

Universidad de Almería

Máster en Ingeniería Informática

Integración de Tecnologías y Servicios Informáticos

Práctica 6

Orquestación de Microservicios - Integración con PostgreSQL y
RabbitMQ

Autor: Johan Eduardo Cala Torra

Fecha: 9 de diciembre de 2025

Curso 2024-2025

Índice

1. Introducción	3
1.1. Objetivos de Aprendizaje	3
2. Fundamentos Teóricos	4
2.1. Nodo PostgreSQL en n8n	4
2.2. Nodos RabbitMQ en n8n	4
2.3. El Desafío de la Red Docker	4
3. Flujo de Trabajo Guiado	5
3.1. Paso 1: Preparar el Entorno de Microservicios	5
3.2. Paso 2: Conectar n8n a la Red de Microservicios	5
3.3. Paso 3: Configurar las Credenciales en n8n	6
3.3.1. Credencial de PostgreSQL	6
3.3.2. Credencial de RabbitMQ	6
3.4. Paso 4: Flujo 1 - Interacción Directa con PostgreSQL	8
3.4.1. Estructura del Flujo	8
3.4.2. Resultado de la Ejecución	8
3.5. Paso 5: Flujo 2 - Productor y Consumidor Asíncrono	9
3.5.1. Flujo 2A: El Productor (Webhook → RabbitMQ)	9
3.5.2. Prueba del Productor	11
3.5.3. Flujo 2B: El Consumidor (RabbitMQ → Procesamiento)	12
3.5.4. Ejecución del Consumidor	12
3.6. Verificación en RabbitMQ Management	13
4. Ejercicios Propuestos	14
4.1. Ejercicio 1: Eliminación de Tareas	14
4.1.1. Objetivo y Requisitos	14
4.1.2. Implementación del Flujo	14
4.1.3. Configuración del Nodo PostgreSQL Delete	15
4.1.4. Ejecución y Resultados	15
4.2. Ejercicio 2: Servicio de Notificación	17
4.2.1. Objetivo y Requisitos	17
4.2.2. Parte A: Completar Tarea vía API	17
4.2.3. Parte B: Notificador de Tareas Completadas	19
4.2.4. Configuración del Nodo RabbitMQ Trigger	19
4.2.5. Procesamiento del Mensaje	20
4.2.6. Preparación de la Notificación	22
4.2.7. Configuración de Gmail	23
4.2.8. Prueba End-to-End	23
4.3. Ejercicio 3: Reemplazo de API con Webhook	25
4.3.1. Objetivo y Requisitos	25
4.3.2. Arquitectura del Flujo	25
4.3.3. Configuración del Webhook	27
4.3.4. Validación de Entrada	29
4.3.5. Inserción en PostgreSQL	31
4.3.6. Publicación en RabbitMQ	33
4.3.7. Respuestas HTTP	33

4.3.8. Pruebas del Endpoint	36
4.3.9. Comparación con API Flask	39
5. Conclusiones	40
5.1. Lecciones Aprendidas	40
5.2. Relación con Práctica 5	40

1. Introducción

Esta práctica conecta nuestra instancia de n8n con el ecosistema de microservicios construido en la Práctica 5. El objetivo es posicionar a n8n como el **orquestador** o “pegamento” de alto nivel que consume y coordina los servicios existentes.

1.1. Objetivos de Aprendizaje

- **Integración de Bases de Datos:** Configurar y utilizar el nodo PostgreSQL para operaciones CRUD directas.
- **Comunicación Asíncrona (Producción):** Implementar el nodo RabbitMQ Send para publicar mensajes en colas.
- **Comunicación Asíncrona (Consumo):** Diseñar flujos reactivos con RabbitMQ Trigger.
- **Gestión de Redes Docker:** Conectar contenedores de diferentes redes de Docker.
- **Orquestación de Procesos:** Coordinar múltiples servicios (API, BBDD, colas) desde n8n.

2. Fundamentos Teóricos

2.1. Nodo PostgreSQL en n8n

El nodo PostgreSQL de n8n permite interactuar directamente con bases de datos relacionales sin pasar por APIs intermedias. Las operaciones disponibles incluyen:

- **Select:** Ejecuta consultas SELECT y devuelve resultados como JSON.
- **Insert:** Inserta registros mapeando claves JSON a columnas.
- **Update:** Actualiza filas basándose en condiciones.
- **Delete:** Elimina filas que coinciden con criterios.
- **Execute Query:** Control total con SQL personalizado.

2.2. Nodos RabbitMQ en n8n

n8n puede participar en arquitecturas de mensajería asíncrona como productor y consumidor:

- **RabbitMQ (Send):** Actúa como productor, publicando mensajes en colas.
- **RabbitMQ Trigger:** Actúa como consumidor, iniciando flujos al recibir mensajes.

2.3. El Desafío de la Red Docker

Por defecto, Docker Compose crea una red interna para sus servicios. El contenedor de n8n está fuera de esa red y no puede resolver nombres como `db` o `mq`. La solución es conectar n8n a la red del Docker Compose:

```
docker network connect task-manager-service_default n8n-practicas
```

3. Flujo de Trabajo Guiado

3.1. Paso 1: Preparar el Entorno de Microservicios

Se verifica que los 6 contenedores de la Práctica 5 están en ejecución:

```
cd practica-05/task-manager-service
docker-compose up -d
docker-compose ps
```

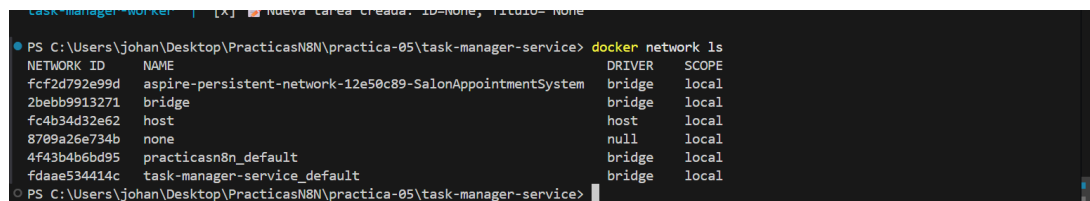
Los servicios activos incluyen: web, worker, notifier, error-handler, db y mq.

3.2. Paso 2: Conectar n8n a la Red de Microservicios

Para que n8n pueda comunicarse con PostgreSQL (db) y RabbitMQ (mq) usando sus nombres de servicio, conectamos el contenedor a la red:

```
# Verificar el nombre de la red
docker network ls

# Conectar n8n a la red
docker network connect task-manager-service_default n8n-practicas
```



```
Task manager worker [x] Nueva tarea creada. ID=None, Titulo=None
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service> docker network ls
NETWORK ID          NAME                                                    DRIVER  SCOPE
fcf2d792e99d        aspire-persistent-network-12e50c89-SalonAppointmentSystem  bridge  local
2bebb9913271        bridge                                                  bridge  local
fc4b34d32e62        host                                                    host    local
8709a26e734b        none                                                    none    local
4f43b4b6bd95        practicasn8n_default                                    bridge  local
fdaae534414c        task-manager-service_default                          bridge  local
PS C:\Users\johan\Desktop\PracticasN8N\practica-05\task-manager-service>
```

Figura 1: Conexión de n8n a la red de Docker Compose

3.3. Paso 3: Configurar las Credenciales en n8n

3.3.1. Credencial de PostgreSQL

Se configura la conexión a la base de datos PostgreSQL del sistema de tareas:

Campo	Valor
Credential Name	PostgreSQL (Tasks)
Host	db
Database	taskdb
User	user
Password	password
Port	5432
SSL	false

Cuadro 1: Configuración de credencial PostgreSQL

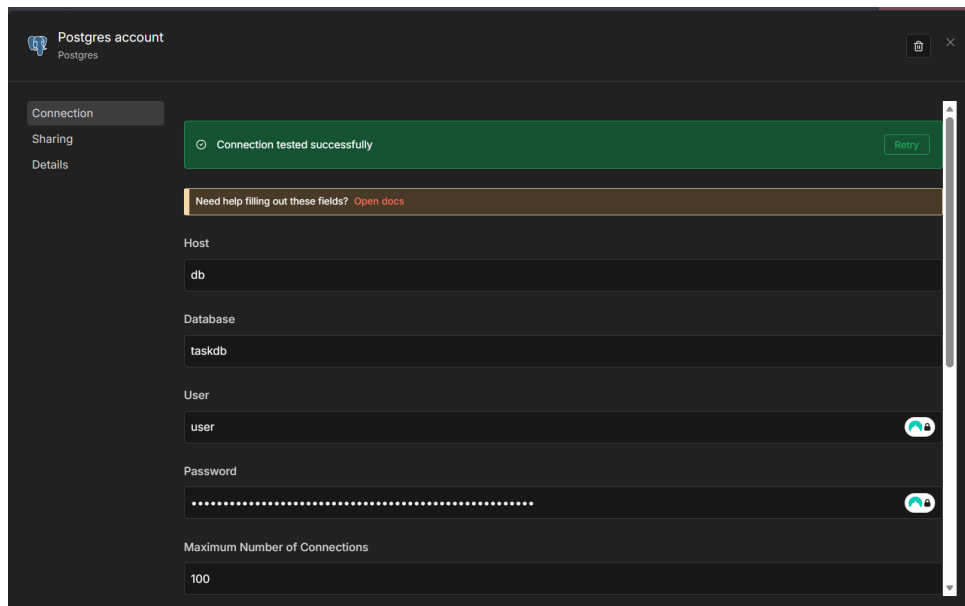
The image shows a screenshot of the n8n web interface for configuring a PostgreSQL credential. The window title is 'Postgres account'. On the left, there's a sidebar with 'Connection', 'Sharing', and 'Details' tabs. The main area shows a green success message: 'Connection tested successfully' with a 'Retry' button. Below this is a warning bar: 'Need help filling out these fields? Open docs'. The form fields are: 'Host' (db), 'Database' (taskdb), 'User' (user), 'Password' (masked with dots), and 'Maximum Number of Connections' (100). Each input field has a small icon on the right.

Figura 2: Formulario de credencial PostgreSQL en n8n

3.3.2. Credencial de RabbitMQ

Se configura la conexión al broker de mensajes RabbitMQ:

Campo	Valor
Credential Name	RabbitMQ (Tasks)
Host	mq
User	guest
Password	guest
Port	5672

Cuadro 2: Configuración de credencial RabbitMQ

RabbitMQ account

RabbitMQ

Connection

Sharing

Details

Connection tested successfully [Retry](#)

Need help filling out these fields? [Open docs](#)

Hostname

mq

Port

5672

User

guest

Password

.....

Vhost

/

Figura 3: Formulario de credencial RabbitMQ en n8n

3.4. Paso 4: Flujo 1 - Interacción Directa con PostgreSQL

Este flujo demuestra cómo n8n puede leer y escribir directamente en la base de datos PostgreSQL sin pasar por la API Flask.

3.4.1. Estructura del Flujo

El flujo consiste en:

1. **Manual Trigger:** Inicia la ejecución manualmente.
2. **PostgreSQL (Select):** Consulta la tabla `tasks` y devuelve todas las tareas.

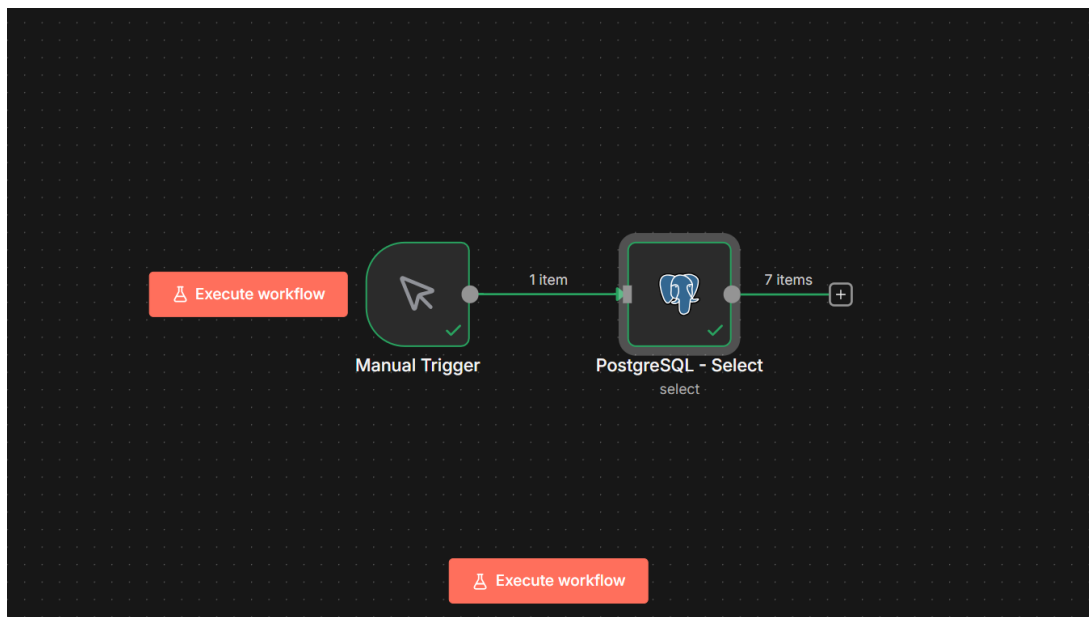


Figura 4: Flujo 1 - Vista del editor con nodos Manual Trigger y PostgreSQL

3.4.2. Resultado de la Ejecución

Al ejecutar el flujo, el nodo PostgreSQL devuelve todas las tareas almacenadas en formato JSON:

La interfaz de n8n muestra los logs de la ejecución. El nodo "PostgreSQL - Select" reportó éxito en 12ms. El panel de "OUTPUT" muestra 7 ítems de datos.

id	title	description	done
2	Implementar Dead Letter Queue	Ejercicio 3 - Manejo de mensajes fallidos	false
3	Crear servicio Notifier	Ejercicio 2 - Notificaciones via webhooks	false
4	Tareas de prueba para logs	Verificar que worker y notifier funcionan	true
5	Configurar sistema multi-contenedor	Usar Docker Compose	false
6	Configurar sistema multi-contenedor test	Usar Docker Compose test	true
7	Mi nueva tarea m1n1	Descripción de la tarea de m1n1	false
8	Mi nueva tarea m1n2	Descripción de la tarea de m1n2	false

Figura 5: Resultado del SELECT mostrando las tareas de la base de datos

3.5. Paso 5: Flujo 2 - Productor y Consumidor Asíncrono

Este ejercicio demuestra el patrón de comunicación asíncrona donde n8n actúa tanto como productor (envía mensajes) como consumidor (recibe mensajes) de RabbitMQ.

3.5.1. Flujo 2A: El Productor (Webhook → RabbitMQ)

Este flujo recibe peticiones HTTP y envía los datos a la cola `task_created` de RabbitMQ.

Estructura del flujo:

1. **Webhook Trigger:** Recibe peticiones POST con datos de la tarea.
2. **RabbitMQ Send:** Envía el mensaje a la cola `task_created`.
3. **Respond to Webhook:** Devuelve confirmación al cliente.

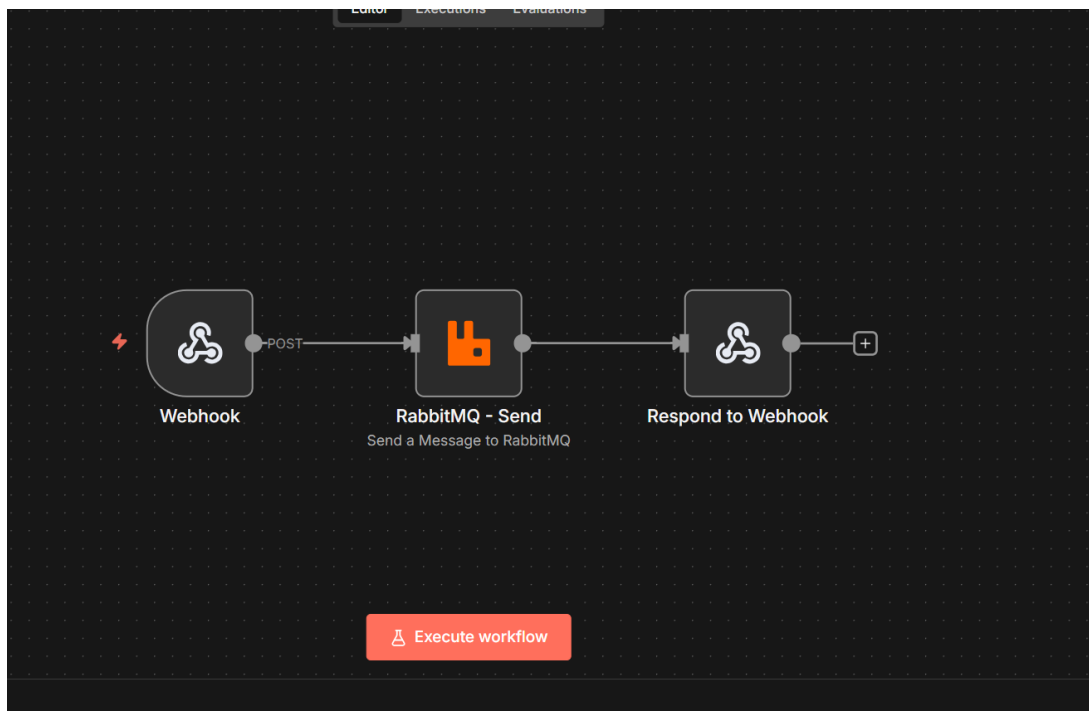


Figura 6: Flujo Productor - Webhook que envía mensajes a RabbitMQ

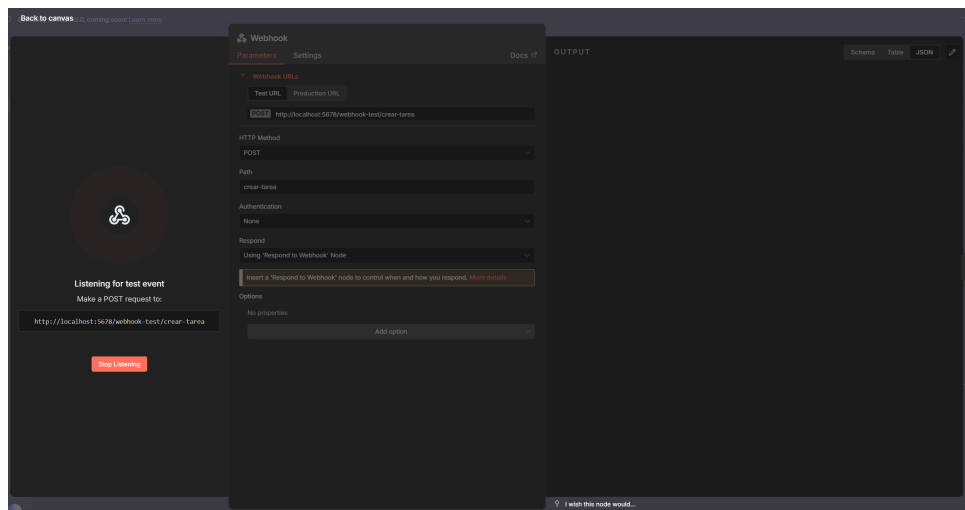


Figura 7: URL del Webhook para enviar tareas

3.5.2. Prueba del Productor

Se envía una petición POST al webhook para crear una tarea:

```
curl -X POST http://localhost:5678/webhook/crear-tarea \
-H "Content-Type: application/json" \
-d '{"title": "Tarea desde n8n", "description": "Test"}'
```

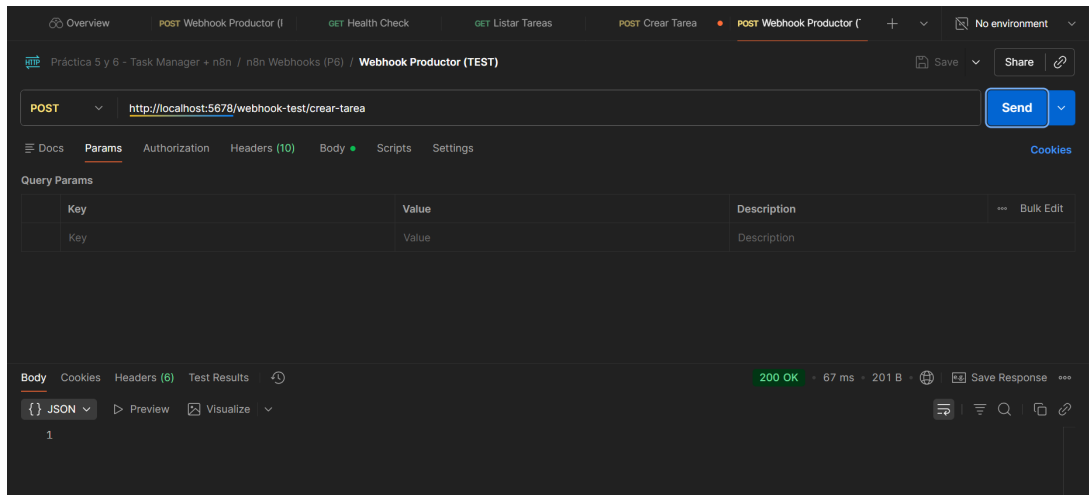


Figura 8: Prueba del webhook con curl/Thunder Client

El worker de Python (Práctica 5) recibe y procesa el mensaje:

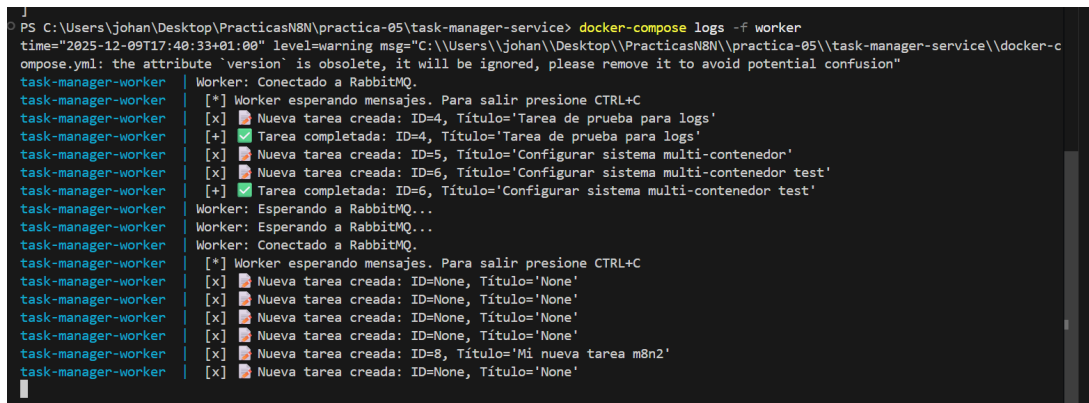


Figura 9: Logs del Worker procesando el mensaje enviado por n8n

3.5.3. Flujo 2B: El Consumidor (RabbitMQ → Procesamiento)

Este flujo escucha la cola `task_created` y se ejecuta automáticamente cuando llega un mensaje.

Estructura del flujo:

1. **RabbitMQ Trigger:** Escucha mensajes en la cola `task_created`.
2. **Edit Fields (Set):** Procesa el mensaje y añade metadatos.

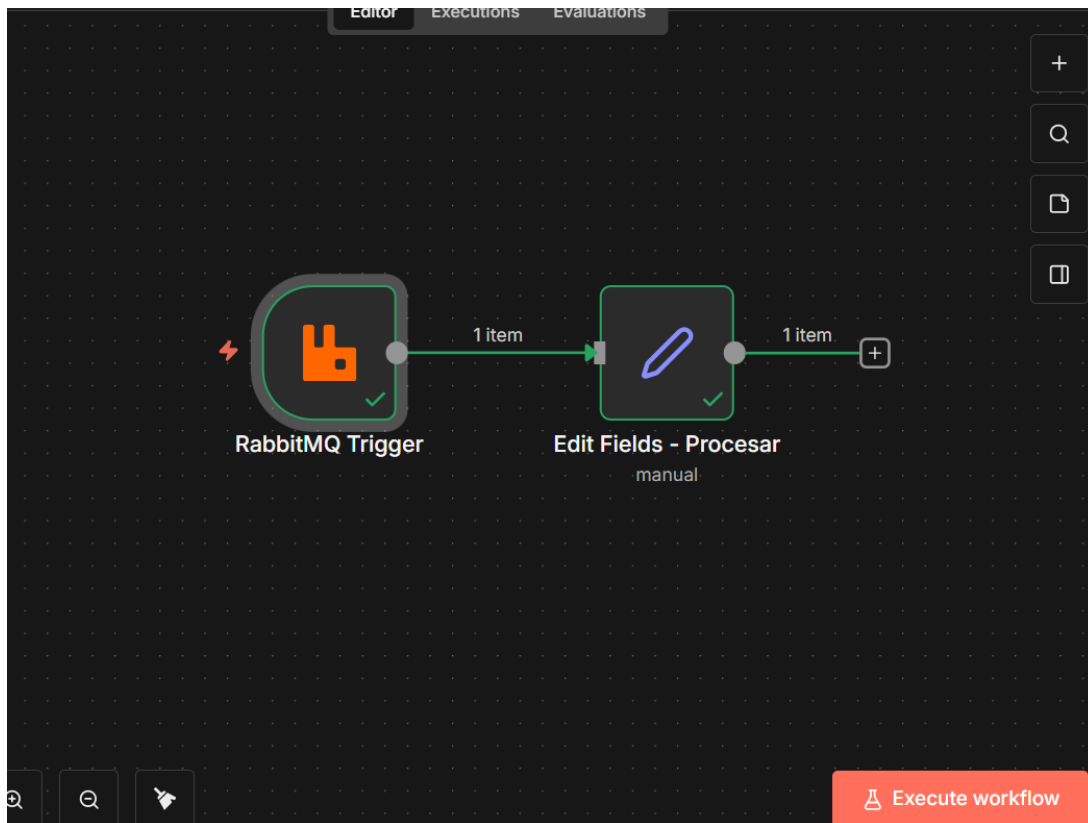


Figura 10: Flujo Consumidor - RabbitMQ Trigger que procesa mensajes

3.5.4. Ejecución del Consumidor

Cuando llega un mensaje a la cola, el flujo se ejecuta automáticamente:

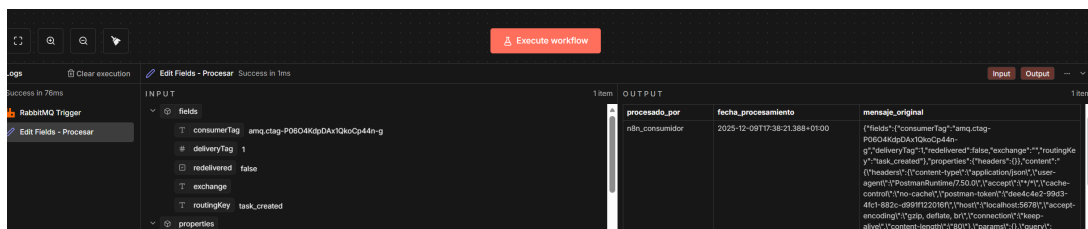


Figura 11: Ejecución del flujo consumidor mostrando el mensaje procesado

Nota sobre Competing Consumers: Cuando múltiples consumidores (Worker Python y n8n) escuchan la misma cola, compiten por los mensajes. Solo uno recibe cada mensaje.

3.6. Verificación en RabbitMQ Management

Se puede verificar el estado de las colas en la interfaz de administración de RabbitMQ:

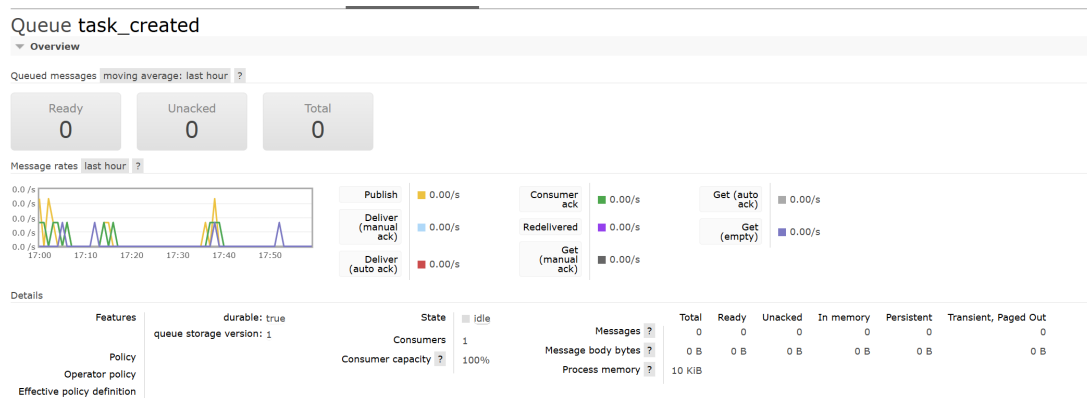


Figura 12: Colas de RabbitMQ mostrando mensajes procesados

Acceso a RabbitMQ Management:

- URL: `http://localhost:15672`
- Usuario: `guest`
- Contraseña: `guest`

4. Ejercicios Propuestos

Esta sección documenta la implementación de tres ejercicios adicionales que profundizan en las capacidades de orquestación de n8n, abordando operaciones CRUD directas, consumo de colas y reemplazo completo de endpoints API.

4.1. Ejercicio 1: Eliminación de Tareas

4.1.1. Objetivo y Requisitos

Objetivo: Implementar un flujo que elimine tareas directamente de PostgreSQL sin pasar por la API.

Dificultad: Baja

Requisitos:

- Iniciar el flujo con un Manual Trigger
- Definir el ID de la tarea a eliminar
- Ejecutar operación DELETE en PostgreSQL
- Verificar la eliminación exitosa

4.1.2. Implementación del Flujo

El flujo implementado consta de 4 nodos principales:

1. **Manual Trigger:** Inicia la ejecución manual del flujo.
2. **Edit Fields - ID a Borrar:** Define el ID de la tarea a eliminar (configurado con valor 8).
3. **PostgreSQL - Delete:** Ejecuta la operación DELETE sobre la tabla `tasks`.
4. **PostgreSQL - Verificar:** Realiza un SELECT para confirmar la eliminación.

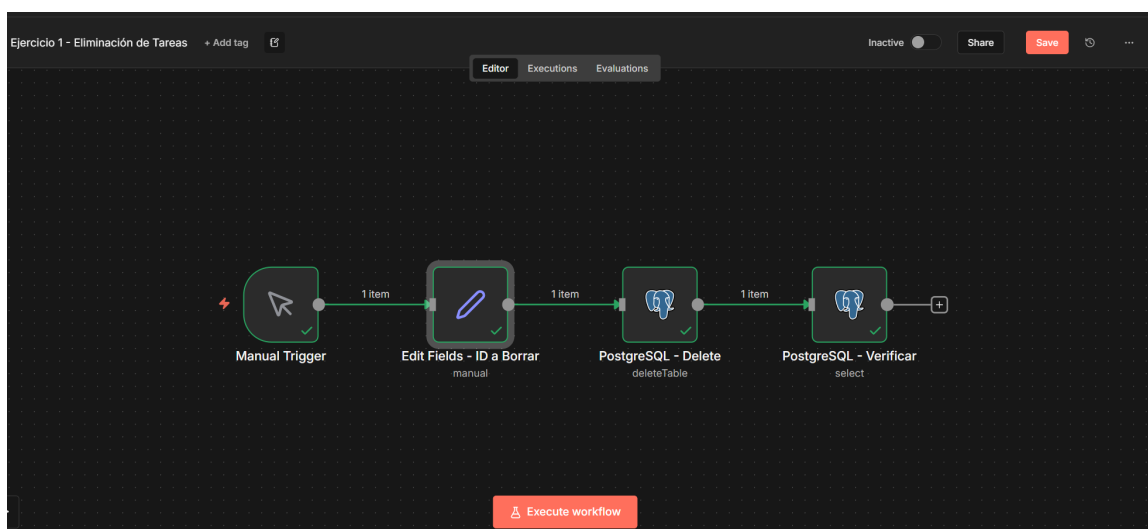


Figura 13: Flujo completo del Ejercicio 1 - Eliminación de Tareas

4.1.3. Configuración del Nodo PostgreSQL Delete

El nodo de eliminación se configuró con los siguientes parámetros:

Parámetro	Valor
Operation	Delete
Schema	public
Table	tasks
Delete Key	id
Delete Value	={{ \$json.id_a_borrar }}

Cuadro 3: Configuración del nodo PostgreSQL Delete

The screenshot shows the configuration interface for a 'PostgreSQL - Delete' node. The interface has a dark theme and includes a red 'Execute step' button in the top right corner. Below the title, there are two tabs: 'Parameters' (selected) and 'Settings'. A 'Docs' link with an external icon is also present. The configuration fields are as follows:

- Credential to connect with:** A dropdown menu showing 'Postgres account' with an edit icon.
- Operation:** A dropdown menu showing 'Delete'.
- Schema:** A section with a 'By Name' dropdown and a text field containing 'public'.
- Table:** A section with a 'raw' dropdown and a text field containing 'tasks'.
- Command:** A dropdown menu showing 'Truncate'.
- Restart Sequences:** A toggle switch that is currently turned off.
- Options:** A section titled 'No properties' with an 'Add option' button and a dropdown arrow.

Figura 14: Configuración del nodo PostgreSQL Delete

4.1.4. Ejecución y Resultados

Al ejecutar el flujo con `id_a_borrar = 8`, se observan los siguientes resultados:

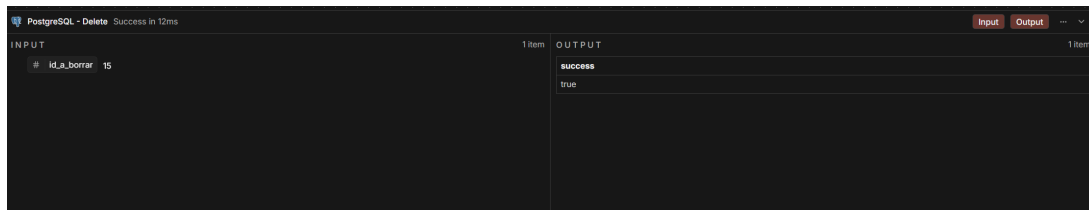


Figura 15: Ejecución exitosa mostrando la tarea eliminada

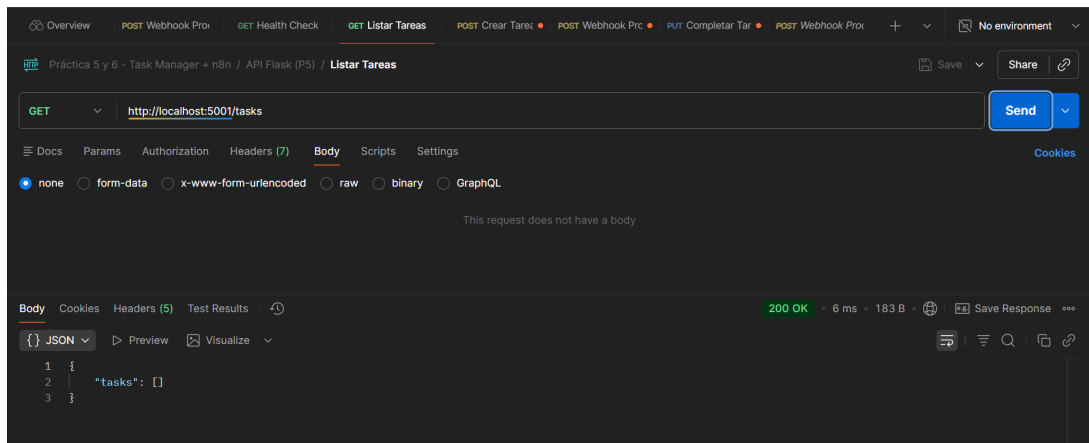


Figura 16: Verificación mostrando que la tarea ID=8 ya no existe en la base de datos

Lecciones Aprendidas:

- Las operaciones DELETE en n8n requieren especificar una clave y valor para identificar las filas.
- Es importante verificar las eliminaciones para confirmar la operación.
- El uso de expresiones permite parametrizar los valores dinámicamente.

4.2. Ejercicio 2: Servicio de Notificación

4.2.1. Objetivo y Requisitos

Objetivo: Implementar n8n como consumidor de la cola `task_completed` y enviar notificaciones por email cuando una tarea se complete.

Dificultad: Media

Requisitos:

- Crear un flujo que invoque el endpoint PUT `/tasks/<id>/complete`
- Implementar un trigger RabbitMQ para escuchar `task_completed`
- Enviar notificación por email con los detalles de la tarea
- Formatear el mensaje de forma profesional

4.2.2. Parte A: Completar Tarea vía API

Este flujo llama al endpoint de la API Flask para marcar una tarea como completada, lo que genera un mensaje en la cola `task_completed`.

Estructura del flujo:

1. **Manual Trigger:** Inicia la ejecución.
2. **Edit Fields - Task ID:** Define el ID de la tarea a completar (configurado con valor 15).
3. **HTTP Request - Complete Task:** Realiza PUT a `http://task-manager-web:5000/tasks/{{task_id}}/complete`.

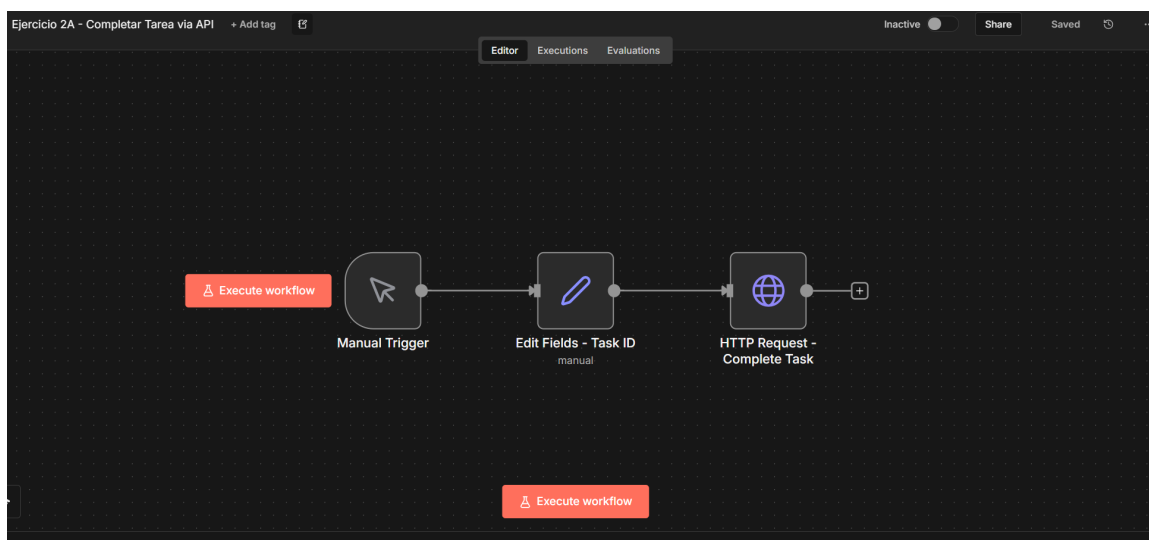


Figura 17: Flujo 2A - Completar tarea vía API

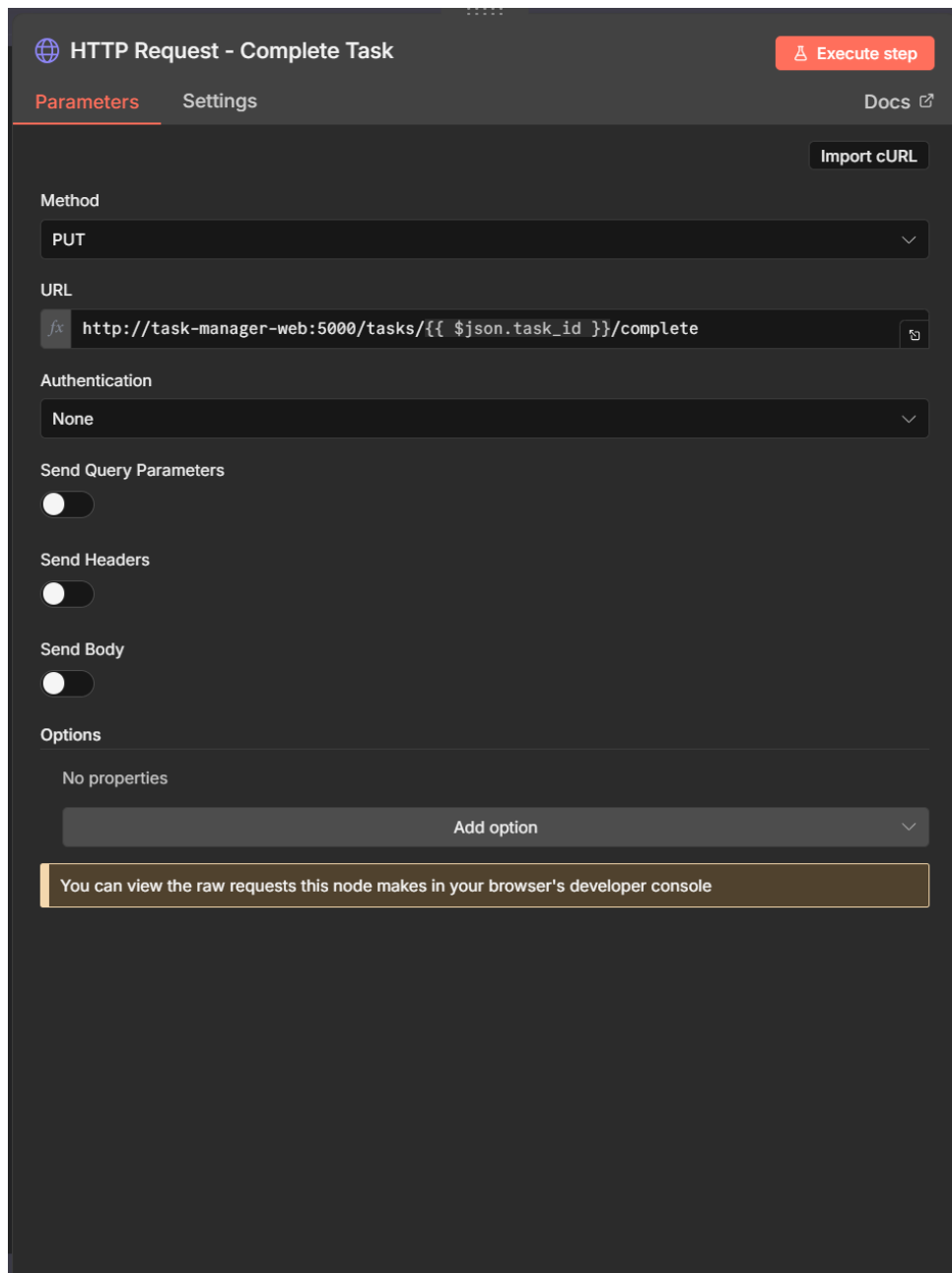


Figura 18: Configuración del nodo HTTP Request

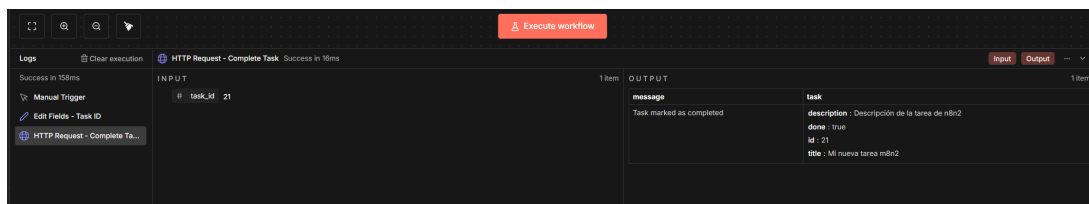


Figura 19: Respuesta exitosa del endpoint mostrando la tarea completada

4.2.3. Parte B: Notificador de Tareas Completadas

Este flujo escucha la cola `task_completed` y envía notificaciones por email cuando detecta una tarea completada.

Estructura del flujo:

1. **RabbitMQ Trigger - task_completed:** Escucha mensajes en la cola.
2. **Code - Parsear Content:** Procesa el mensaje JSON recibido.
3. **Edit Fields - Preparar Notificación:** Construye el asunto y cuerpo del email.
4. **Gmail - Enviar Notificación:** Envía el email de notificación.

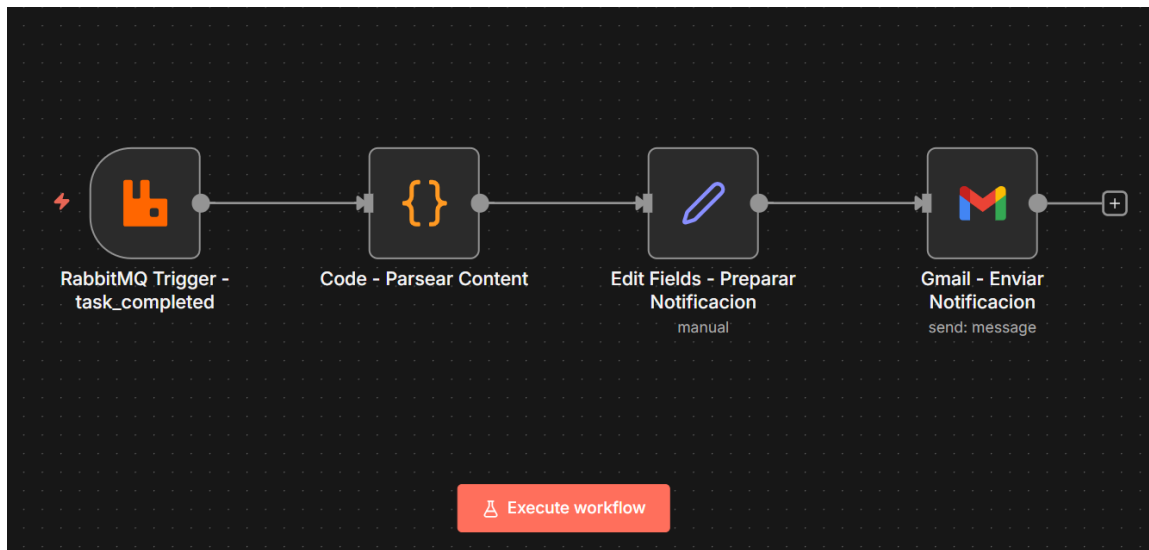


Figura 20: Flujo 2B - Notificador automático de tareas completadas

4.2.4. Configuración del Nodo RabbitMQ Trigger

Parámetro	Valor
Queue	task_completed
Content Is Binary	false
Credential	RabbitMQ (Tasks)

Cuadro 4: Configuración del RabbitMQ Trigger

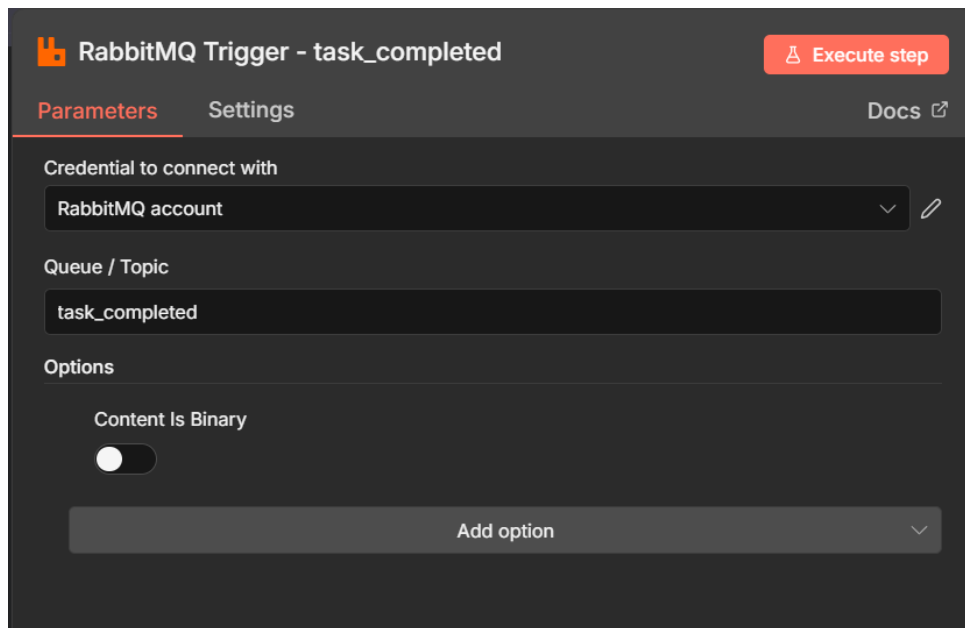


Figura 21: Configuración del trigger RabbitMQ

4.2.5. Procesamiento del Mensaje

El nodo Code parsea el contenido JSON recibido:

Listing 1: Código de parseo del mensaje RabbitMQ

```
// Parsear el content que viene como string JSON
const content = JSON.parse($input.first().json.content);

return {
  id: content.id,
  title: content.title,
  description: content.description,
  done: content.done
};
```

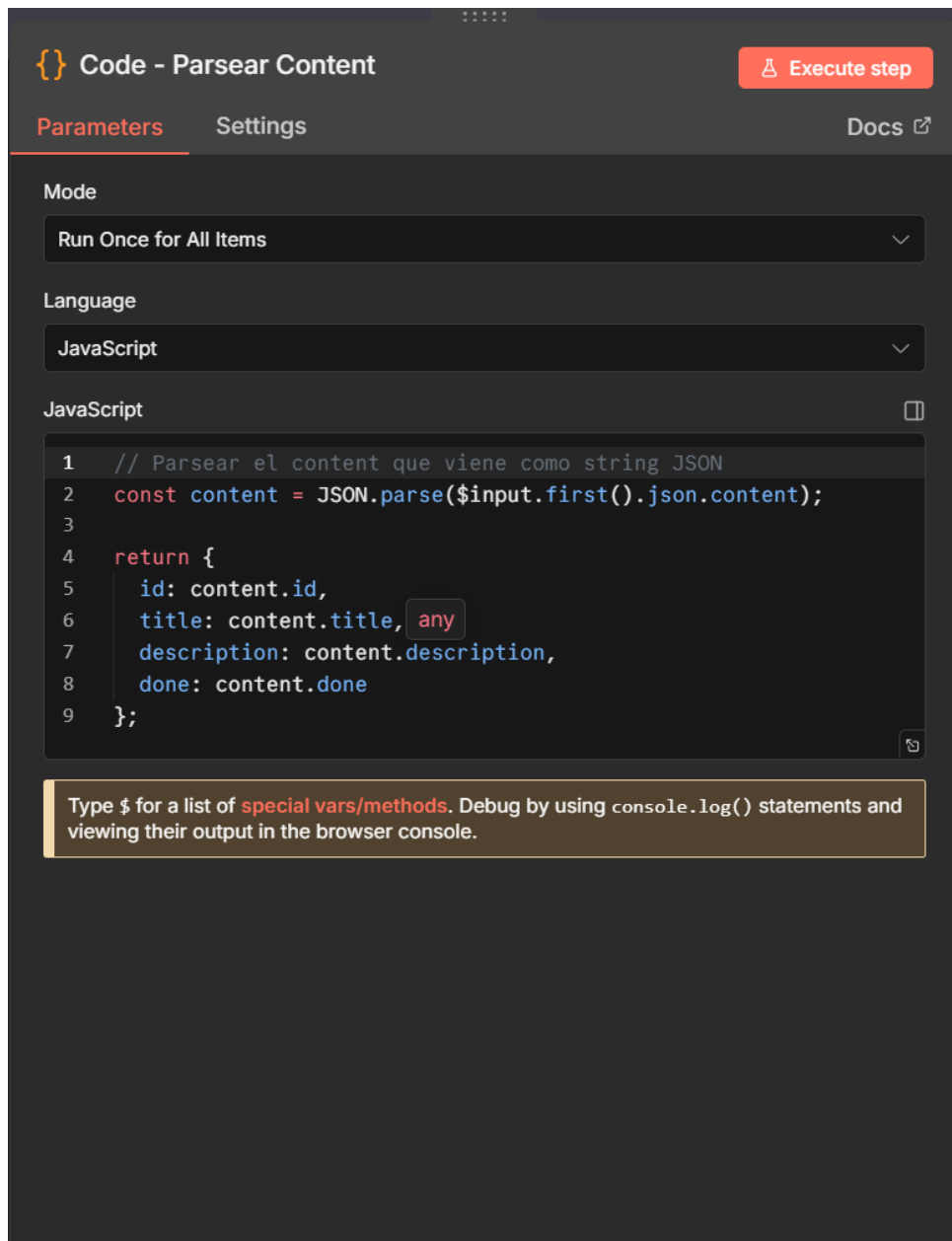


Figura 22: Nodo Code para parsear el mensaje

4.2.6. Preparación de la Notificación

El nodo Edit Fields construye el email con formato HTML:

Edit Fields - Preparar Notificación Execute step

Parameters Settings Docs

Mode: Manual Mapping

Fields to Set

Field	Type	Value
titulo_tarea	String	{{ \$json.title }}
id_tarea	Number	{{ \$json.id }}
descripcion	String	{{ \$json.description }}
fecha_notificacion	String	{{ \$now.format('dd/MM/yyyy HH:mm:ss') }}
email_subject	String	☑ Tarea Completada: {{ \$json.title }}
email_body	String	<h2>☑ Tarea Completada</h2><p>Título: {{ \$json.title }}</p><p>ID: {{ \$json.id }}</p><p>Descripción: {{ \$json.description }}</p><p>Fecha: {{ \$now.format('dd/MM/yyyy HH:mm:ss') }}</p><hr><p>Notificación enviada automáticamente por n8n</p>

Drag input fields here or Add Field

Include Other Input Fields ☐ Fixed Expression

Options

Figura 23: Configuración de campos para la notificación

Los campos generados incluyen:

- email_subject: " Tarea Completada: {{title}}"
- email_body: HTML formateado con detalles de la tarea
- fecha_notificacion: Timestamp con formato dd/MM/yyyy HH:mm:ss

4.2.7. Configuración de Gmail

The screenshot shows the configuration interface for a Gmail node in a workflow tool. The title is "Gmail - Enviar Notificacion". There are two tabs: "Parameters" (selected) and "Settings". A red button labeled "Execute step" is in the top right. Below the tabs, there are several configuration fields:

- Credential to connect with:** A dropdown menu showing "Gmail account".
- Resource:** A dropdown menu showing "Message".
- Operation:** A dropdown menu showing "Send".
- To:** A text field containing "johan.eduardo.cala2002@gmail.com".
- Subject:** A text field containing a JSON path expression: `{{ $json.email_subject }}`.
- Email Type:** A dropdown menu showing "HTML".
- Message:** A text field containing a JSON path expression: `{{ $json.email_body }}`.
- Options:** A section with the text "No properties" and a button labeled "Add option".

Figura 24: Configuración del nodo Gmail

4.2.8. Prueba End-to-End

Proceso completo:

1. Se ejecuta el Flujo 2A para completar la tarea ID=15
2. La API Flask publica un mensaje en `task_completed`
3. El Flujo 2B se activa automáticamente al detectar el mensaje
4. Se envía un email de notificación

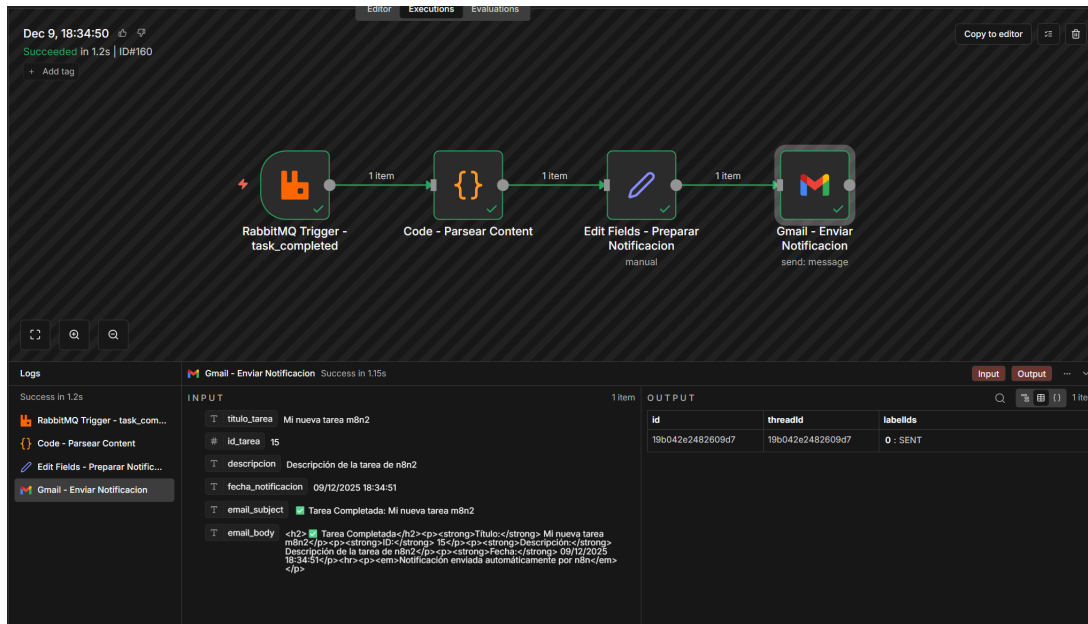


Figura 25: Ejecución automática del flujo al recibir mensaje

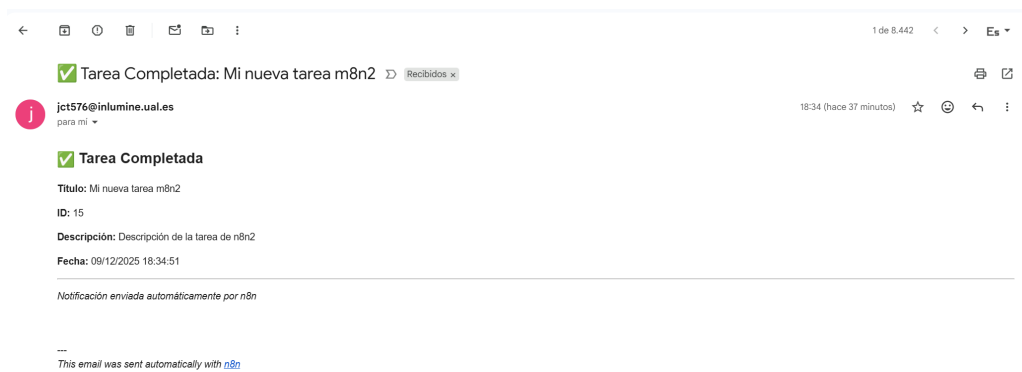


Figura 26: Email de notificación recibido en Gmail

Lecciones Aprendidas:

- Los RabbitMQ Triggers se ejecutan automáticamente sin intervención manual.
- Es necesario parsear el contenido JSON antes de procesarlo.
- Los mensajes HTML en Gmail permiten notificaciones más profesionales.
- La integración OAuth2 con Gmail requiere configuración previa en las credenciales.

4.3. Ejercicio 3: Reemplazo de API con Webhook

4.3.1. Objetivo y Requisitos

Objetivo: Reemplazar completamente el endpoint `POST /tasks` de Flask con un flujo de n8n que replique toda su funcionalidad.

Dificultad: Alta

Requisitos:

- Crear un webhook que reciba peticiones `POST` con `title` y `description`
- Validar que el campo `title` existe y no está vacío
- Insertar la tarea en PostgreSQL
- Publicar mensaje en la cola `task_created`
- Responder con código HTTP 201 Created
- Manejar errores con código HTTP 400 Bad Request

4.3.2. Arquitectura del Flujo

Este flujo implementa una API REST completa con validación, persistencia y mensajería asíncrona.

Estructura del flujo:

1. **Webhook - POST /api/tasks:** Recibe peticiones HTTP POST.
2. **IF - Validar Title:** Verifica que el campo `title` existe y no está vacío.
3. **PostgreSQL - Insert Task:** Inserta la tarea en la base de datos.
4. **RabbitMQ - Publicar task_created:** Envía mensaje a la cola.
5. **Respond - 201 Created:** Responde con éxito al cliente.
6. **Respond - 400 Error:** (Rama alternativa) Responde con error de validación.



Figura 27: Flujo completo del Ejercicio 3 - API Webhook para crear tareas

4.3.3. Configuración del Webhook

El webhook se configura para aceptar peticiones POST y procesar el body JSON:

Parámetro	Valor
HTTP Method	POST
Path	api/tasks
Response Mode	responseNode
Authentication	None

Cuadro 5: Configuración del Webhook Trigger

Webhook - POST /api/tasks

Listen for test event

Parameters

Settings

Docs

Webhook URLs

Test URL

Production URL

POST

http://localhost:5678/webhook/api/tasks

HTTP Method

POST

Path

api/tasks

Authentication

None

Respond

Using 'Respond to Webhook' Node

Insert a 'Respond to Webhook' node to control when and how you respond. More details

Options

No properties

Add option

Figura 28: Configuración del Webhook Trigger

URL del Webhook:

```
http://localhost:5678/webhook/api/tasks
```

4.3.4. Validación de Entrada

El nodo IF implementa dos condiciones que deben cumplirse simultáneamente:

1. **Check Title Exists:** Verifica que `$json.body.title` existe
2. **Check Title Not Empty:** Verifica que `$json.body.title` no está vacío

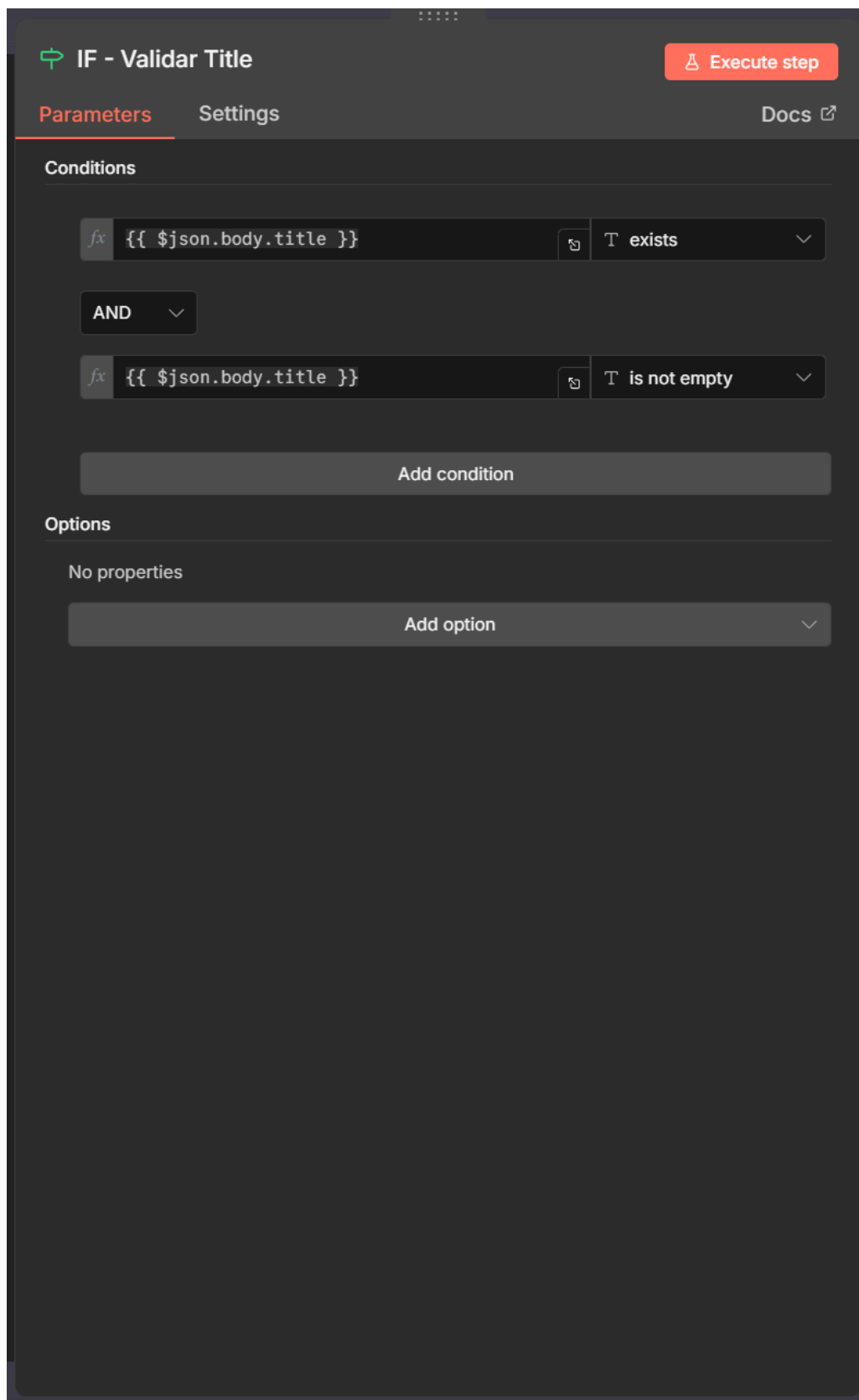


Figura 29: Configuración de las condiciones de validación

Configuración de condiciones:

- **Combinator:** AND (ambas condiciones deben cumplirse)

- **Case Sensitive:** true
- **Type Validation:** strict

4.3.5. Inserción en PostgreSQL

El nodo PostgreSQL Insert mapea los campos del body JSON a las columnas de la tabla:

Columna	Expresión
title	={{ \$json.body.title }}
description	={{ \$json.body.description '' }}
done	false

Cuadro 6: Mapeo de campos para la inserción

PostgreSQL - Insert Task

Execute step

Parameters

Settings

Docs

Credential to connect with

Postgres account

Operation

Insert

Schema

From list

public

Table

From list

tasks

Mapping Column Mode

Map Each Column Manually

Values to Send

title

fx

{{ \$json.body.title }}

description

fx

{{ \$json.body.description || '' }}

done

☐

Options

No properties

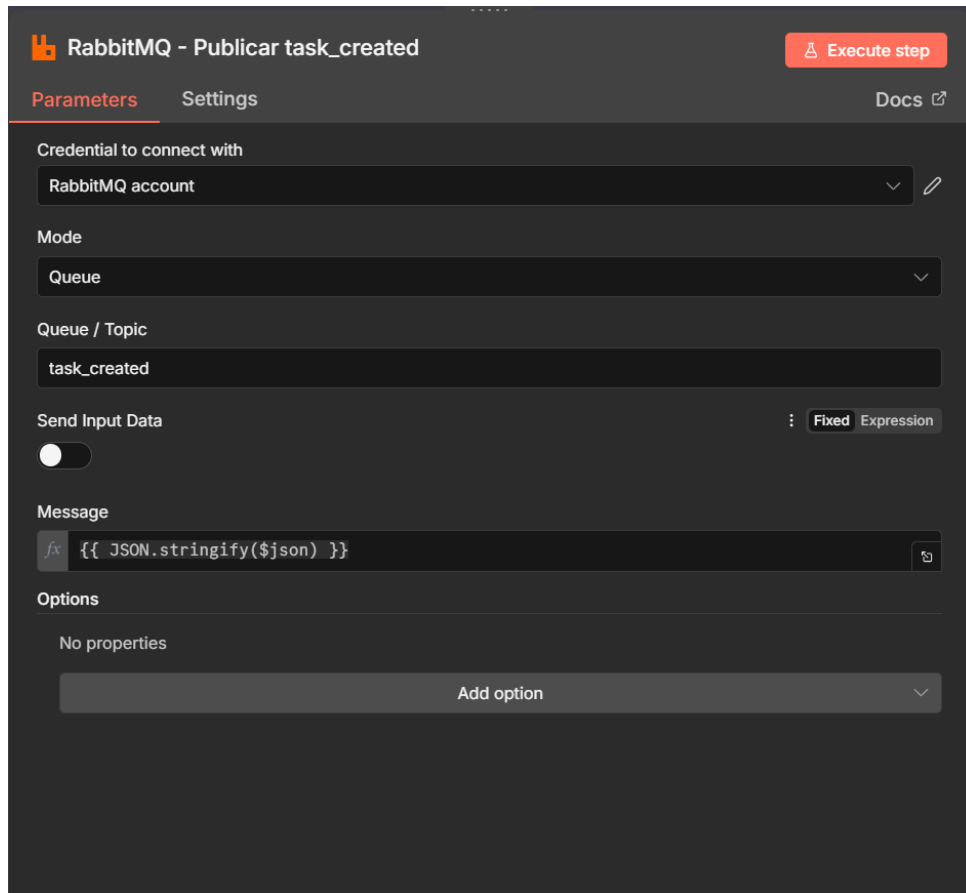
Add option

Figura 30: Configuración del nodo PostgreSQL Insert

Nota: El operador `||` proporciona un valor por defecto (string vacío) si `description` no está presente.

4.3.6. Publicación en RabbitMQ

El nodo RabbitMQ envía los datos de la tarea recién creada a la cola `task_created`:



The screenshot shows the configuration for a 'RabbitMQ - Publicar task_created' step. It includes tabs for 'Parameters' and 'Settings', and a 'Docs' link. The 'Parameters' tab is active, showing fields for 'Credential to connect with' (set to 'RabbitMQ account'), 'Mode' (set to 'Queue'), and 'Queue / Topic' (set to 'task_created'). There is a 'Send Input Data' toggle switch and a 'Message' field containing the expression `{{ JSON.stringify($json) }}`. The 'Options' section shows 'No properties' and an 'Add option' button.

Figura 31: Configuración del nodo RabbitMQ Send

Configuración:

- **Queue:** `task_created`
- **Send Input Data:** `false`
- **Message:** `{{ JSON.stringify($json) }}`

4.3.7. Respuestas HTTP

Respuesta exitosa (201 Created):

```
1 {  
2   "message": "Task created successfully",  
3   "task": { /* datos de la tarea */ },  
4   "code": 201  
5 }
```

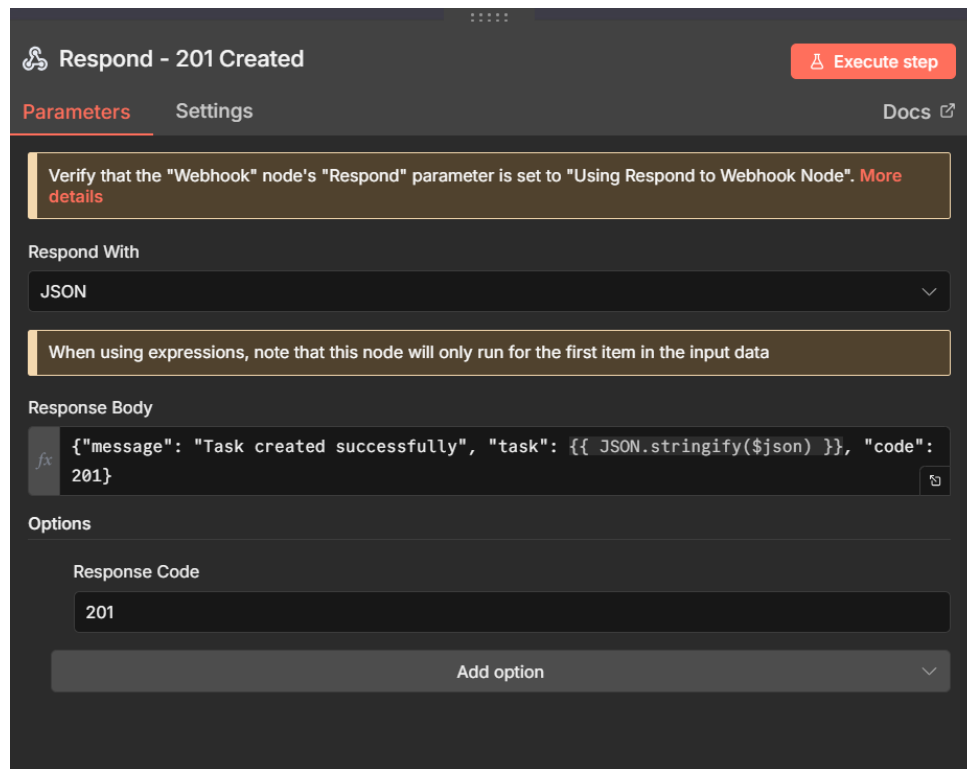



Figura 32: Configuración de la respuesta 201 Created

Respuesta de error (400 Bad Request):

```
1 {  
2   "error": "Bad Request",  
3   "message": "El campo 'title' es requerido",  
4   "code": 400  
5 }
```

 Respond - 400 Error

Execute step

Parameters

Settings

Docs

Verify that the "Webhook" node's "Respond" parameter is set to "Using Respond to Webhook Node". [More details](#)

Respond With

JSON

When using expressions, note that this node will only run for the first item in the input data

Response Body

fx

`{"error": "Bad Request", "message": "El campo 'title' es requerido", "code": 400}`

`{"error": "Bad Request", "message": "El campo 'title' es requerido", "code": 400}`

Options

Response Code

400

Add option

Figura 33: Configuración de la respuesta 400 Bad Request

4.3.8. Pruebas del Endpoint

Prueba 1: Creación exitosa

```
curl -X POST http://localhost:5678/webhook/api/tasks \
-H "Content-Type: application/json" \
-d '{
  "title": "Nueva tarea desde webhook",
  "description": "Prueba del ejercicio 3"
}'
```

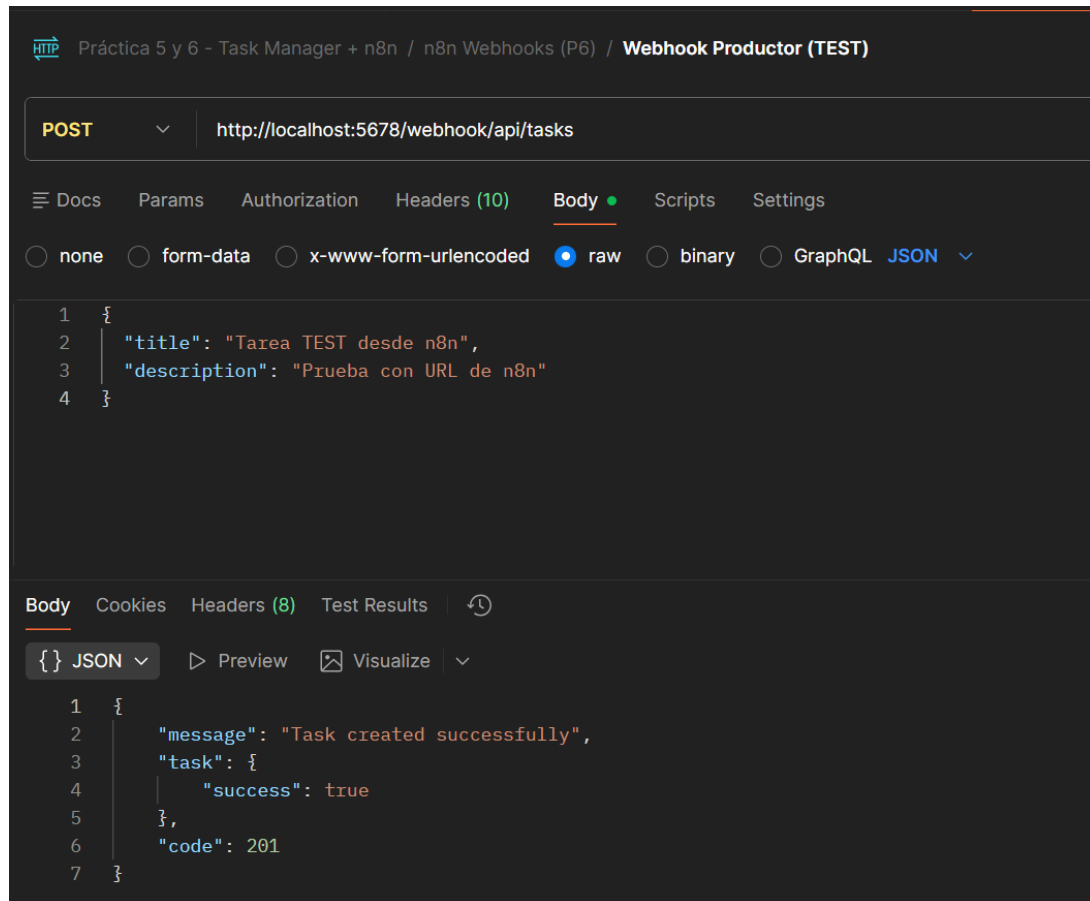


Figura 34: Prueba exitosa mostrando respuesta 201 Created

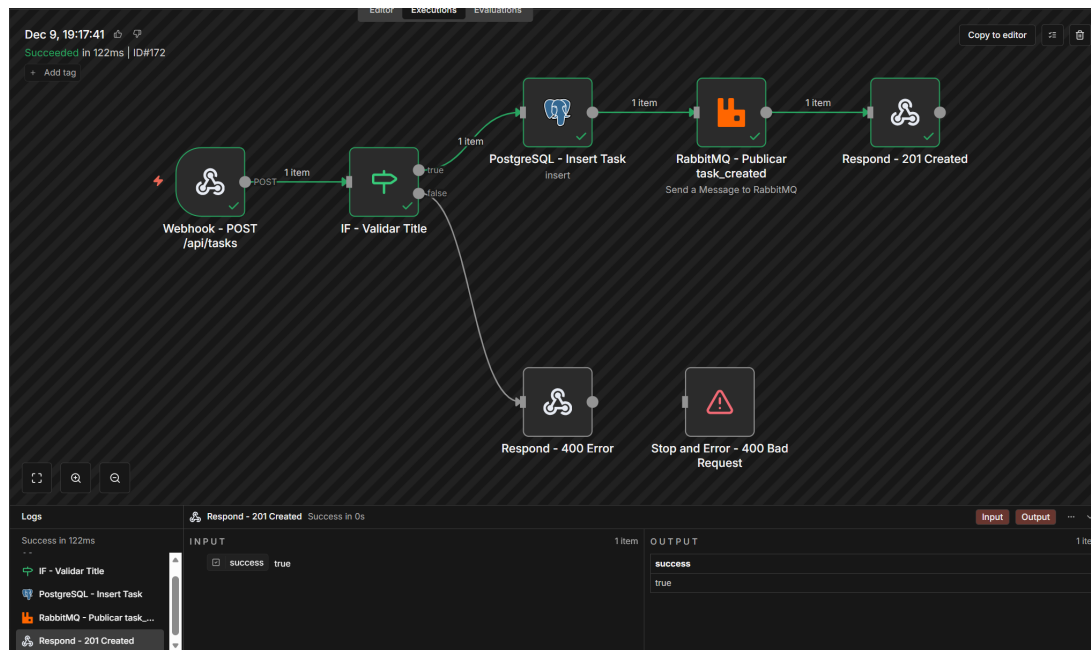


Figura 35: Ejecución del flujo mostrando todos los nodos en verde

Prueba 2: Validación de error (sin title)

```
curl -X POST http://localhost:5678/webhook/api/tasks \
-H "Content-Type: application/json" \
-d '{
  "description": "Tarea sin titulo"
}'
```

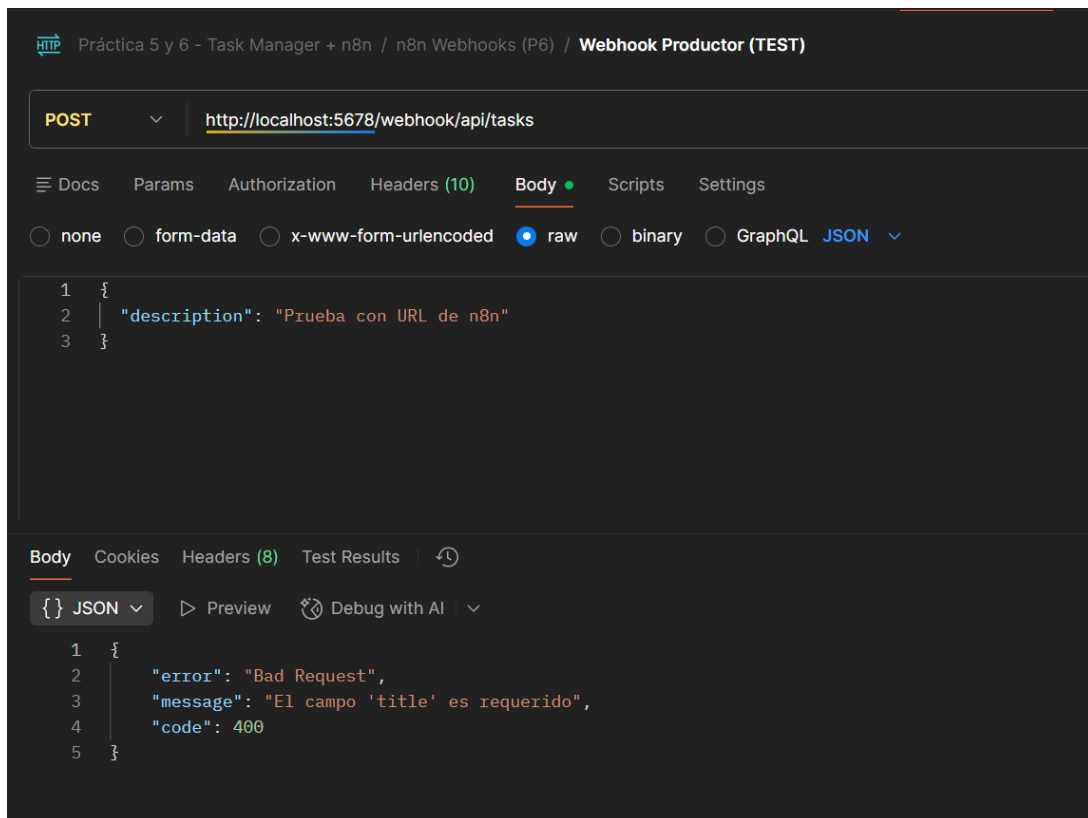


Figura 36: Prueba de error mostrando respuesta 400 Bad Request

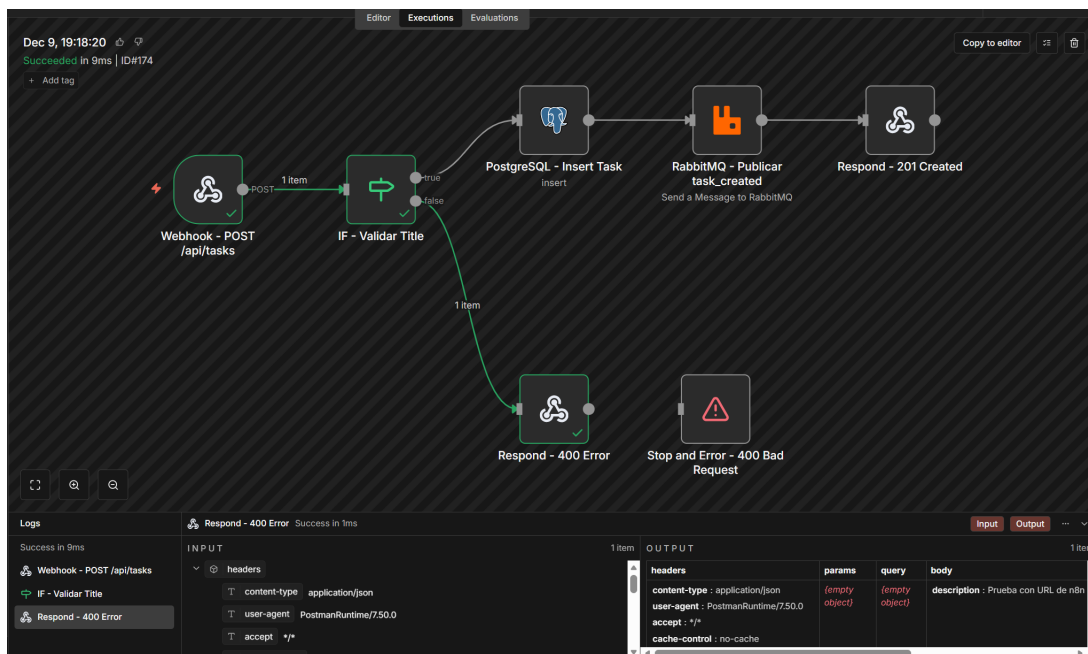


Figura 37: Ejecución del flujo mostrando la rama de error

4.3.9. Comparación con API Flask

Aspecto	API Flask	Webhook n8n
Código	Requiere Python y Flask	Sin código, configuración visual
Validación	Lógica en Python	Nodos IF y condiciones
Base de datos	SQLAlchemy ORM	Nodo PostgreSQL nativo
Mensajería	Librería pika	Nodo RabbitMQ nativo
Respuestas HTTP	jsonify() y códigos	Nodos Respond to Webhook
Mantenimiento	Requiere desarrollo	Modificación visual
Escalabilidad	Servidor web + workers	n8n maneja concurrencia

Cuadro 7: Comparación entre API Flask y Webhook n8n

Lecciones Aprendidas:

- n8n puede reemplazar completamente endpoints API simples a medianos.
- La validación visual es más clara que la validación en código.
- El modo `responseNode` permite control total sobre las respuestas HTTP.
- Los webhooks de n8n pueden integrarse perfectamente con microservicios existentes.
- La expresión `JSON.stringify($json)` es necesaria para serializar objetos a RabbitMQ.

5. Conclusiones

En esta práctica se han alcanzado los siguientes objetivos:

1. **Integración con PostgreSQL:** n8n puede leer y escribir directamente en la base de datos sin depender de la API Flask.
2. **Producción de mensajes:** Se implementó un flujo que recibe webhooks y publica mensajes en RabbitMQ.
3. **Consumo de mensajes:** Se creó un flujo reactivo que se ejecuta automáticamente al recibir mensajes.
4. **Conectividad Docker:** Se resolvió el desafío de comunicación entre contenedores de diferentes redes.
5. **Orquestación:** n8n actúa como el “pegamento” que coordina los microservicios.

5.1. Lecciones Aprendidas

- **Redes Docker:** Los contenedores deben estar en la misma red para comunicarse por nombre.
- **Credenciales:** Usar nombres de servicio (`db`, `mq`) en lugar de `localhost`.
- **Webhooks:** Distinguir entre URLs de test y producción en n8n.
- **Competing Consumers:** Múltiples consumidores de la misma cola compiten por mensajes.
- **RabbitMQ Trigger:** Se activa automáticamente, no requiere ejecución manual.

5.2. Relación con Práctica 5

Esta práctica extiende el sistema de la Práctica 5 al integrar n8n como orquestador:

Componente	Práctica 5	Práctica 6
API Flask	Creación	Uso opcional
PostgreSQL	Almacenamiento	Acceso directo desde n8n
RabbitMQ	Mensajería entre servicios	n8n como productor/consumidor
Worker Python	Procesamiento	Compite con n8n por mensajes
n8n	No usado	Orquestador central

Cuadro 8: Evolución del sistema entre prácticas