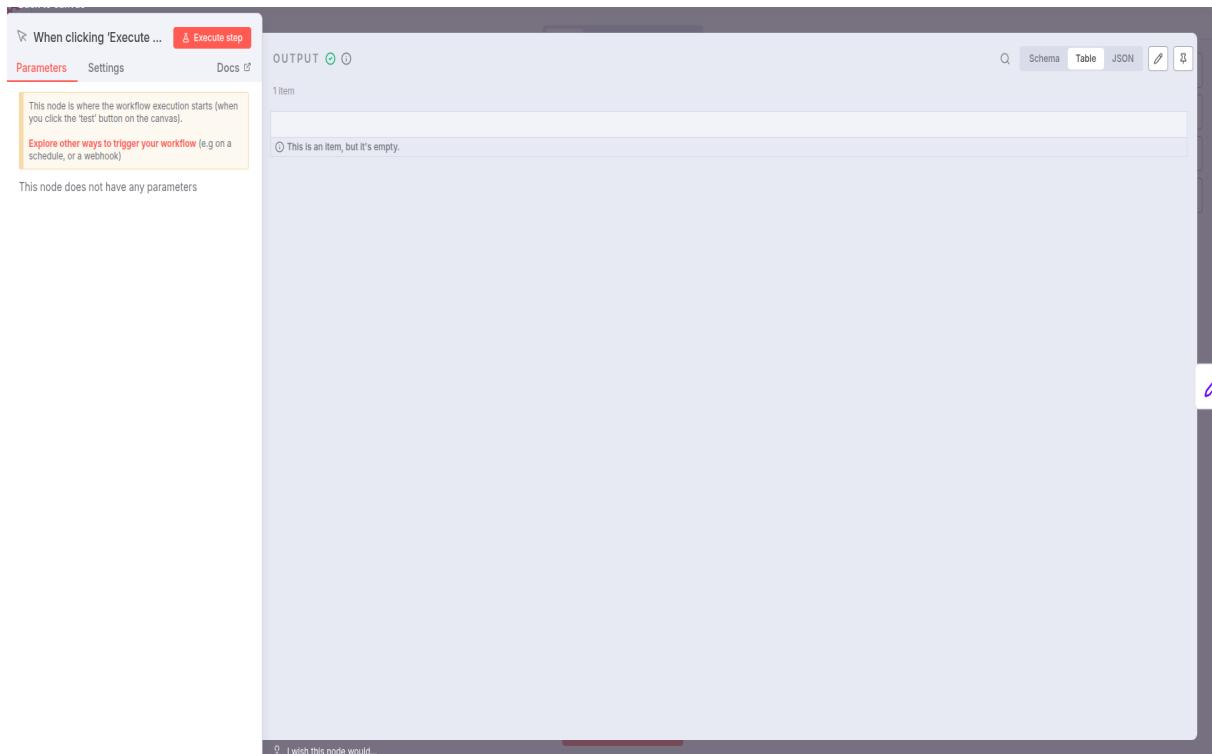
The 'OUTPUT' panel on the right shows the modified data after the 'Edit Fields' node:

fields
procesado_por : n8n_consumidor

Capítulo 2

Ejercicio 1

El objetivo del ejercicio 1 es crear un flujo de trabajo que elimine una tarea de la base de datos PostgreSQL. Para conseguir el objetivo lo primero que se debe hacer es añadir un Manual Trigger para dar comienzo al flujo de trabajo.



A continuación, se añade un Edit Fields(Set) para simular que tarea eliminar. Para definir la tarea se usará su ID.

El último paso será añadir un no PostgreSQL con la operación delete. Para eliminar la tarea se configura la condición WHERE para que eliminar la fila donde la id sea igual a la id pasada.

Capítulo 3

Ejercicio 2

El objetivo del ejercicio 2 es crear un endpoint para actualizar una tarea para que pase de la cola creada a completada. Para conseguir el objetivo debemos añadir el endpoint de PUT en el archivo app.py.

```
@app.route("/tasks/<int:task_id>/complete", methods=["PUT"])
def complete_task(task_id):
    task = Task.query.get(task_id)

    if not task:
        return jsonify({"error": "task not found"}), 404

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

    event = {
        "id": task.id,
        "title": task.title,
        "status": "completed"
    }

    publish_message("task_completed", event)

    return jsonify(task.to_dict()), 200
```

El siguiente paso será crear un flujo de trabajo que comience con un nodo RabbitMQ Trigger para que escuche la cola task-completed.

The screenshot shows a workflow editor interface with a central workspace and two panels on the right: INPUT and OUTPUT.

RabbitMQ Trigger Node Configuration:

- Parameters:**
 - Credential to connect with: RabbitMQ account
 - Queue / Topic: task_completed
 - Options: No properties
- OUTPUT:** Shows 1 item with fields, properties, and content. The content is a JSON object:


```
{"id": 5, "title": "Tarea de prueba", "description": null, "done": false}
```

Bottom Panel:

- When will this node trigger my flow? ▾
- I wish this node would... ▾

Cuando reciba un mensaje, el flujo debe usar un nodo Send Email para notificar que la tarea ha sido completada. En el email se pondrá el título y ID de la tarea.

The screenshot shows a workflow editor interface with a central workspace and two panels on the right: INPUT and OUTPUT.

Send Email Node Configuration:

- Parameters:**
 - Credential to connect with: SMTP account
 - Operation: Send
 - From Email: jp451@inlumine.ual.es
 - To Email: jp451@inlumine.ual.es
 - Subject: Tarea completada: {{JSON.parse(\$json.content).title}}
 - Content (HTML): La tarea "{{JSON.parse(\$json.content).title}} (ID: {{JSON.parse(\$json.content).id}}) ha sido marcada como completada.
- OUTPUT:** Shows 1 item with fields, properties, and content. The content is a JSON object:


```
{
  "accepted": [
    "jp451@inlumine.ual.es"
  ],
  "rejected": [
  ],
  "ehlo": [
    "SIZE 3588257",
    "8BITMIME",
    "AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH",
    "ENHANCEDSTATUSCODES",
    "PIPELINING",
    "CHUNKING",
    "SMTPUTF8"
  ],
  "envelopeTime": 276,
  "messageTime": 504,
  "messageSize": 711,
  "response": "250 2.0.0 OK 1764191713 5b1f17b1804b1-4790b0c3a28sm64914665e9.9 - gsmtp",
  "envelope": {
    "from": "jp451@inlumine.ual.es",
    "to": [
      "jp451@inlumine.ual.es"
    ]
  },
  "messageId": "<56c2aceb-7528-ff47-7059-cc145ae7ee82@inlumine.ual.es>"
}
```

Bottom Panel:

- I wish this node would... ▾
- Logs
- Node executed successfully

Capítulo 4

Ejercicio 3

El objetivo del ejercicio 3 es reemplazar completamente la lógica del endpoint POST /tasks de la API de Flask con un flujo de trabajo de n8n. Para ello es necesario crear un nuevo flujo de trabajo que comience con un Webhook Trigger.

The screenshot shows the n8n interface with a 'Webhook' node selected. The configuration pane on the left includes:

- Parameters**: Test URL: http://localhost:5678/webhook-test/task, HTTP Method: POST, Path: task, Authentication: None, Respond: Immediately.
- A note: "If you are sending back a response, add a 'Content-Type' response header with the appropriate value to avoid unexpected behavior".
- Options**: No properties, Add option.

The right panel shows the **OUTPUT** tab with the following JSON response:

```
[{"headers": {"host": "localhost:5678", "user-agent": "curl/8.5.0", "accept": "*/*", "content-type": "application/json", "content-length": "76"}, "params": {}, "query": {}, "body": {"id": 123, "title": "Tarea de prueba", "description": "Esto es una prueba"}, "webhookUrl": "http://localhost:5678/webhook-test/task", "executionMode": "test"}]
```

A continuación, se añadirá un nodo IF para validar que el campo de title existe y no esta vacio.

The screenshot shows the MuleSoft Anypoint Studio interface with an 'If' node selected. The 'Parameters' tab is active. In the 'Conditions' section, there is a condition: `if {{ $json.body.title }} is not empty`. Below it, there is a button 'Add condition' and a link 'Convert types where required'. The 'True Branch' output shows the original JSON with the 'title' field. The 'False Branch' output is empty.

Si el campo esta vacio, se usará un nodo Stop and Error para comunicar el error al usuario.

The screenshot shows the MuleSoft Anypoint Studio interface with a 'Stop and Error' node selected. The 'Parameters' tab is active. The 'Error Type' dropdown is set to 'Error Message' and the 'Error Message' input field contains 'Fallido'. The 'OUTPUT' panel indicates 'No data' and 'Execute this node to view data or set mock data'.

Si la entrada de datos es correcta se dirigirá a un nodo PostgreSQL para insertar una nueva tarea a la tabla task.

The screenshot shows the Mule ESB Anypoint Studio interface with the 'Insert rows in a table' node selected. The 'INPUT' tab displays a JSON payload representing a webhook message. The 'Parameters' tab is active, showing the connection to a 'Postgres account', the operation as 'Insert', and the schema as 'public'. The 'Table' dropdown is set to 'task'. The 'Values to Send' section maps the JSON fields 'title' and 'description' to database columns. The 'done' field is set to true. The 'OUTPUT' tab shows the resulting JSON output, which is a single item containing the inserted task details.

El último paso será publicar los datos de la tarea en la cola de task-created de Rabbit. Para ello, se usará un nodo RabbitMQ.

The screenshot shows the Mule ESB Anypoint Studio interface with the 'RabbitMQ' node selected. The 'INPUT' tab shows the same JSON payload from the previous step. The 'Parameters' tab is active, connecting to a 'RabbitMQ account', performing a 'Send a Message to RabbitMQ' operation, and using a 'Queue' mode. The 'Queue / Topic' is set to 'task_created'. The 'Send Input Data' field contains the expression '\$JSON.stringify(\$json)'. The 'OUTPUT' tab shows the resulting JSON output, which is a single item indicating success.