Práctica 6:

Orquestación de Microservicios - Integración con PostgreSQL y RabbitMQ

INTEGRACIÓN DE TECNOLOGÍAS Y SERVICIOS INFORMÁTICOS

Daniel Salas Alonso

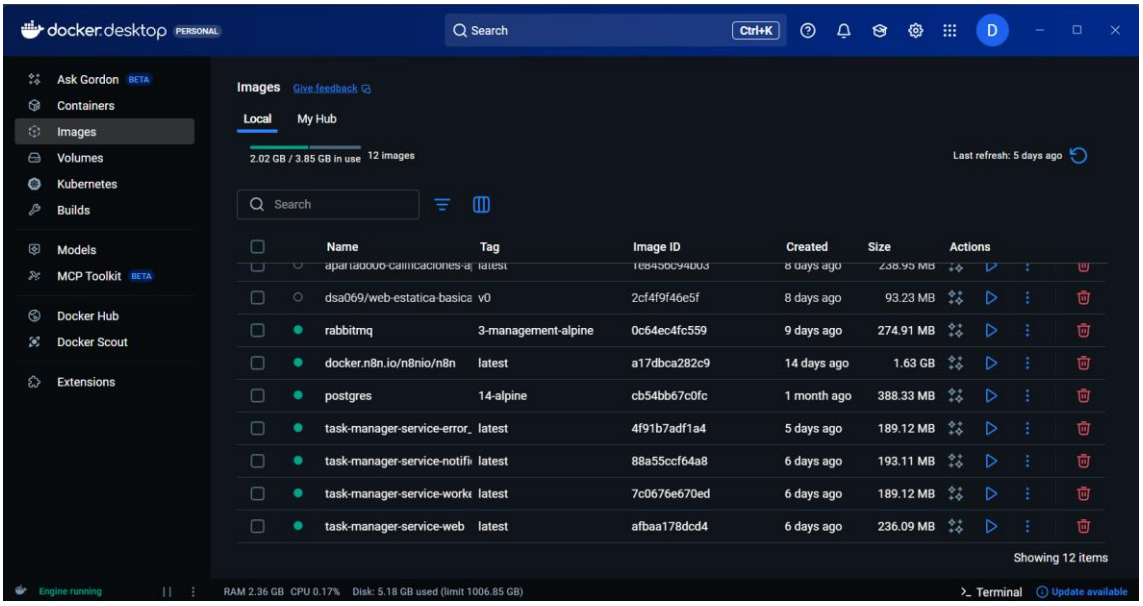
3. Desarrollo del Flujo de Trabajo Guiado

3.1. Paso 1: Preparar el Entorno de Microservicios

Comenzamos preparando el entorno desplegando los microservicios necesarios. Ejecutamos el comando docker-compose up -d --build en la terminal. Observamos cómo el sistema construye e inicia los contenedores definidos (task-manager-service-web, worker, error_handler, notifier, db, mq), asegurando que toda la infraestructura esté operativa.

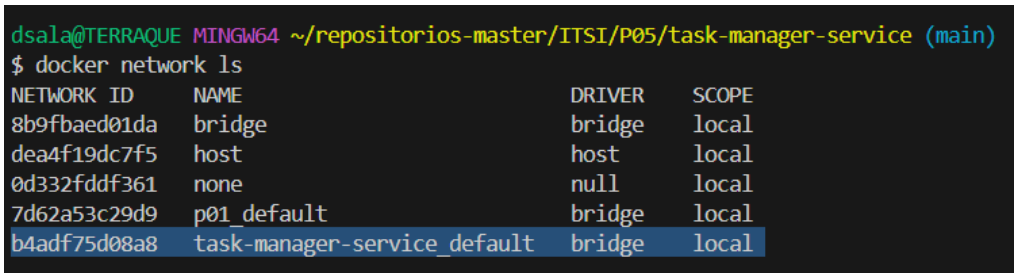
```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P05/task-manager-service (main)
$ docker-compose up -d --build
 => [web] resolving provenance for metadata file                              0.1s
 => [worker] resolving provenance for metadata file                           0.0s
 => [error_handler] resolving provenance for metadata file                    0.0s
[+] Running 11/11
 ✓ task-manager-service-web                 Built                             0.0s
 ✓ task-manager-service-web                 Built                             0.0s
 ✓ task-manager-service-worker              Built                             0.0s
 ✓ task-manager-service-error_handler       Built                             0.0s
 ✓ task-manager-service-notifier            Built                             0.0s
 ✓ Network task-manager-service_default     Created                           0.1s
 ✓ Container task-manager-db                Started                           0.6s
 ✓ Container task-manager-mq                Started                           0.6s
 ✓ Container task-manager-worker            Started                           0.9s
 ✓ Container task-manager-error-handler     Started                           0.8s
 ✓ Container task-manager-notifier          Started                           0.9s
 ✓ Container task-manager-web               Started                           0.9s
```

Verificamos en Docker Desktop que las imágenes y contenedores anteriores has sido iniciados correctamente junto con el contenedor de n8n.

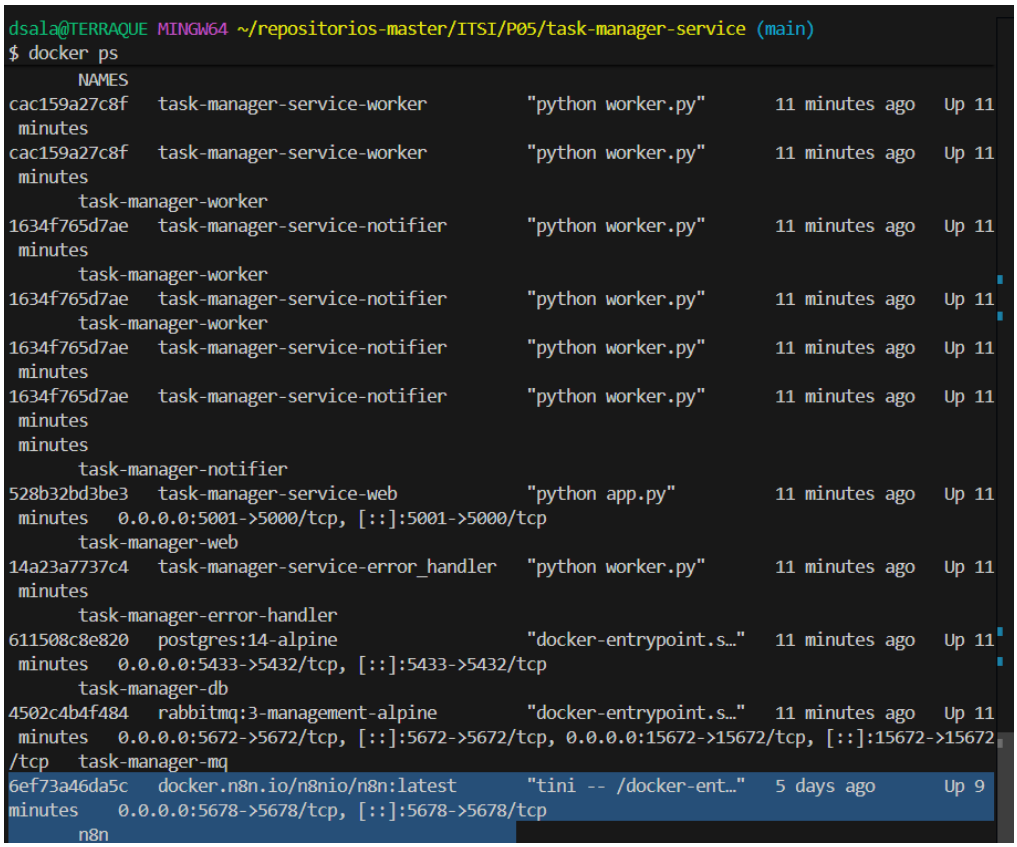


3.2. Paso 2: Conectar n8n a la Red de Microservicios

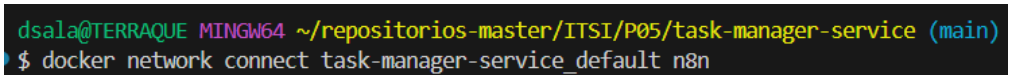
A continuación, identificamos la red creada por Docker Compose (task-manager-service_default) mediante el comando docker network ls.



También verificamos con docker ps el nombre de nuestro contenedor n8n, el cual es “n8n”.

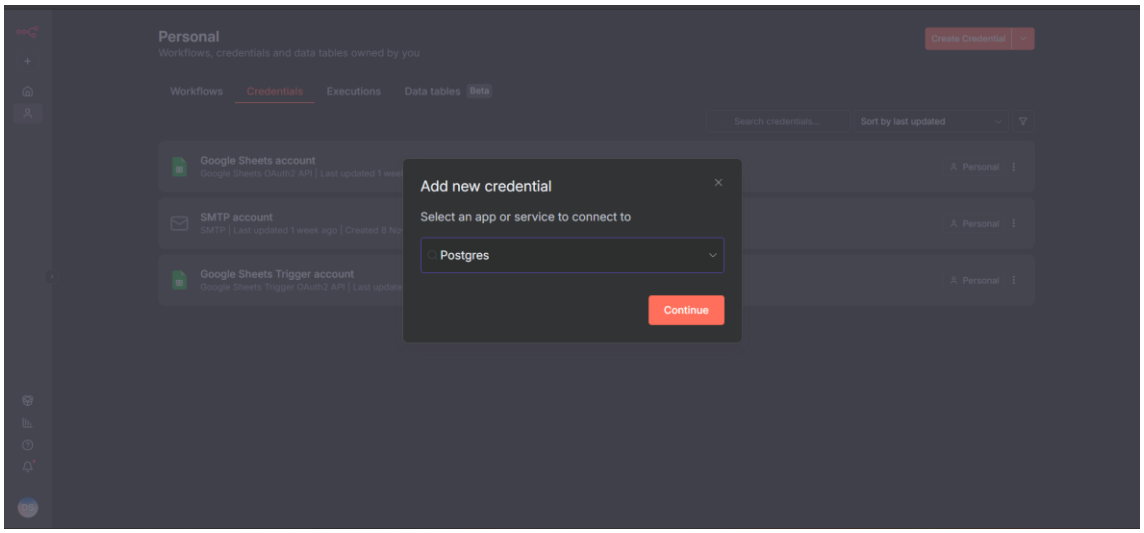


Para permitir la comunicación entre n8n y el resto de servicios, ejecutamos el siguiente comando. Esto conecta el contenedor de n8n a la misma red que la base de datos y el gestor de colas, permitiendo el acceso mediante los nombres de host del servicio (db, mq).



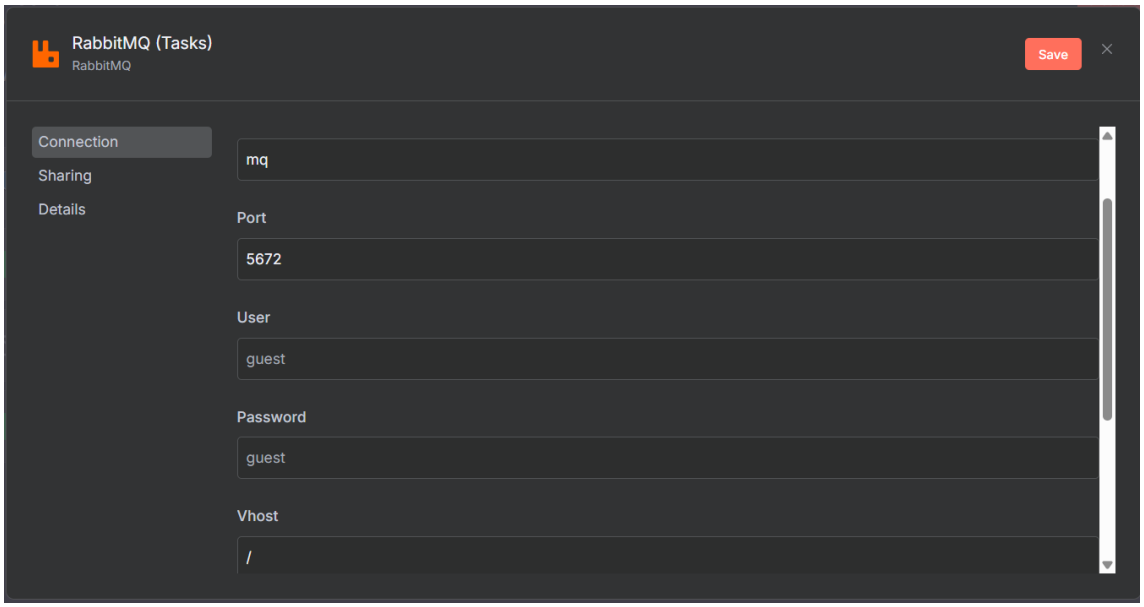
3.3. Paso 3: Configurar las Credenciales en n8n

Accedemos al panel de credenciales en n8n y seleccionamos "Create Credential". Buscamos y seleccionamos "Postgres" para configurar el acceso a la base de datos de tareas.

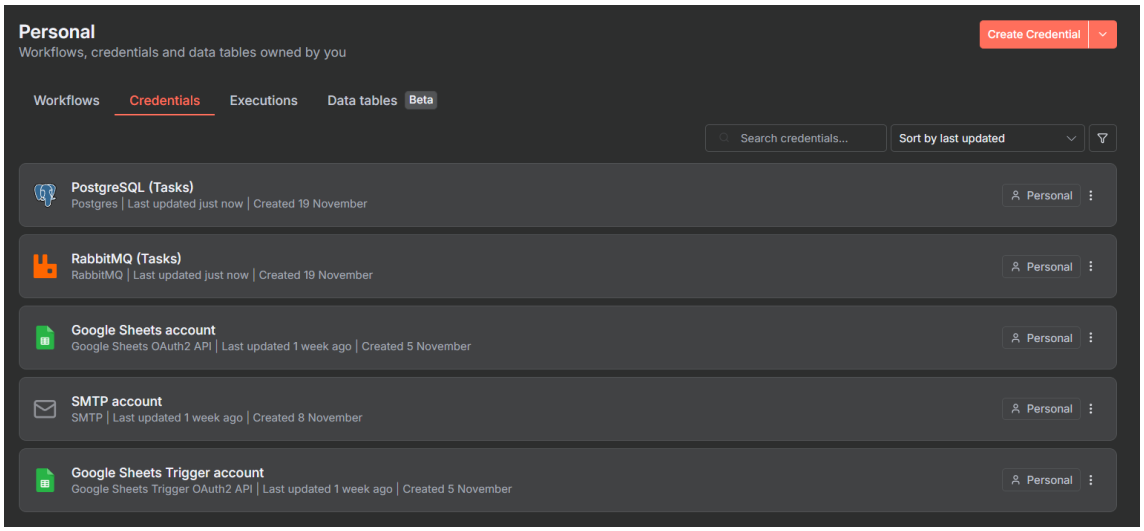


En la configuración de la credencial PostgreSQL, definimos los parámetros de conexión. Establecemos el Host como db (el nombre del servicio en Docker), la Database como taskdb, el User como user y la contraseña correspondiente. Guardamos la credencial para usarla en los nodos posteriores.

Repetimos el proceso para RabbitMQ. Creamos una nueva credencial y configuramos el Host como mq, el Port 5672, y las credenciales de usuario (guest/guest). Esto permitirá a n8n publicar y consumir mensajes de las colas de nuestro proyecto.

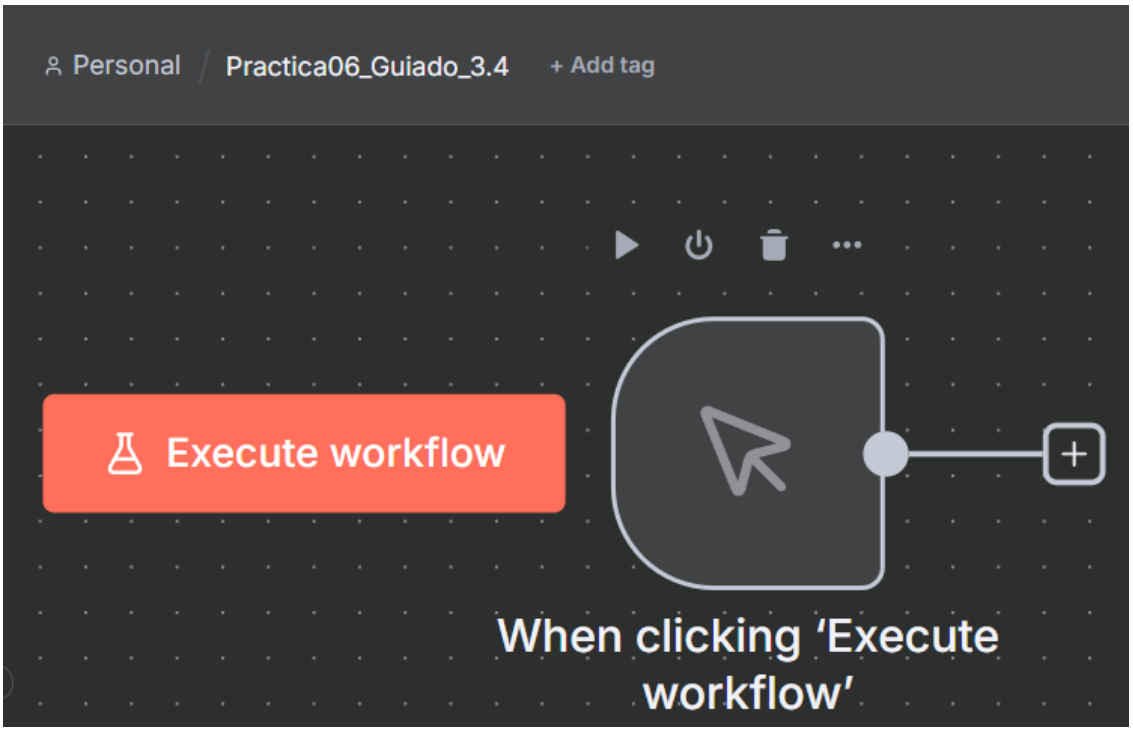


Observamos el panel de credenciales con "PostgreSQL (Tasks)" y "RabbitMQ (Tasks)" creadas y listas para ser utilizadas en nuestros workflows.

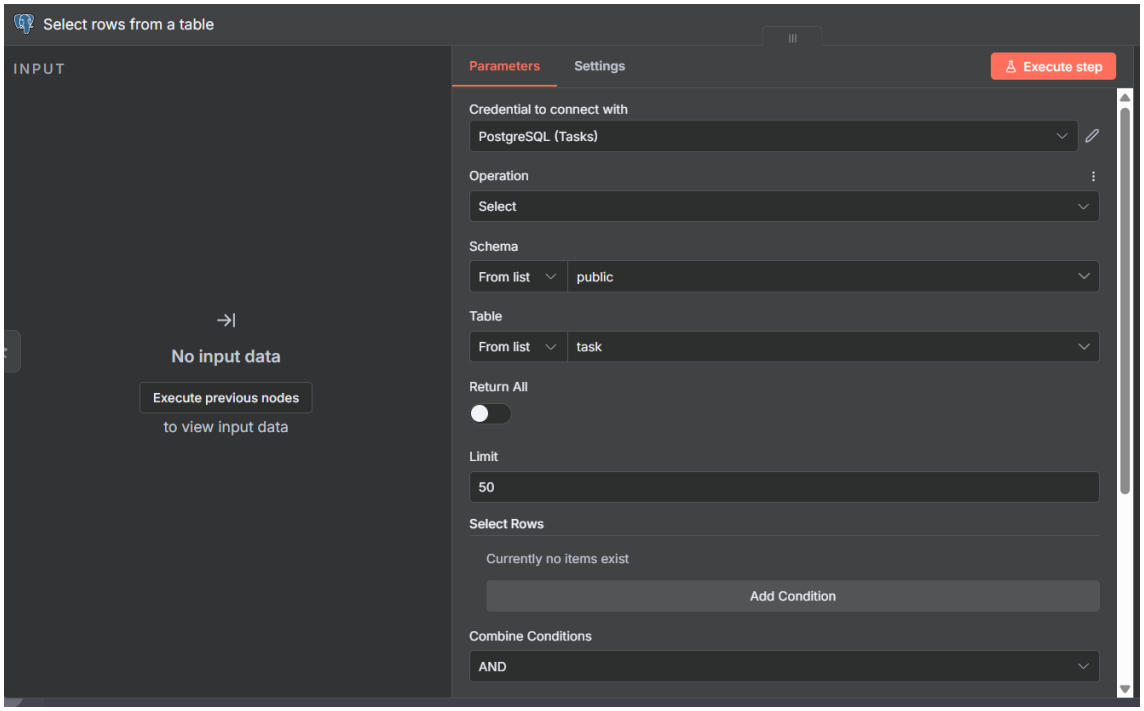


3.4. Paso 4: Flujo 1 - Interacción Directa con la Base de Datos

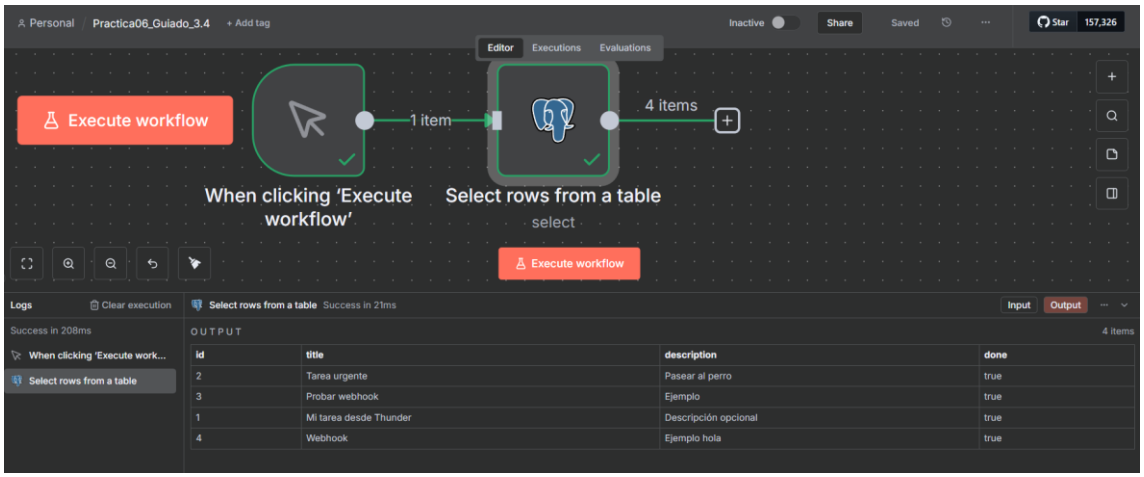
Creamos un nuevo workflow (Practica06_Guiado_3.4) y añadimos el nodo inicial "Manual Trigger" para poder disparar el flujo manualmente durante las pruebas.



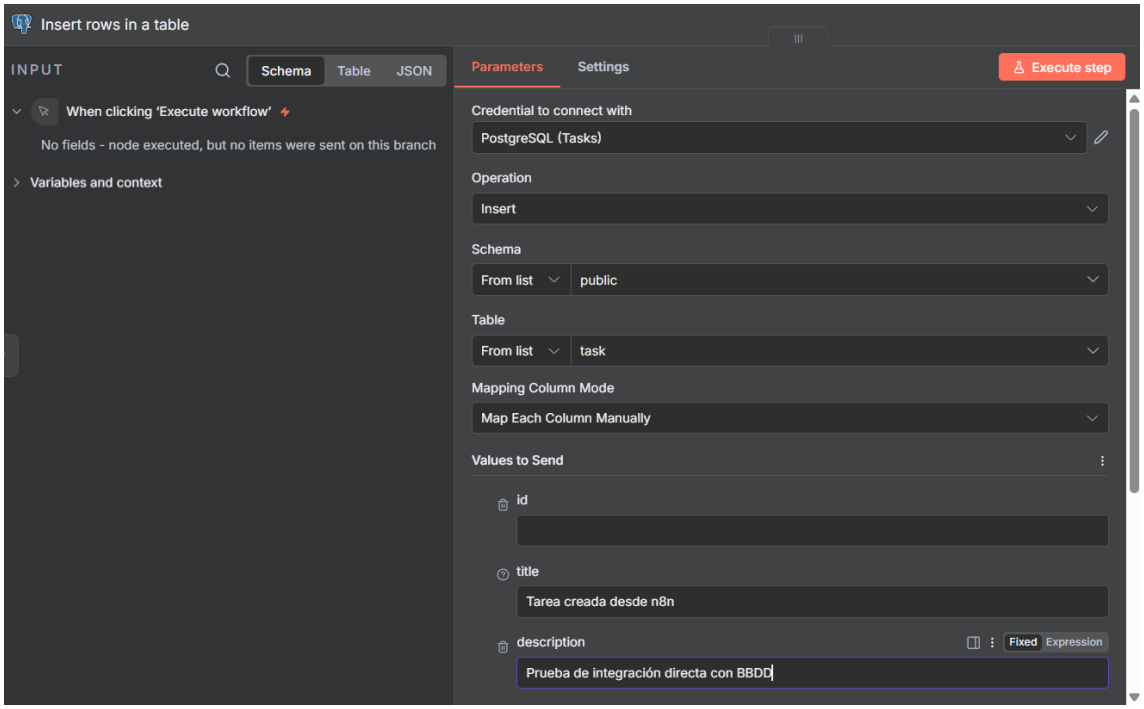
A continuación, conectamos un nodo PostgreSQL configurado con la operación "Select". Seleccionamos la credencial creada anteriormente, el esquema public y la tabla task. Esto se hace para obtener todas las entradas de nuestra tabla tareas.



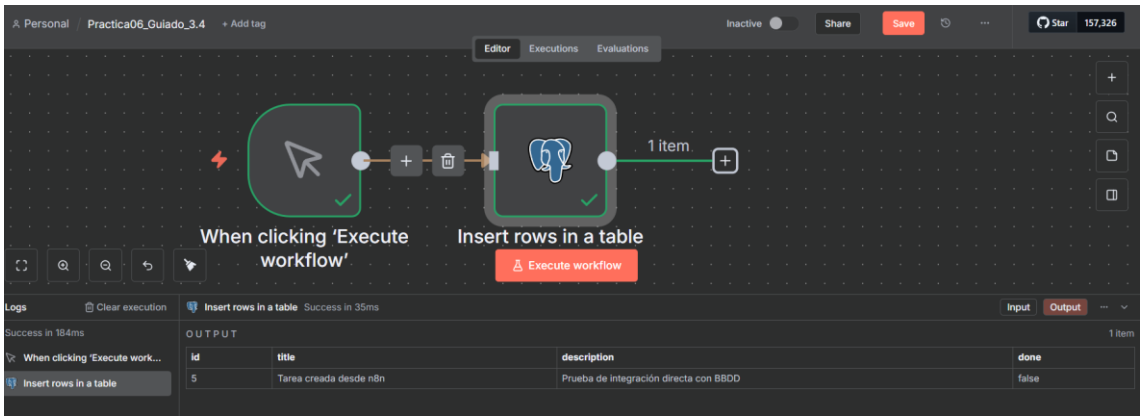
Ejecutamos el flujo y observamos la salida del nodo PostgreSQL. Vemos que nos devuelve un array de tareas con los campos id, title, description y done, confirmando la conexión exitosa y la recuperación de datos.



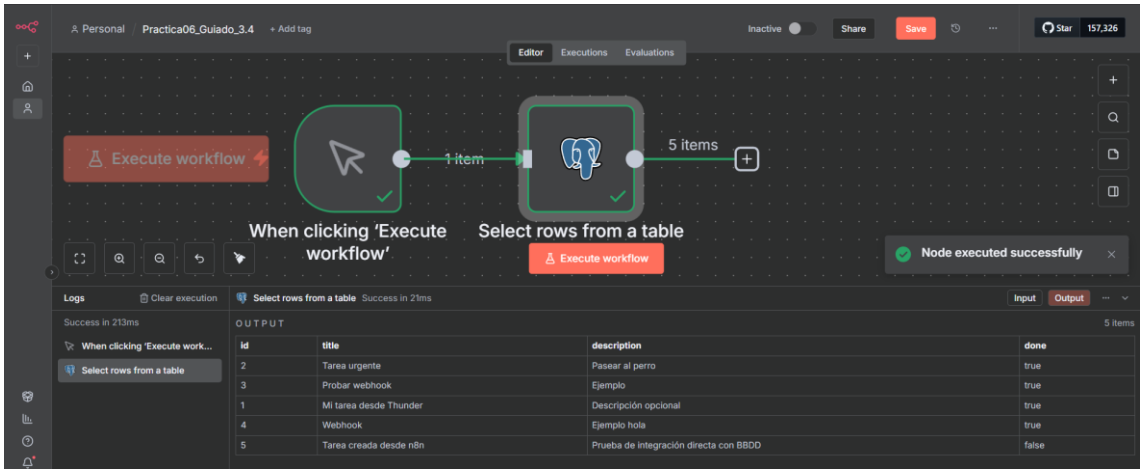
Para insertar nuevos datos, añadimos otro nodo PostgreSQL, esta vez con la operación "Insert". Asignamos un valor fijo a title ("Tarea creada desde n8n") y a description ("Prueba de integración directa con BBDD").



Vemos el flujo de trabajo completo para la inserción. Al ejecutar el nodo de inserción, observamos en la salida el nuevo registro creado con su id generado automáticamente (ej. id: 5), confirmando que la escritura en la base de datos funciona correctamente.



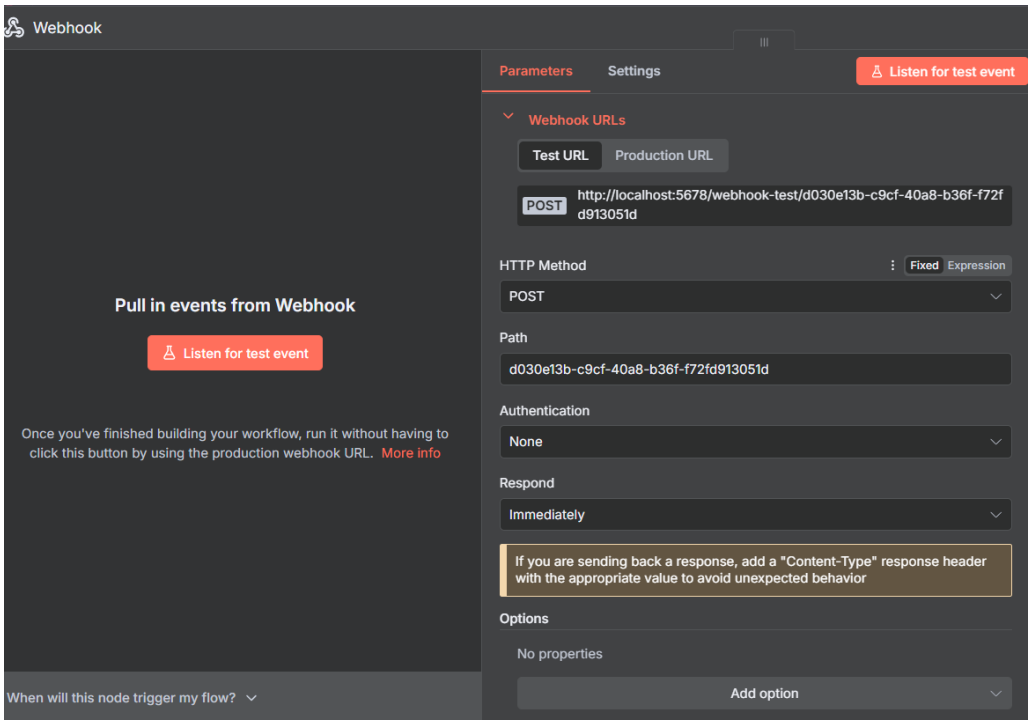
Volvemos a modificar el nodo para que sea de tipo “Select” y ejecutamos nuevamente el nodo. Observamos que la nueva tarea (ID 5) aparece ahora en la lista de resultados junto con las tareas preexistentes.



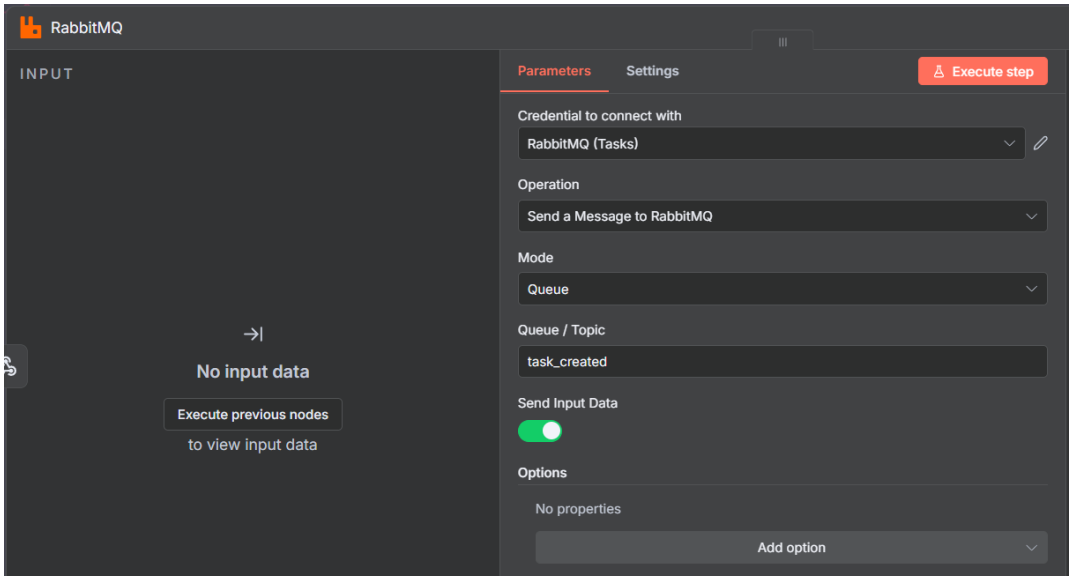
3.5. Paso 5: Flujo 2 - n8n como Productor y Consumidor (Asíncrono)

1. *Flujo A: El Productor (Webhook → RabbitMQ)*

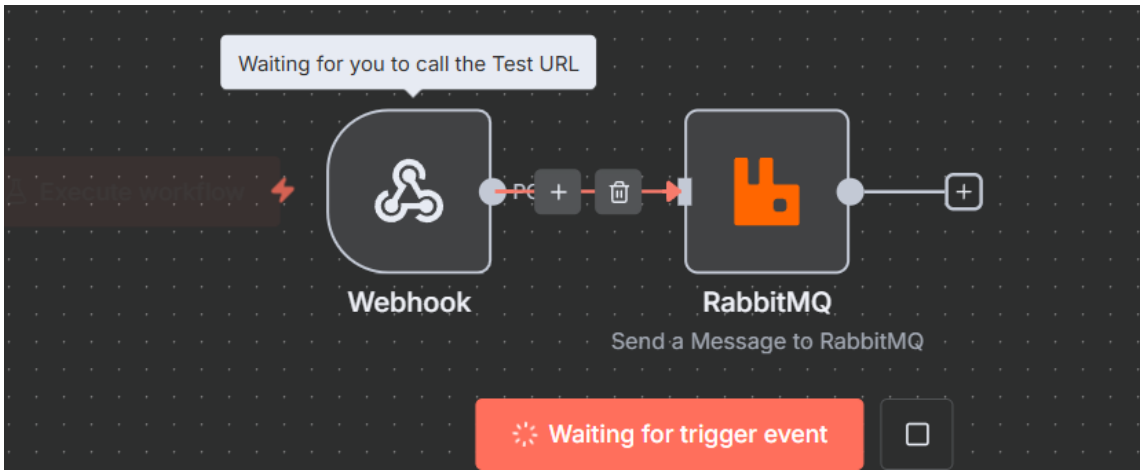
Comenzamos el flujo "Productor de Tareas" con un nodo "Webhook". Configuramos el método HTTP a POST y copiamos la URL de prueba. Este nodo recibirá los datos de la tarea (título y descripción) desde una llamada externa.



Conectamos un nodo RabbitMQ de acción a la salida del Webhook. Seleccionamos las credenciales, la operación "Send a Message to RabbitMQ", el modo "Queue" y especificamos la cola task_created. Activamos "Send Input Data" para enviar el JSON recibido por el webhook directamente a la cola.

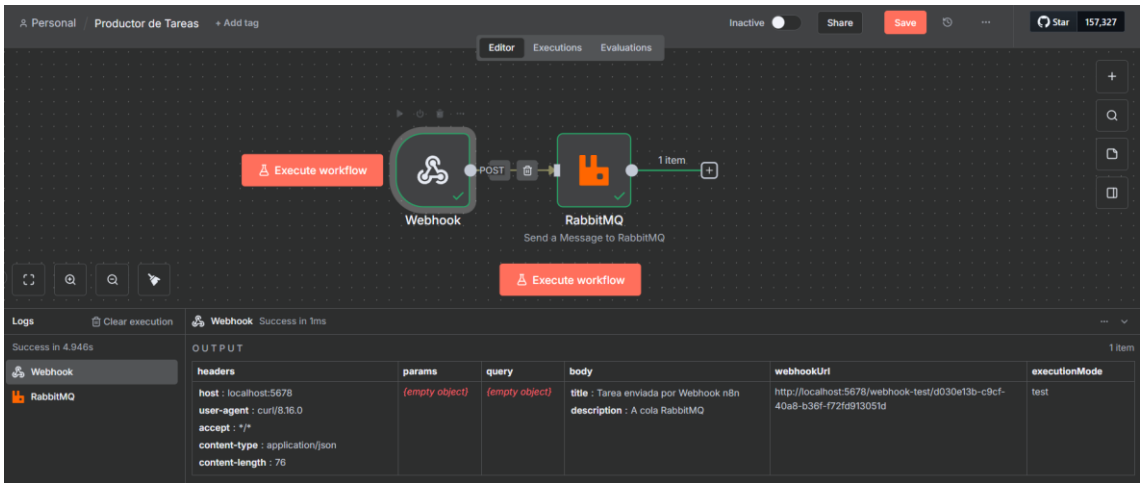


Iniciamos el flujo y este espera a la petición. Realizamos una prueba enviando una petición curl tipo POST al webhook de n8n con un JSON que contiene title y description.



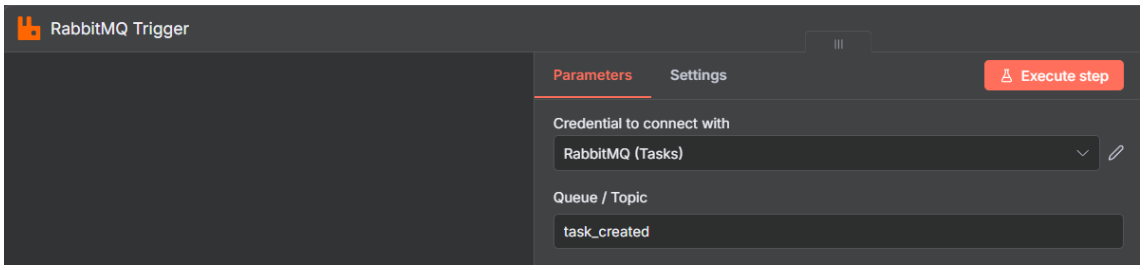

```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P01
$ curl -X POST -H "Content-Type: application/json" -d "{\"title\": \"Tarea enviada por Webhook n8n\", \"description\
": \"A cola RabbitMQ\"}" http://localhost:5678/webhook-test/d030e13b-c9cf-40a8-b36f-f72fd913051d
{"message":"Workflow was started"}
```

Observamos la ejecución exitosa del flujo productor. El Webhook recibe los datos ("Tarea enviada por Webhook n8n") y el nodo RabbitMQ confirma el envío del mensaje a la cola.

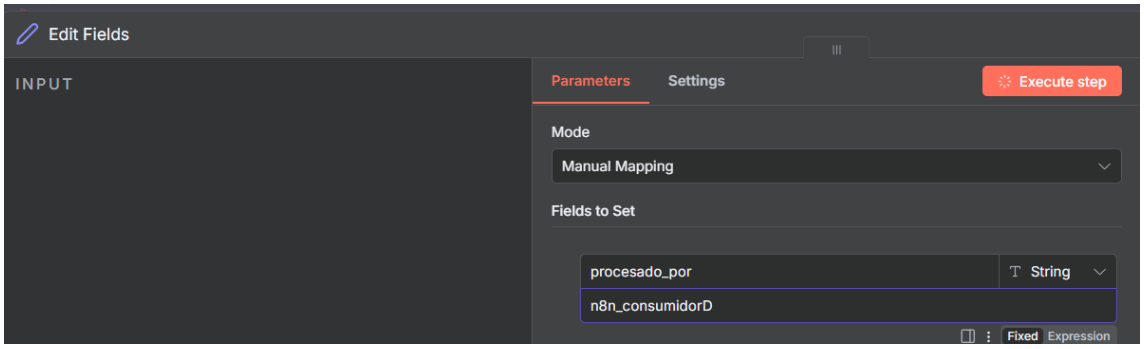


2. *Flujo B: El Consumidor (RabbitMQ → Log)*

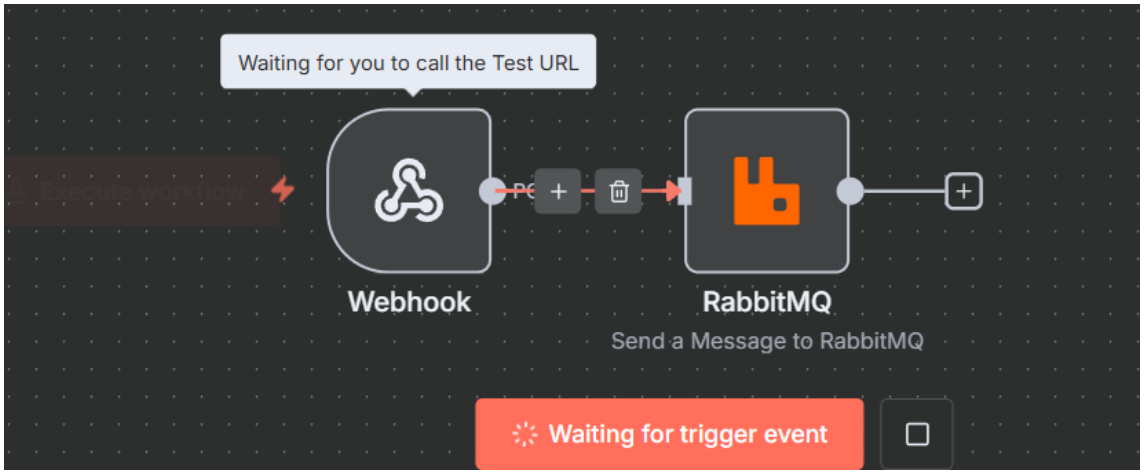
Creamos un nuevo flujo "Consumidor de Tareas". El nodo inicial es un "RabbitMQ Trigger". Lo configuramos con las credenciales y especificamos la cola task_created para que el flujo se active automáticamente cada vez que llegue un mensaje a dicha cola.

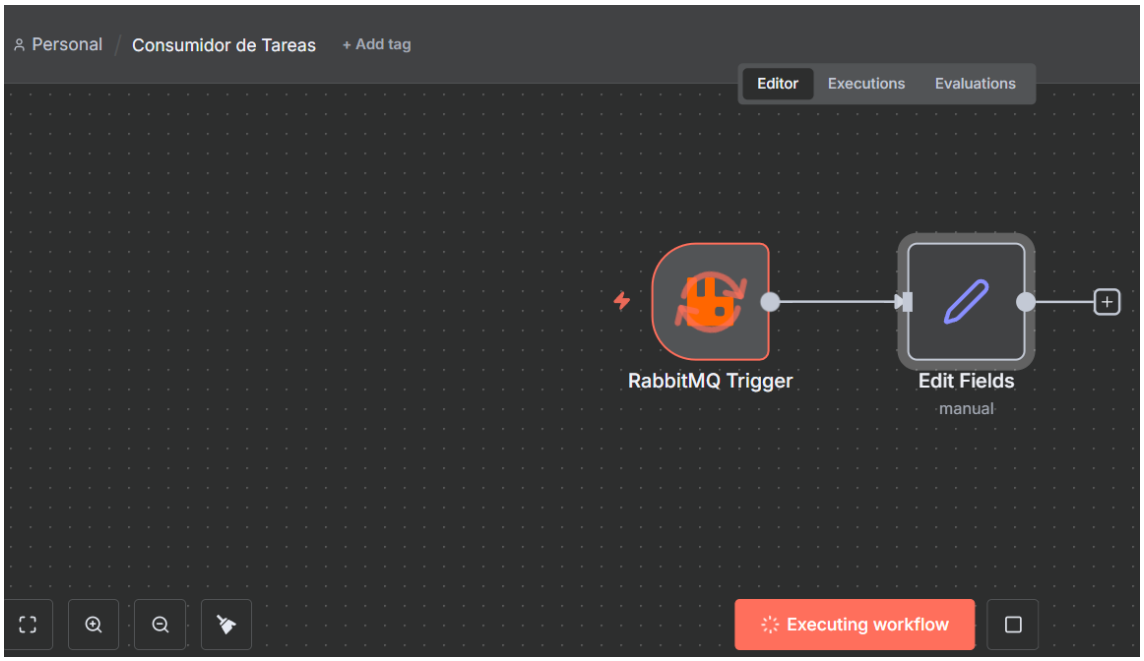


A continuación, para simular un procesamiento de datos, conectamos un nodo Edit Fields a la salida del Trigger. Configuramos un nuevo campo llamado procesado_por y le asignamos el valor fijo n8n_consumidor. Esto nos servirá para identificar en el futuro qué servicio procesó el mensaje.



Guardamos y activamos el flujo productor y consumidor.

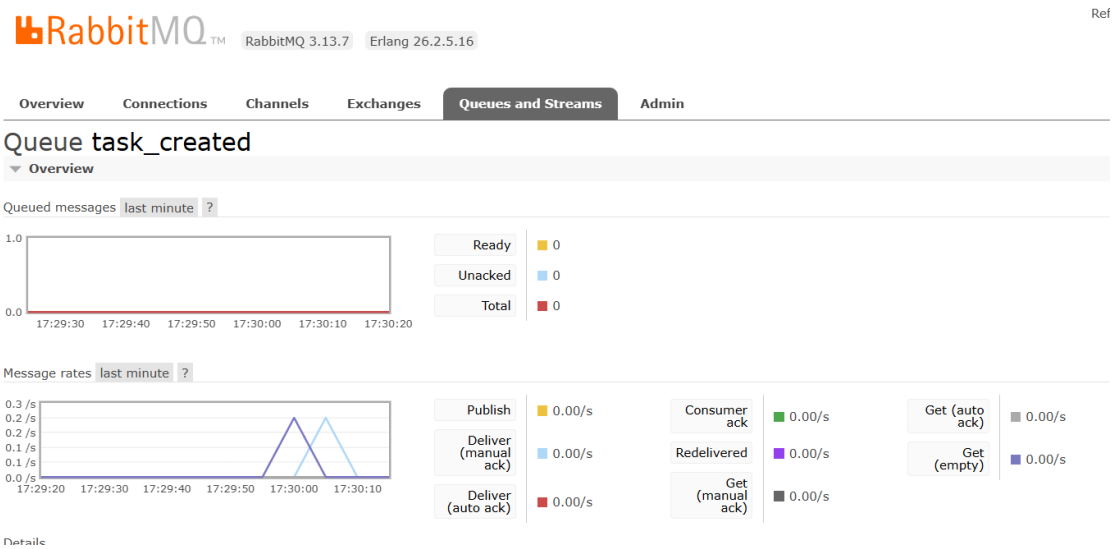


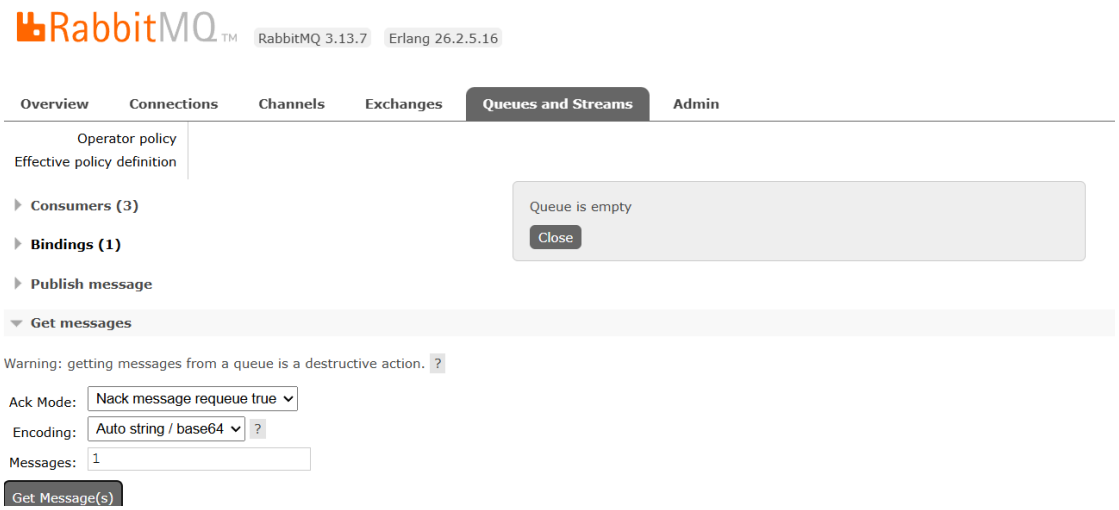


Al volver a ejecutar el comando curl, nos encontramos ante un escenario de consumo en competencia. Dado que tanto nuestro flujo de n8n como el contenedor worker de Python (que sigue activo) están escuchando la misma cola (task_created), solo uno de ellos procesará el mensaje.

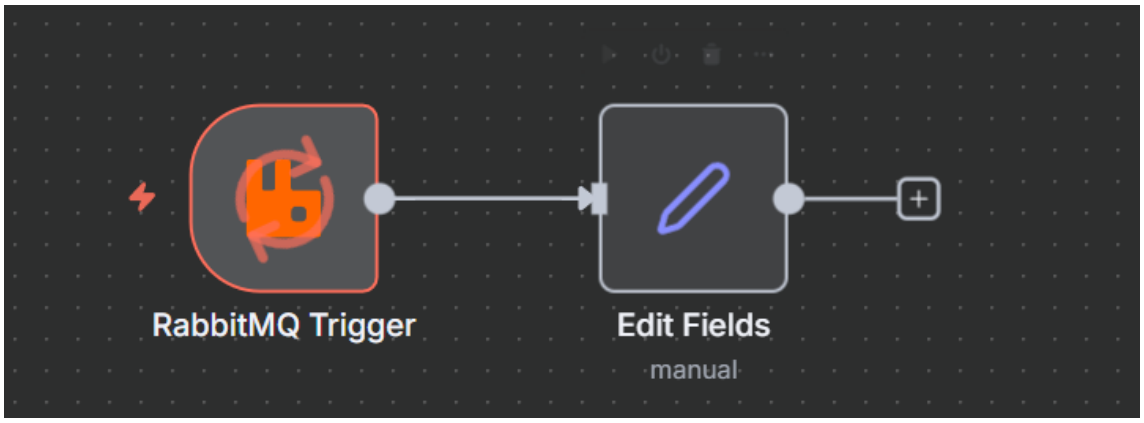
```
after you click this button) }
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P01
$ curl -X POST -H "Content-Type: application/json" -d "{\"title\": \"Tarea enviada por Webhook n8n\", \"description\
": \"A cola RabbitMQ\"}" http://localhost:5678/webhook-test/d030e13b-c9cf-40a8-b36f-f72fd913051d
{"message":"Workflow was started"}
```

Vemos como el mensaje si ha sido procesado por el worker de Python y no hay ningún mensaje en la cola pendiente por procesar:





Observamos que n8n no muestra ninguna ejecución. Esto significa que el worker de Python ha sido más rápido, ha consumido el mensaje de la cola y, por tanto, n8n no ha llegado a recibirlo.

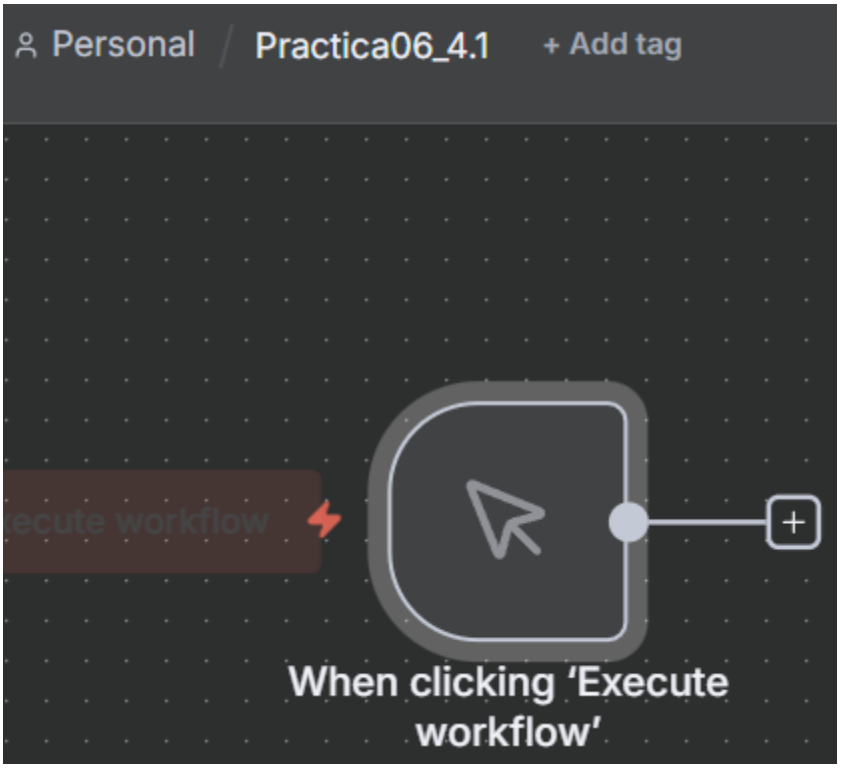


4. Ejercicios Propuestos

4.1. Ejercicio 1: Eliminación de Tareas (Dificultad: Baja)

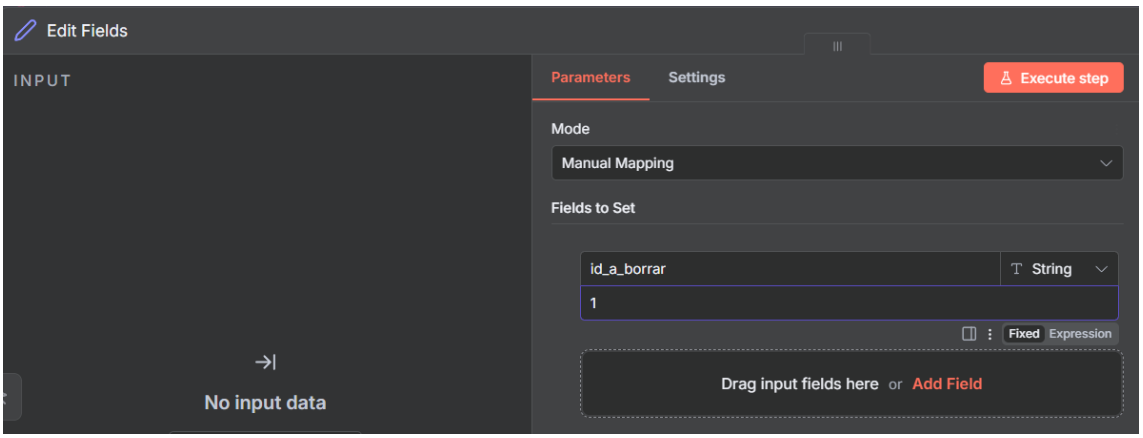
1. *Cree un flujo que comience con un Manual Trigger.*

Creamos un flujo que comienza con un "Manual Trigger".



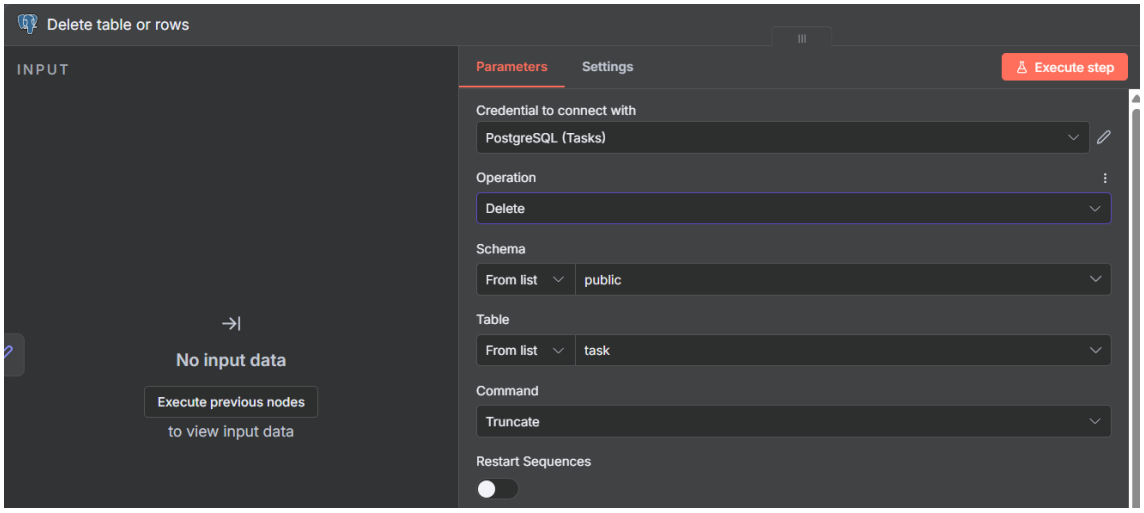
2. *Añada un nodo Edit Fields (Set) para definir la ID de la tarea a eliminar (ej. {"id_a_borrar": 1}).*

A continuación, añadimos un nodo "Edit Fields" donde definimos una variable id_a_borrar con el valor del ID de la tarea que queremos eliminar (ej. 1).



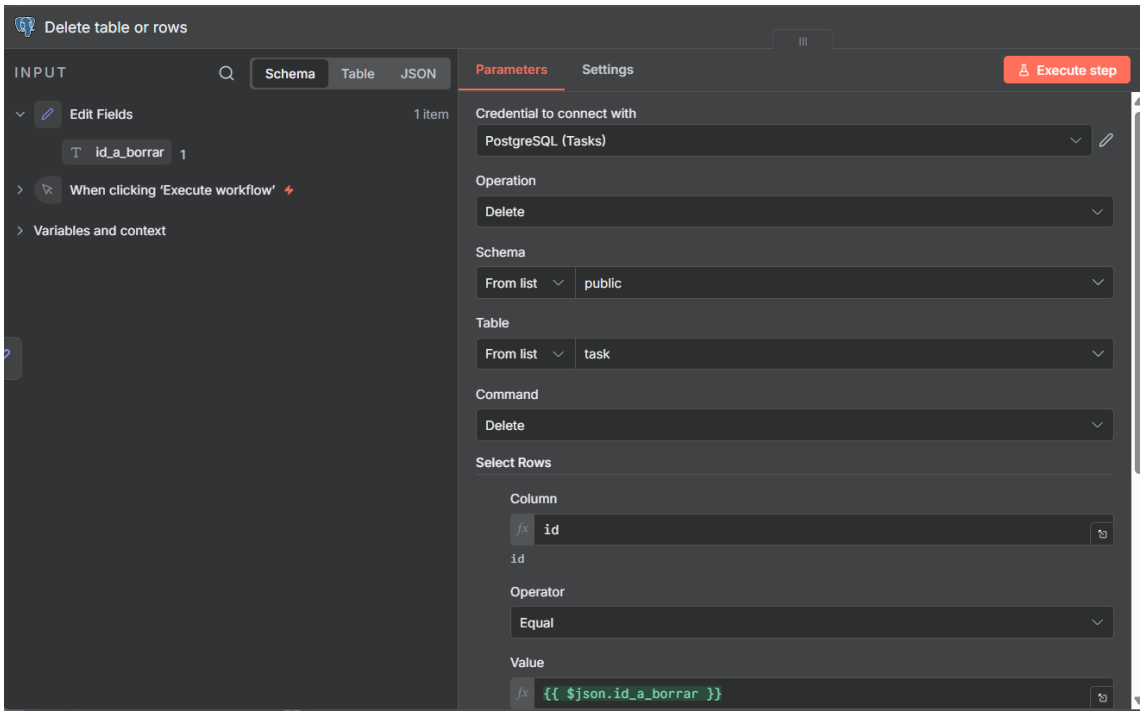
3. *Use un nodo PostgreSQL con la operación Delete.*

Conectamos un nodo PostgreSQL configurado con la operación "Delete". Utilizamos las credenciales añadidas anteriormente y definimos que la operación se de en la tabla “tasks”.



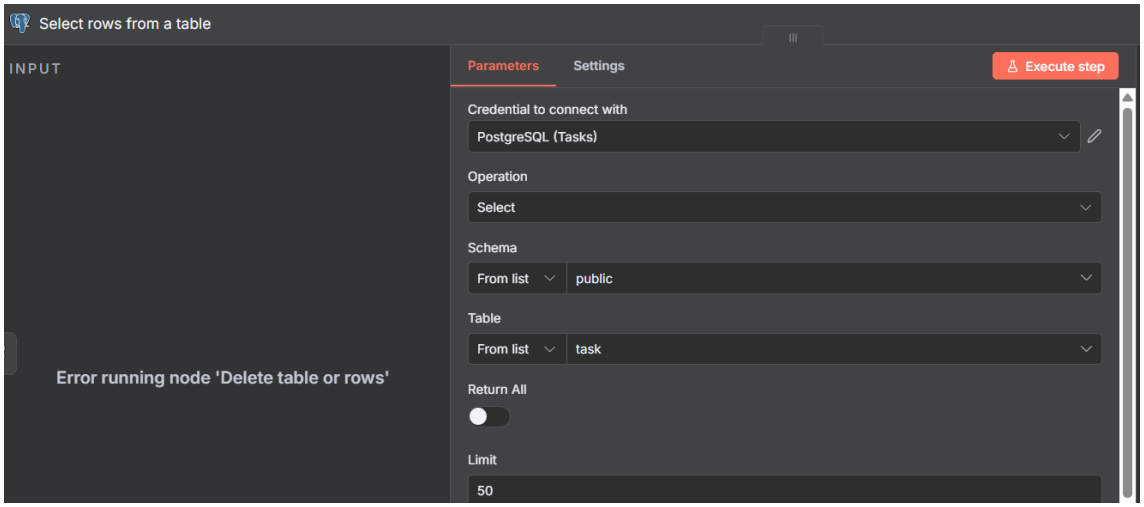
4. *Configure la condición WHERE para que elimine la fila donde id sea igual a la expresión {{ $json.id_a_borrar }}.*

En la sección "Select Rows", establecemos una cláusula WHERE para que solo elimine la columna id que es igual a la expresión {{ $json.id_a_borrar }} obtenida del nodo anterior.

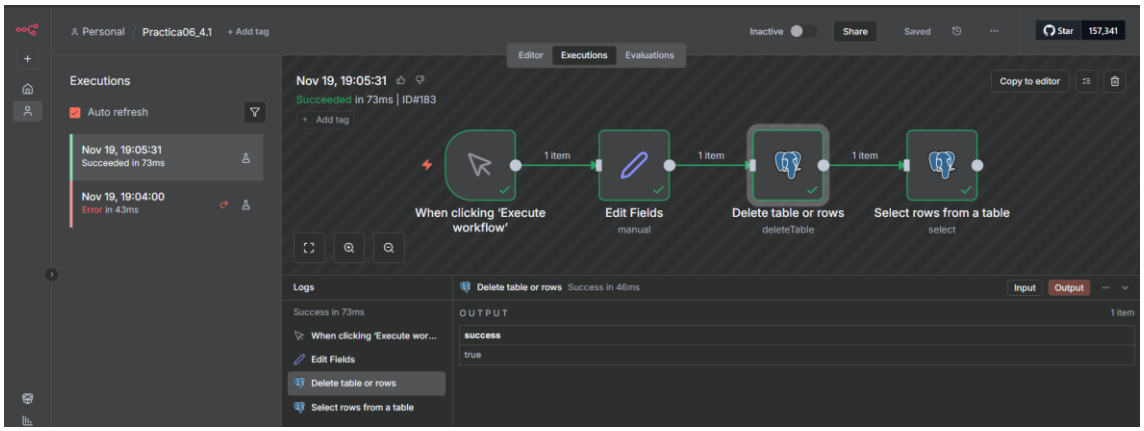


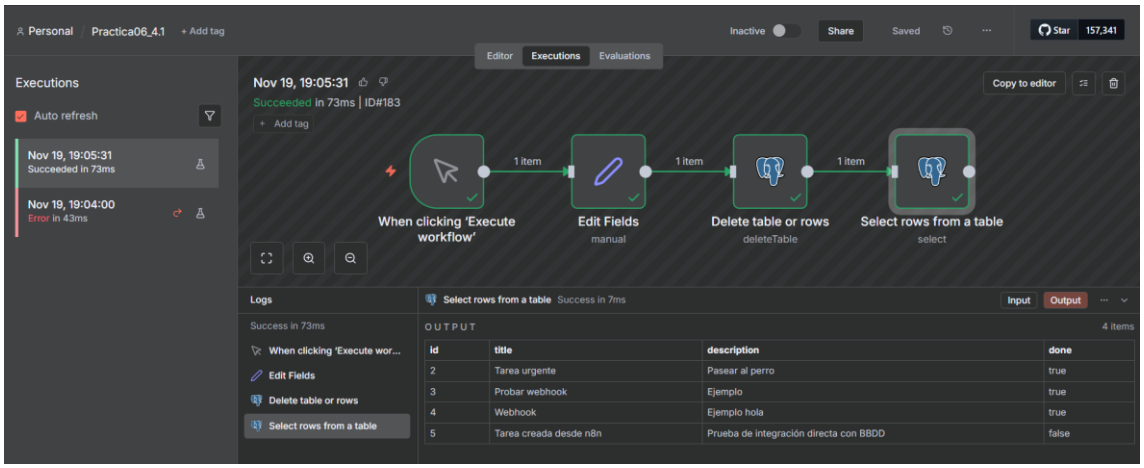
5. *(Opcional) Añada otro nodo PostgreSQL (Select) al final para verificar que la tarea ha sido eliminada.*

Añadimos un nodo "Select" al final para ver todos los elementos de la tabla tras la eliminación.



Al ejecutar el flujo completo, el nodo Delete devuelve success: true, y en el nodo Select de Postgress, verificaríamos que el ID 1 ya no existe en la lista.

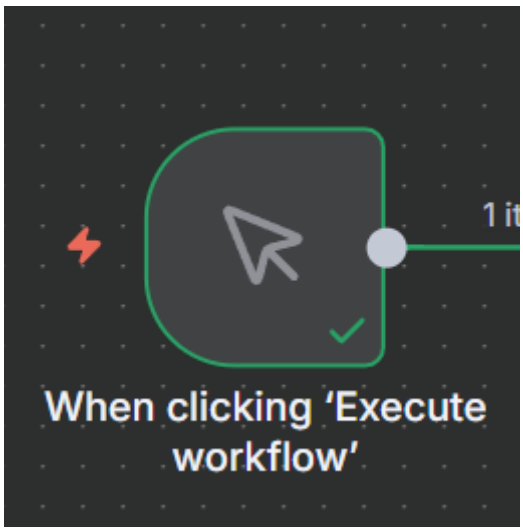




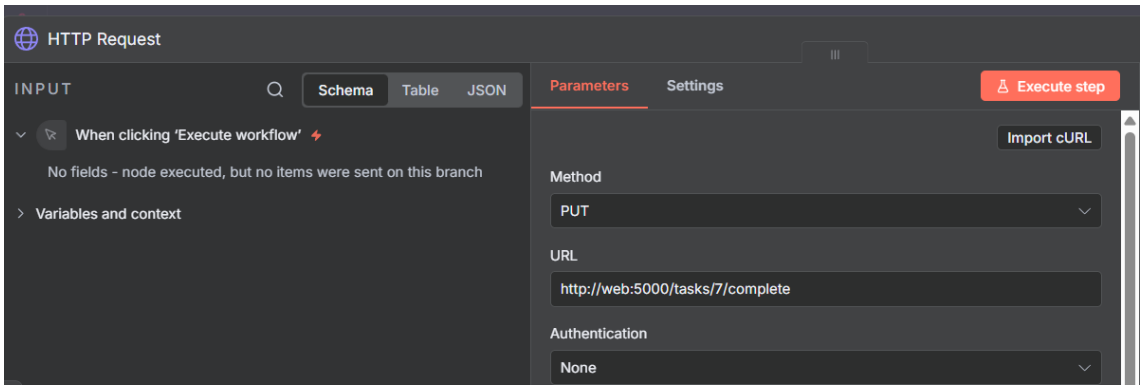
4.2. Ejercicio 2: Servicio de Notificación (Dificultad: Media)

1. *Modifique el flujo "Productor de Tareas" (o cree uno nuevo) para que llame al endpoint PUT /tasks/<id>/complete de la API web de la Práctica 5.*

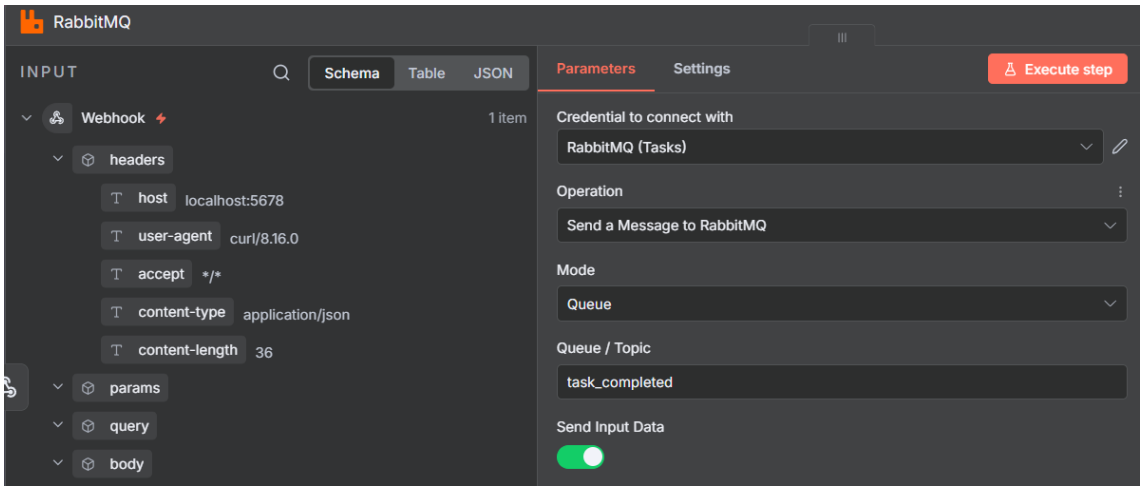
Creamos un nuevo flujo productor que comienza con un "Manual Trigger".



Realizamos una llamada a la API del proyecto utilizando el nodo “HTTP Request”. En este definimos el método PUT y la ruta correspondiente para actualizar la tarea con id 7 al estado completado. Un dato a destacar, es que debemos especificar el nombre del contenedor web con el que n8n lo está viendo el lugar de “localhost”, en este caso “web:5000”.

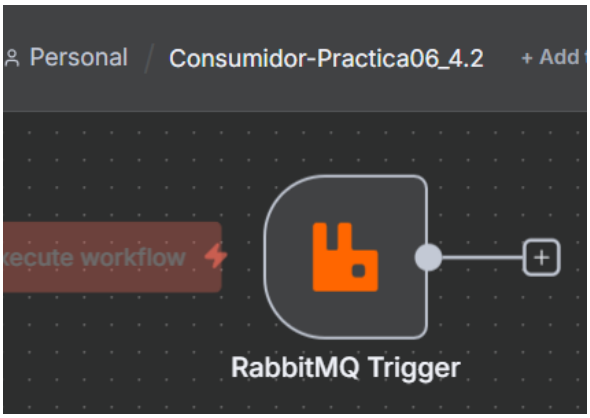


Por último, creamos un nodo “RabbitMQ” para que encole la petición anterior en la cola task_completed y que esta pueda ser atendida posteriormente por el consumidor.



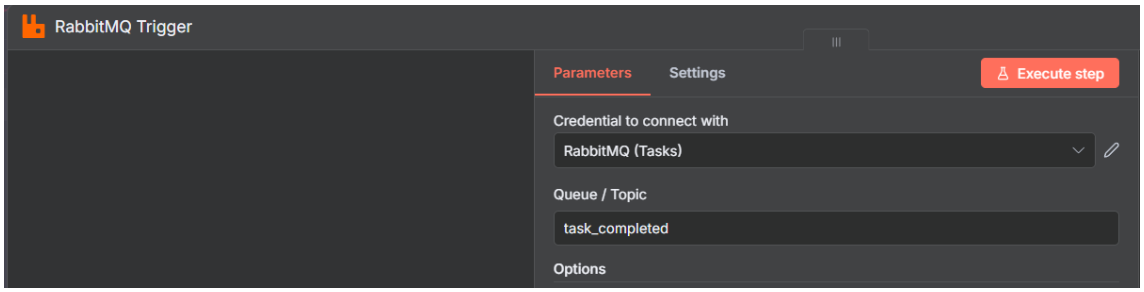
2. *Cree un nuevo flujo en n8n que comience con un RabbitMQ Trigger.*

Creamos un nuevo flujo consumidor “Consumidor-Práctica06_4.2” que inicia con un "RabbitMQ Trigger".



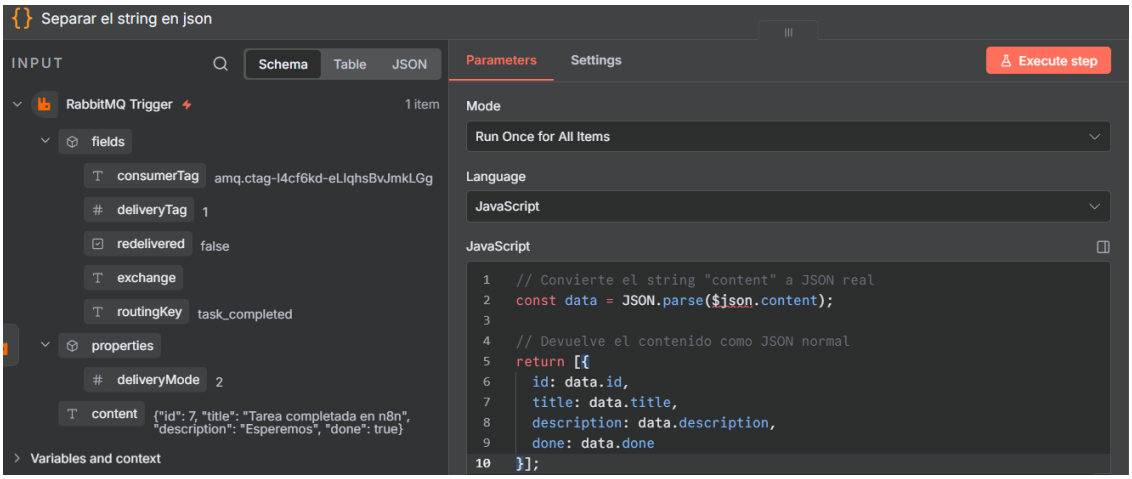
3. *Configure este trigger para que escuche en la cola task_completed.*

Esta vez, configuramos el nodo para que escuche la cola task_completed.

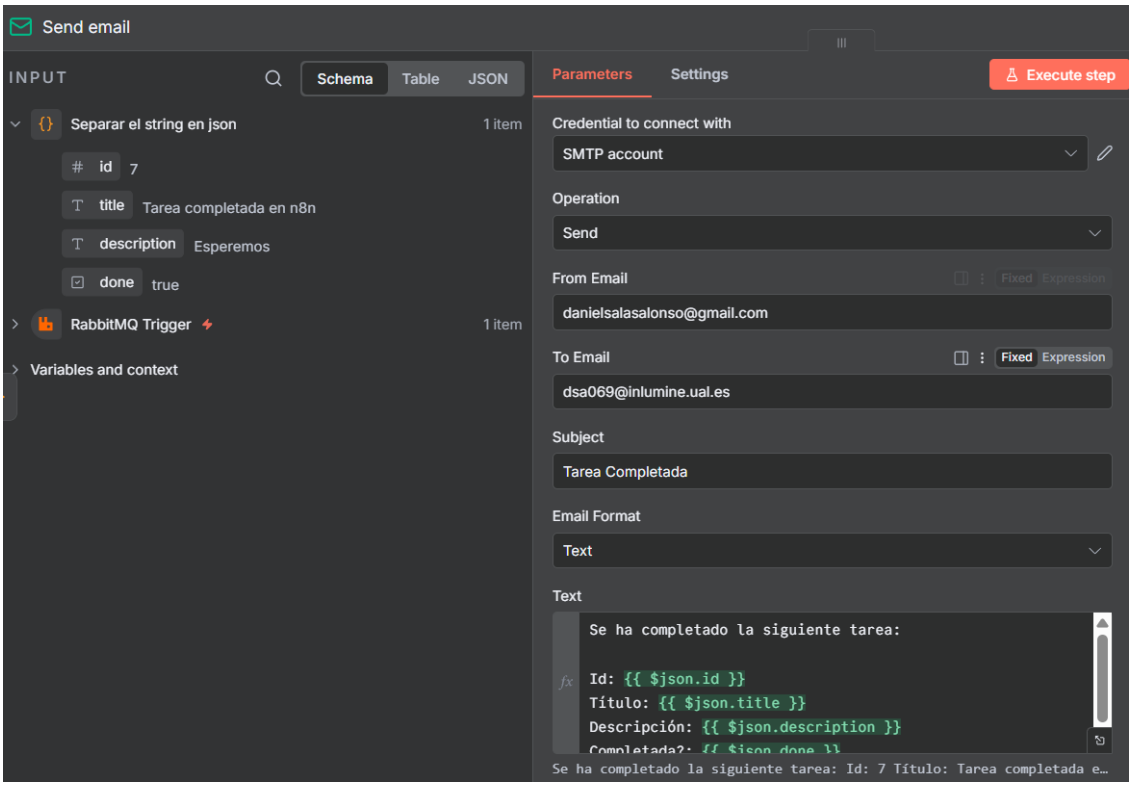


4. *Cuando reciba un mensaje, el flujo debe usar un nodo Send Email (o Slack, etc.) para notificar que la tarea ha sido completada, incluyendo el título de la tarea en el cuerpo del mensaje.*

Añadimos un nodo "Code" para convertir el contenido del mensaje recibido de RabbitMQ (que viene como string en content) a un objeto JSON estructurado para poder acceder a campos como title o id.

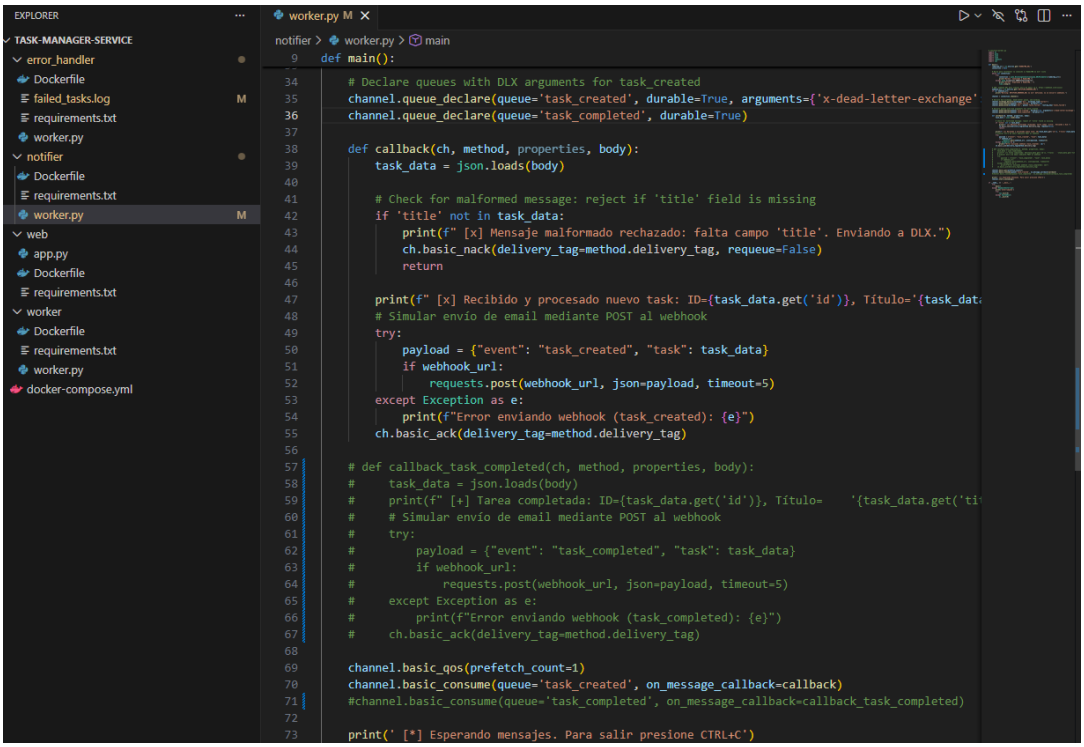


Conectamos un nodo "Send Email". Configuramos el correo emisor, el destinatario y el asunto ("Tarea Completada"). En el cuerpo del mensaje, utilizamos expresiones para mapear los datos dinámicos provenientes del nodo anterior (Id, Título, Descripción y si ha sido completada).

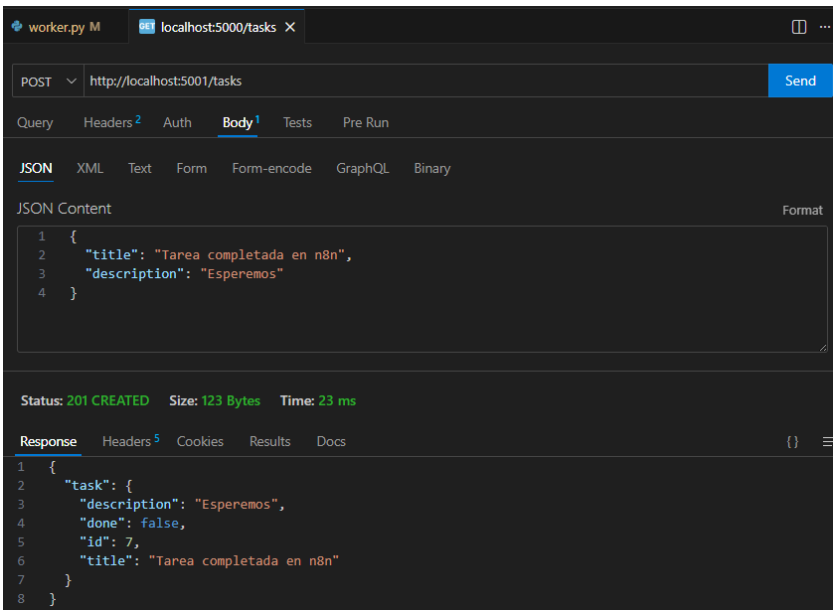


5. *Asegúrese de que el worker de Python de la P5 no esté escuchando en esta cola, para que n8n sea el único consumidor.*

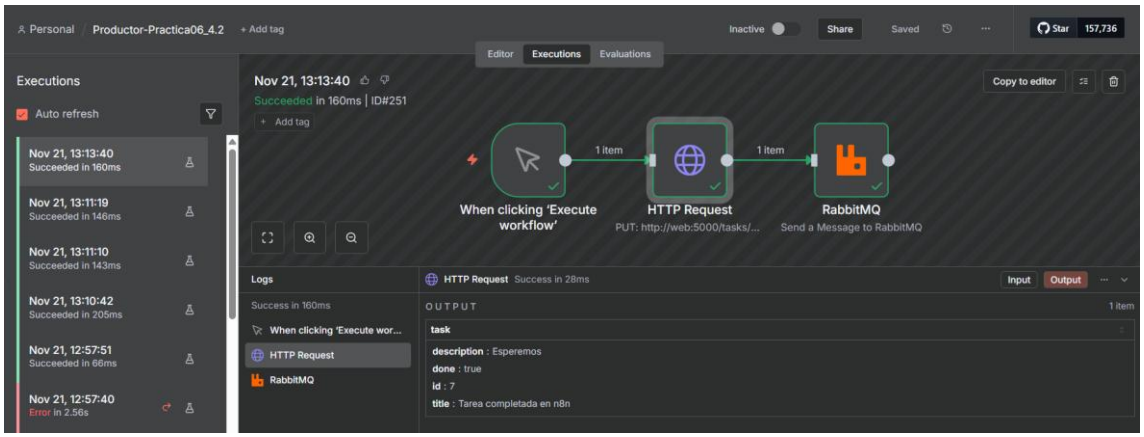
Nos aseguramos en el código del worker del notificador, en nuestro proyecto, de comentar el método callback de “task_completed” y el consume de este método, para que no esté escuchando en esta cola y no consuma los mensajes.



Creamos una nueva tarea para realizar la prueba. Esta tarea será la que especificaremos en la llamada a la API con el endpoint PUT en el nodo “HTTP Request” del productor (Id: 7).

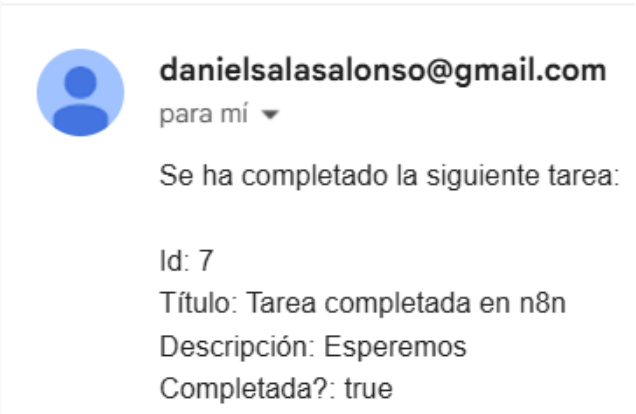


Observamos la ejecución exitosa del flujo completo: los nodos se muestran en verde, y en la salida del nodo HTTP Request verificamos que la API ha devuelto el objeto tarea con el estado done: true y el ID correspondiente (ID: 7), confirmando que la operación se realizó correctamente antes de encolar el mensaje.



Observamos la ejecución del flujo: el nodo “RabbitMQ Trigger” recibe el mensaje de la cola, el JSON se procesa y el nodo de Email envía el correo exitosamente.

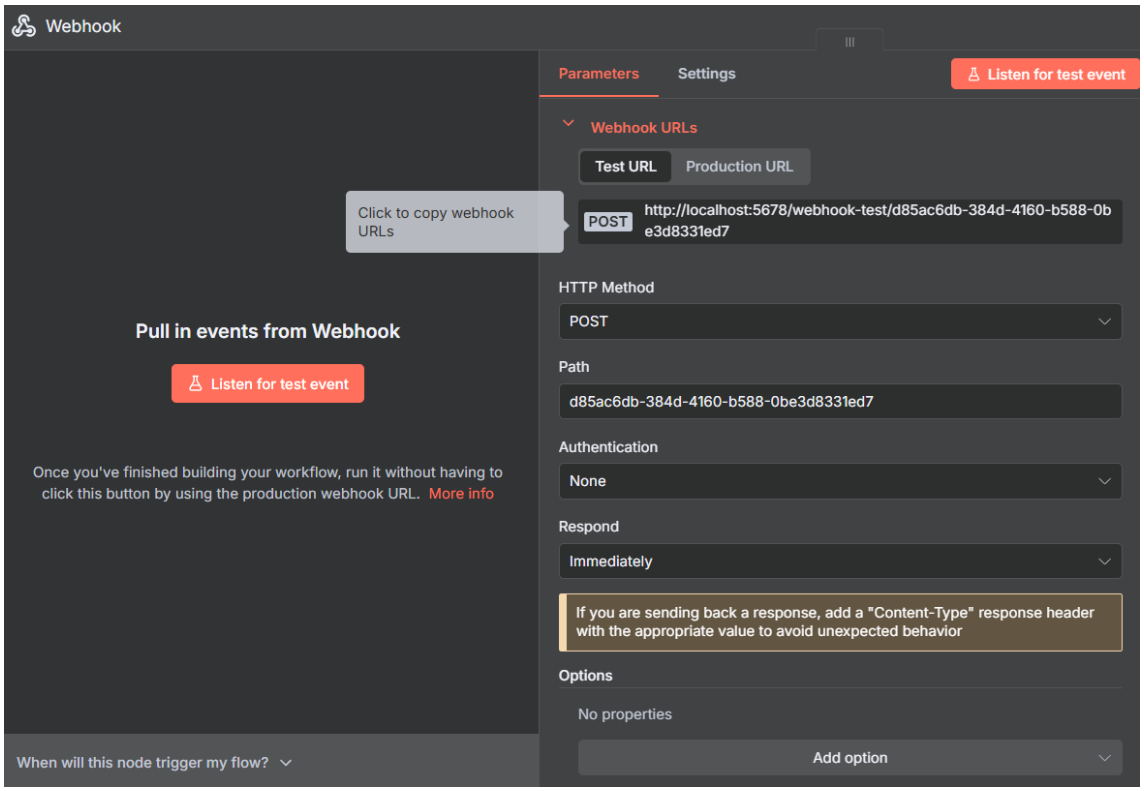
Verificamos la bandeja de entrada donde recibimos el correo con el formato definido, confirmando que el sistema de notificación funciona correctamente.



4.3. Ejercicio 3: Reemplazo de la API con Webhook (Dificultad: Alta)

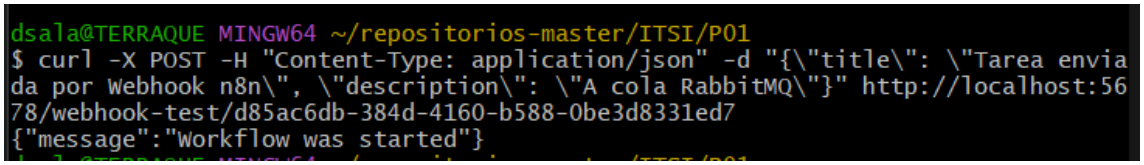
1. *Cree un nuevo flujo de trabajo que comience con un Webhook Trigger.*

Creamos un nuevo flujo que reemplazará la funcionalidad de creación de tareas de la API. Iniciamos con un "Webhook Trigger" configurado en método POST.



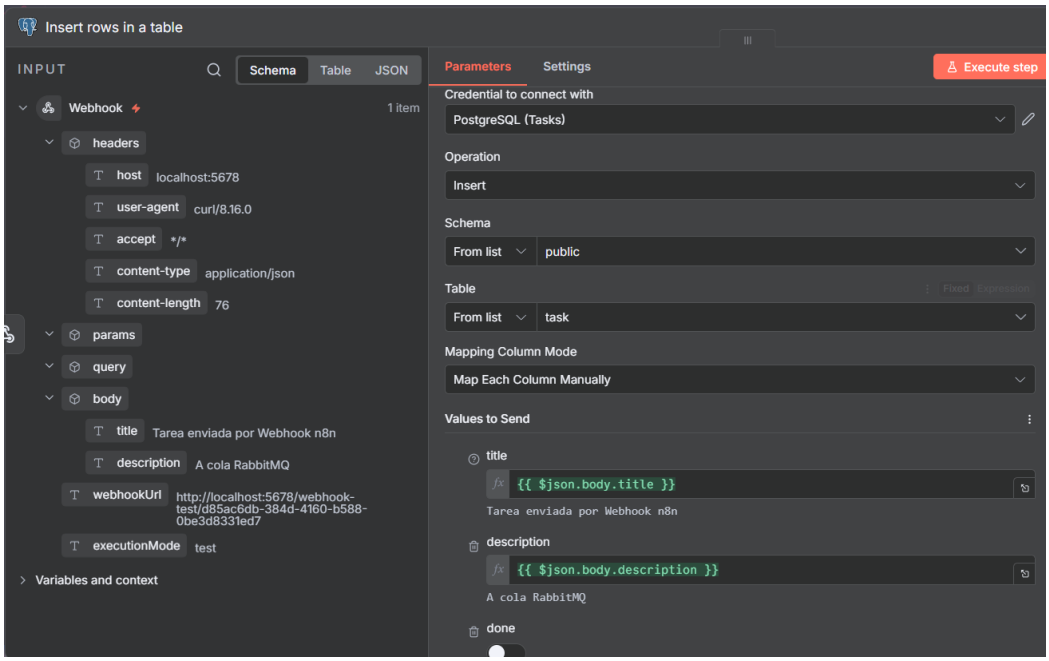
2. *El flujo recibirá un JSON con title y description (igual que la API de Flask).*

El “Webhook” recibirá una petición curl POST que contendrá el JSON con title y description.



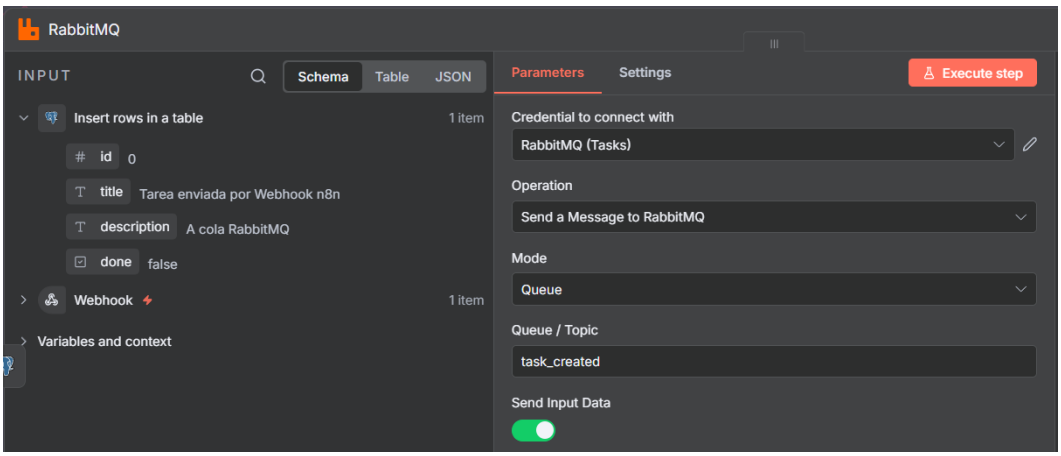
3. *Use un nodo PostgreSQL (Insert) para insertar la nueva tarea en la tabla task.*

Conectamos el Webhook a un nodo PostgreSQL en modo "Insert" para crear una nueva tarea en la tabla “task”. Mapeamos los campos title y description utilizando las expresiones {{ $json.body.title }} y {{ $json.body.description }} provenientes del Webhook.



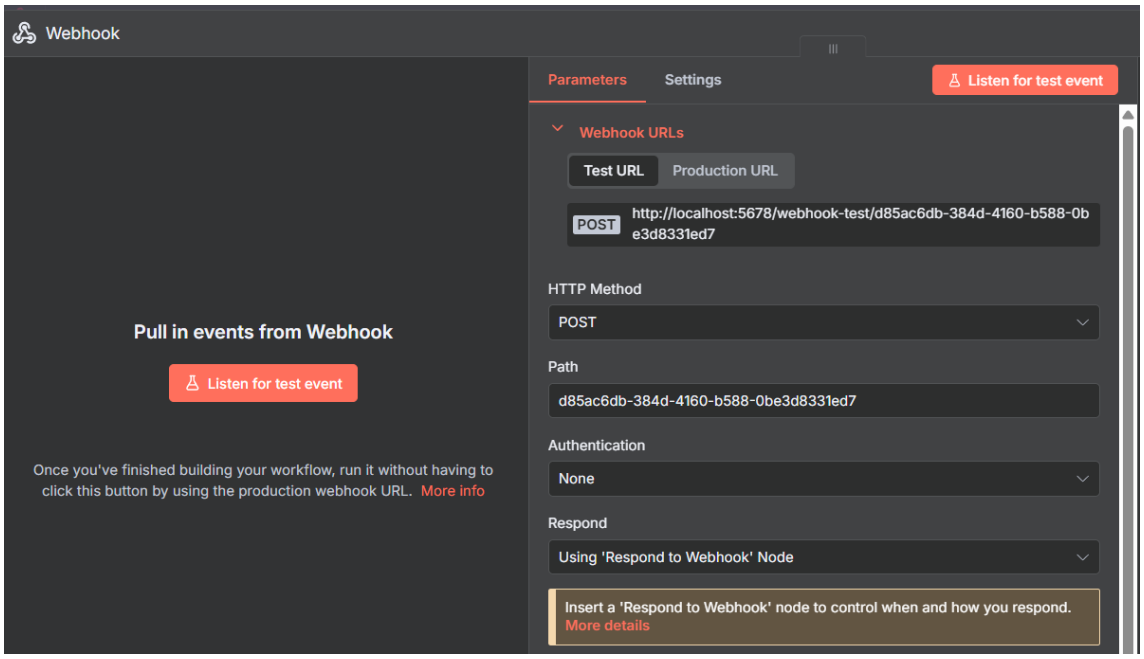
4. *Use un nodo RabbitMQ (Send) para publicar los datos de la tarea recién creada (incluyendo la id devuelta por el nodo de inserción) en la cola task_created.*

A continuación, conectamos un nodo RabbitMQ para notificar la creación. Configuramos la operación "Send a Message" a la cola task_created, enviando los datos de la tarea recién insertada (incluyendo el nuevo ID generado por la BBDD) marcando la opción “Send Input Data”.

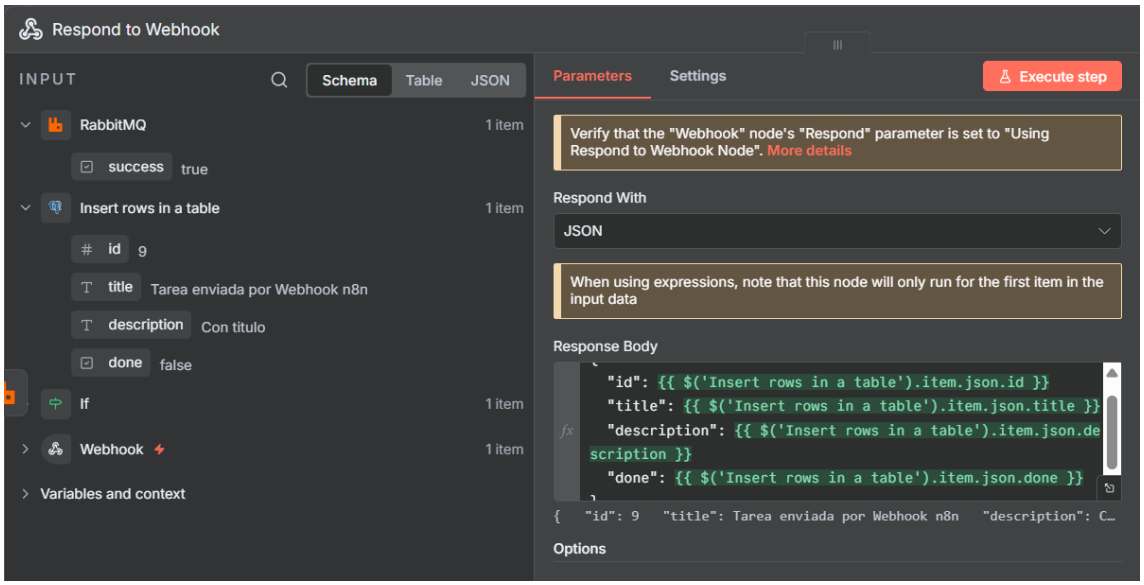


5. *Configure el nodo Webhook Trigger para que responda inmediatamente con los datos de la tarea creada (simulando la respuesta 201 Created de la API).*

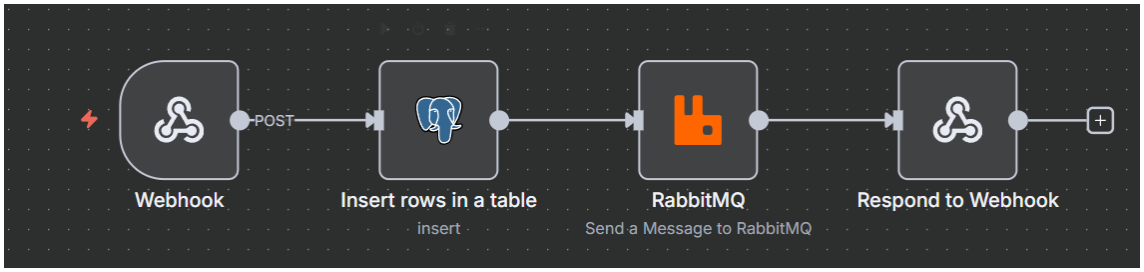
Para ello, en el nodo “Webhook” inicial, modificamos el campo “Respond” para que devuelva como respuesta lo especificado en nuestro nuevo nodo "Respond to Webhook" que crearemos a continuación.



Finalmente, cerramos el flujo con un nodo "Respond to Webhook". Lo configuramos para responder con un JSON que contiene los datos de la tarea creada, simulando una respuesta 201 Created estándar de una API REST.

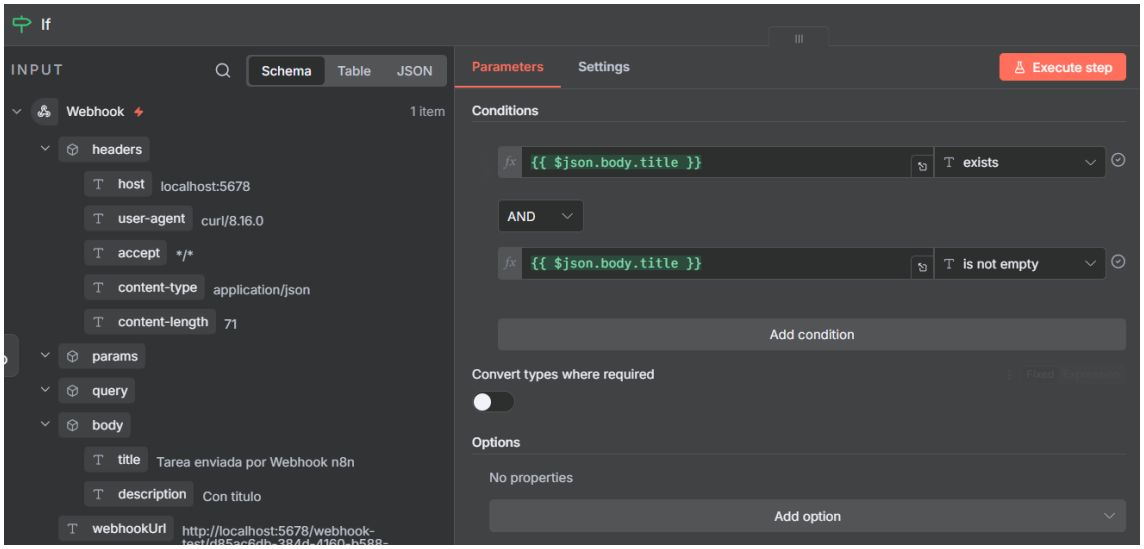


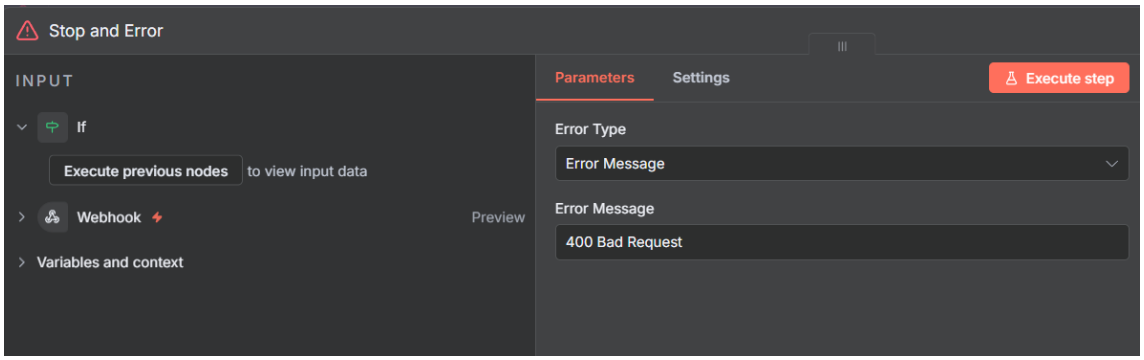
Observemos la estructura del flujo hasta ahora:



6. *(Bonus) Añada un IF para validar que el campo title existe, y use un nodo Stop and Error si no es así, simulando la respuesta 400 Bad Request.*

Para añadir robustez, insertamos un nodo "IF" después del “Webhook” inicial. Configuramos la condición para verificar si el campo title existe y no está vacío. Si la condición es falsa, derivamos a un nodo "Stop and Error" (o Respond to Webhook con error 400), protegiendo el flujo de datos inválidos.

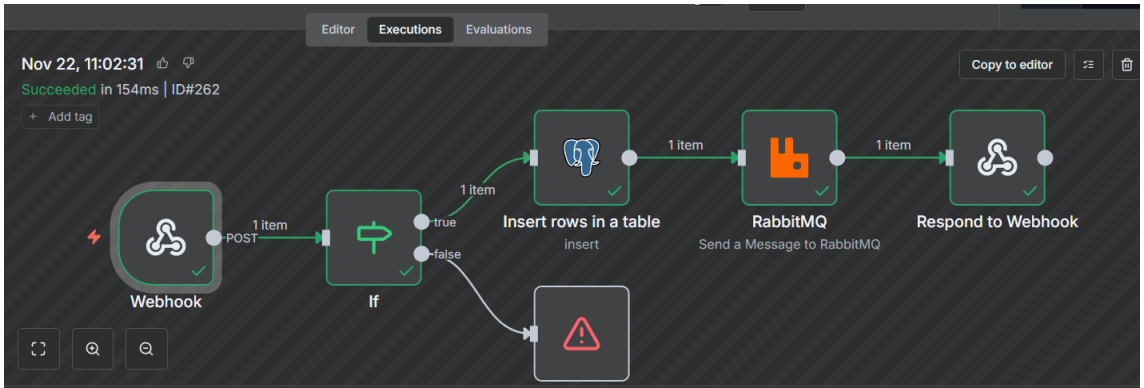


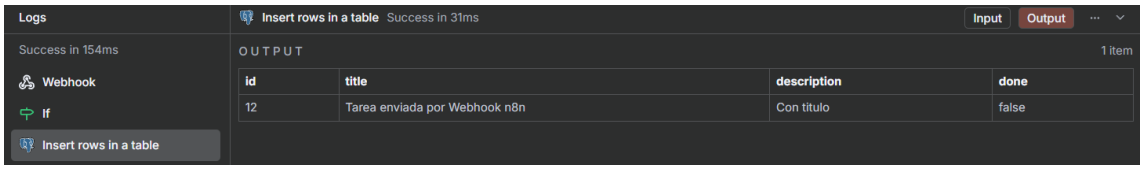


Para comprobar el correcto funcionamiento del flujo, realizamos una nueva petición POST mediante curl al Webhook, indicando el título y la descripción de la nueva tarea. Observamos como obtenemos una respuesta inmediata por parte del nodo “Webhook Response” indicando los datos de la nueva tarea y por tanto la correcta creación de esta.

```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P01
$ curl -X POST -H "Content-Type: application/json" -d "{\"title\": \"Tarea enviada por Webhook n8n
\", \"description\": \"Con titulo\"}" http://localhost:5678/webhook-test/d85ac6db-384d-4160-b588-0
be3d8331ed7
{
  "id": 12,
  "title": Tarea enviada por Webhook n8n,
  "description": Con titulo,
  "done": false
}
```

En la ejecución del flujo se observa que el Webhook recibe la petición correctamente, el nodo If evalúa la condición de forma adecuada, se inserta la nueva entrada en la base de datos, el mensaje resultante se envía a la cola mediante RabbitMQ y, finalmente, el Webhook devuelve la respuesta configurada.





Finalmente, ponemos a prueba esta última modificación en el flujo, realizando una petición curl mal formada, en la que omitimos intencionadamente el campo title y solo enviamos la description.

```
dsala@TERRAQUE MINGW64 ~/repositorios-master/ITSI/P01
$ curl -X POST -H "Content-Type: application/json" -d "{\"description\": \"Sin titulo\"}" http://l
ocalhost:5678/webhook-test/d85ac6db-384d-4160-b588-0be3d8331ed7
```

Observamos en la traza de ejecución que el nodo If evalúa la entrada y, al no encontrar el título, dirige el flujo hacia la rama false (False Branch). Esto activa el nodo Stop and Error, deteniendo el proceso y devolviendo el mensaje "400 Bad Request", lo que confirma que nuestro sistema de validación está protegiendo correctamente la base de datos contra datos incompletos.