CLOUD COMPUTING

Lab CC 2023 Executor Service in the Cloud

Autores:
Iván Encinas
David Valdaliso
Yazid Berrekia

ÍNDICE

1. INTRODUCCIÓN

Antes de empezar con todo, tenemos que entender que es un FaaS, donde podríamos decir que son las siglas Function as a Services, es un tipo de servicio de Cloud Computing que permite que los desarrolladores realicen funciones, como por ejemplo diseñar o ejecutar aplicaciones como funciones sin tener que preocuparse del mantenimiento de la infraestructura, en resumen es un servicio que permite ejecutar código en respuesta de sucesos, sin gestionar la infraestructura de las aplicaciones de microservicios. Podemos decir también que se basa en los eventos y se ejecuta en contenedores sin estados [1].

Con la FaaS, el proveedor de servicios en la nube gestiona automáticamente el hardware físico, el sistema operativo de máquinas virtuales y la gestión del software del servidor web. Esto permite a los desarrolladores centrarse únicamente en funciones individuales en el código de su aplicación [2].

Para entender definitivamente el FaaS consideramos importante conocer los servicios ofrecidos, estos son [3]:

- Servicios de autenticación: Encargados del login de los usuarios del programa.
- Servicio de base de datos: Almacenan información en las bases de datos del programa para poder utilizar esta información.
- Almacenamiento de archivos.

2. DESARROLLO DEL PROBLEMA

2.1. OAUTH2-PROXY

En primer lugar nos encargamos de desarrollar el oauth2-proxy, que ni más ni menos es un proxy de autenticación que se integra con proveedores de autenticación basados en OAuth 2.0. Su principal función es gestionar la autenticación y la autorización para aplicaciones web al actuar como intermediario entre el usuario y la aplicación protegida. En el caso particular de este proyecto utilizamos google, empleando una imagen de docker bitname/auth2-proxy, luego de crear un proyecto en google llamando authproxy y luego creando ID de clientes OAuth2.0, cuando creamos esto nos entrega un id-Cliente y un secrete cliente que desde la imagen de docker, podemos proceder a la autenticación.

Figura 2.1. Logo OAuth2-Proxy

2.2. REST API

Una API REST es una interfaz de comunicación que utiliza el protocolo HTTP (HyperText Transfer Protocol) para la obtención y ejecución de datos, podremos como hemos dicho anteriormente ejecutar operaciones sobre los datos en formato JSON. Esta interfaz de comunicación se basa en el modelo cliente-servidor, siendo el cliente el que solicita realizar una operación sobre los datos que se encuentran en la API, mientras que el servidor se encarga de entregar y procesar los datos a solicitud del cliente [4].
En resumen, las API son conjuntos de definiciones y protocolos que se utilizan para diseñar e integrar el software de las aplicaciones.
NestJS es un framework progresivo de Node.js para la creación de aplicaciones eficientes, confiables y escalables del lado del servidor, el cual está construido y es compatible con TypeScript, combinando elementos de la programación orientada a objetos [5].

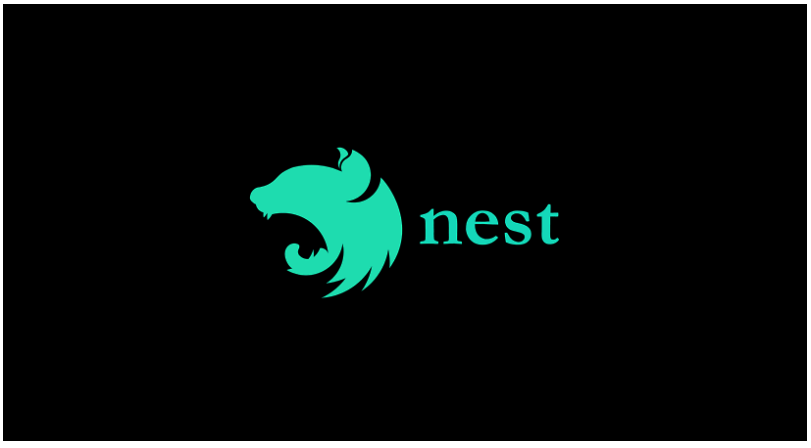


Figura 2.2. Logo NestJS

En nuestra API, destacan cuatro funcionalidades principales.
La primera, conocida como "Enviar Trabajo", constituye la piedra angular de la API. Al invocar esta función, se requiere proporcionar un nombre y una URL que apunte al código almacenado en GitHub, el cual se desea ejecutar. Además, se genera un identificador único utilizando la biblioteca 'node-uuid' para identificar estos trabajos. En esencia, esta función se encarga de transmitir un trabajo a través de NATS al microservicio llamado "workers", el cual se responsabiliza de ejecutar dicho trabajo y generar los resultados correspondientes

La segunda tarea que realiza nuestra API es obtener el estado del trabajo. Consultamos este estado a través de un API dedicado que conoce los estados de las tareas, enviándole el ID del trabajo, y nos retorna información sobre si ha terminado o está pendiente.
La tercera tarea es muy similar a la anterior, pero obteniendo el resultado del trabajo ejecutado. Aquí, podemos almacenar un resultado exitoso o un error de ejecución, como por ejemplo, una URL de GitHub incorrecta o problemas en el código de Python presente en el archivo 'main.py', o cualquier otro error relacionado con la ejecución del trabajo.
La cuarta tarea es para obtener un listado de los resultados que se tiene por usuario.
La quinta función consiste en obtener las métricas de la media general de la duración de las tareas y conocer el estado del sistema. Tomamos tiempos antes de comenzar la ejecución de la tarea y luego de que termine, independientemente del resultado, esto lo almacenamos y realizamos una operación de media, tomando la diferencia entre el tiempo que comienza la tarea y termina.

Para salvaguardar la seguridad de nuestra Frontend (API), implementamos una herramienta de NestJS llamada AuthGuard. Esta herramienta ofrece una solución versátil al permitir su aplicación a endpoints específicos. Su función principal radica en la validación de la autenticidad de los tokens, garantizando así un acceso seguro a los recursos protegidos.
En la variable entorno que tiene el proyecto Fe en el archivo .env se define el usuario que tiene permisos para obtener métricas.
Destacó que cada microservicio tiene este archivo .env, para sus variables de conexión

2.3. WORKER

Cuando hablamos de worker nos referimos a un consumidor de mensajes que está suscrito a un tema específico para procesar los mensajes que se publican en ese tema. Podríamos definir algunos puntos claves sobre cómo funcionan los worker:

1. Suscripción a temas.
2. Recepción de mensajes.
3. Procesamiento de mensajes.
4. Escalabilidad.
5. Balanceo de carga.

Los "workers" en NATS son parte de un sistema de mensajería que permite la comunicación eficiente y desacoplada entre diferentes componentes de una aplicación o entre diferentes aplicaciones. Esta arquitectura facilita la construcción de sistemas escalables y tolerantes a fallos.
Nuestro worker está implementado por Node.js que realiza trabajos de forma asíncrona. Podemos realizar un resumen función a función de lo que realiza cada una de ellas:

- **Imports:** Importa las dependencias necesarias, incluyendo connect y NatsConnection de la biblioteca nats, así como algunos módulos locales como JetstreamHandler, ejecutarScriptShell.
- **Clase Server**: Define una clase llamada Server que maneja la conexión y la ejecución de trabajos.
- **Constructor**: El constructor recibe una URL y un puerto, establece la URL completa para la conexión, inicializa algunos valores y llama a la función connect() para establecer la conexión con el servidor NATS.

- **Método connect**: Intenta conectarse al servidor NATS usando la URL proporcionada. Si la conexión tiene éxito, instancia un JetstreamHandler para manejar las operaciones con Jetstream.
- **Método jetstream**: Este método se encarga de publicar un mensaje en Jetstream con un ID y un estado específico para guardar los estados del trabajo correspondientes en el KV store de nats.
- **Método storeTrabajo**: Almacena el resultado de un trabajo en el ObjectStore de nats.
- **Método listener**: Establece un suscriptor para el tema 'work' en NATS. Cuando recibe un mensaje en este tema, lo decodifica, ejecuta la función asociada al trabajo y luego almacena el resultado.
- **Método ejecutarFuncion**: Este método ejecuta la función asociada al trabajo. En este caso, ejecuta un script de shell alojado en un repositorio de GitHub y almacena el resultado.

El proceso técnico previamente expuesto describe el funcionamiento esencial del sistema. A continuación, procederemos a detallar las funcionalidades específicas. En el frontend (API), se establece una conexión mediante una cola basada en NATS con el worker. Esta conexión se realiza a través de una suscripción, mediante la cual el worker recibe la información necesaria para llevar a cabo la tarea asignada. Dentro de este flujo operativo, se emplea un script denominado "script.sh" con la función de clonar el proyecto desde el repositorio de GitHub y ejecutar el script principal ("main.py") contenido en el mismo. Los resultados generados por la ejecución del script se almacenan en el Object Store provisto por NATS.
Por tanto hemos realizado un worker que se encarga de recibir mensajes de trabajos a través de NATS, ejecutar las tareas asociadas a esos trabajos, y luego almacenar y notificar el resultado a través de Jetstream.

2.4. NATS

NATS es un sistema de mensajería de código abierto que potencia los sistemas distribuidos, nace para dar respuesta a la dificultad que conllevaba desarrollar e implementar aplicaciones y servicios que se comunican entre sí en sistemas distribuidos. En consecuencia, es responsable de direccionar, descubrir e intercambiar mensajes que impulsan los patrones comunes en los sistemas distribuidos, haciendo y respondiendo preguntas, también conocidos como servicios/microservicios.
NATS incorpora un sistema de persistencia distribuida denominado Jetstream. Jetstream permite nuevas funcionalidades y una mayor durabilidad y persistencia sobre las funcionalidades básicas de NATS, además está integrado en el servidor NATS.
NATS se considera bastante fácil de usar y admite múltiples casos de uso y patrones de mensajes, como pub/sub (publicación/suscripción) y solicitud-respuesta [6].

Algunas de las características que podríamos destacar de NATS es que es un sistema fácil de usar, tiene un alto rendimiento, es extremadamente ligero, ofrece soporte para servicios observables y escalables y flujos de datos/eventos[7].

Figura 2.3. Logo NATS

Hacemos uso exhaustivo de las diversas funcionalidades ofrecidas por NATS. Utilizamos su KV Store para almacenar los estados de los trabajos, su Object Store para preservar los resultados obtenidos. Además, empleamos una cola para la transmisión eficiente de tareas desde el Frontend (API) hacia el worker o workers. Asimismo, aprovechamos los mensajes para notificar al observador tanto el inicio como la finalización de cada tarea y en el momento de obtener las métricas, del promedio de las diferencias del tiempo de los trabajos, desde Fe(api) le pedimos las métricas al observer.

2.5. OBSERVER

El observer en nuestro caso podríamos definirlo como una función que hemos diseñado para observar y registrar todo lo que está pasando en el sistema, como por ejemplo la ejecución de una función, tiempos de respuesta o incluso errores, el código implementado lo hemos construido de la siguiente manera:

- **Importaciones:** Se importan las bibliotecas necesarias, como axios para hacer solicitudes HTTP y dotenv para cargar variables de entorno desde un archivo .env. También se importa la clase Client que está en la ruta ./socket/nats, que es parte de una implementación de cliente NATS para la comunicación.
- **Definición de tipos**: Se define la interfaz WorkerData para representar los datos de inicio y fin de un trabajo.
- **Variables globales.**
- **workerData**: Un objeto que almacena los datos de inicio y fin de cada trabajo.
- **workerInExce**: Un array que almacena los IDs de los trabajos en ejecución.
- **Función calcularMediaDiferencia**: Calcula la media de la diferencia entre las horas de inicio y fin de los trabajos almacenados en workerData.
- **Función estadoCola**: Gestiona el estado de la cola de trabajos. Cuando recibe un mensaje de inicio de un trabajo, registra la hora de inicio y añade el ID del trabajo al array workerInExce. Si hay más trabajos en ejecución de lo permitido según la variable de entorno NUM_PROCESOS_IN_QUEUE_FOR OTHER WORKER, puede tomar alguna acción, como agregar más workers. Cuando recibe un mensaje de finalización de un trabajo, registra la hora de finalización y elimina el trabajo del array workerInExce.
- **Función main**: Inicializa el cliente NATS con la URL y el puerto obtenidos de las variables de entorno. Pasa las funciones estadoCola y calcularMediaDiferencia al constructor del cliente.

Podemos ver que la funcionalidad principal es la gestión del estado de la cola de trabajos y el cálculo de métricas de rendimiento como la media de la duración de los trabajos.

2.6. WORKERSTATUS

Esta clase maneja el estado y los resultados de trabajos a través de una API RESTful utilizando Express [8] y NATS JetStream[9] , lo hemos construido de la siguiente manera :

- **Importaciones**:Importa las herramientas necesarias de NATS y Express.
- **Clase jobStatus:**Esta clase se encarga de gestionar el estado y los resultados de los trabajos.Tiene propiedades privadas para la conexión NATS (nc), la URL y el puerto, un códec para las cadenas (sc), un manejador de JetStream (jetstreamHandler) y una aplicación Express (app).
- **Constructor**: Recibe una URL y un puerto para conectar NATS e inicializa las propiedades y llama al método connect().
- **Método connect**:Intenta conectar a NATS utilizando la URL y el puerto proporcionados.Si hay un error, lo registra en la consola.Una vez conectado, instancia el jetstreamHandler y llama a setupExpress.
- **Método setupExpress**: Configura las rutas de la API REST.
  Define tres rutas:
  - /getworkstatus/:workid: Para obtener el estado de un trabajo por su ID.
  - /getworkresult/:workid/:userid: Para obtener el resultado de un trabajo para un usuario específico.
  - /getworkresult/:userid: Para obtener todos los resultados de trabajos para un usuario específico.

  Cada ruta llama a métodos del jetstreamHandler para manejar las solicitudes y enviar respuestas adecuadas.
  Por último, inicia el servidor Express en el puerto especificado por la variable de entorno PORT_API

2.7. DOCKER

Docker ha revolucionado la forma en que desarrolladores y equipos de operaciones despliegan y gestionan sus aplicaciones. Es una plataforma de código abierto que permite empaquetar, distribuir y ejecutar aplicaciones en contenedores ligeros y portátiles. Estos contenedores encapsulan todo lo necesario para que una aplicación se ejecute de manera consistente en cualquier entorno, desde el desarrollo hasta la producción.

En nuestro proyecto, adoptamos Docker como una herramienta fundamental. Para cada uno de nuestros microservicios, como Frontend, Worker, Observer y Worker Status, creamos un archivo Dockerfile dedicado. Posteriormente, utilizando un archivo docker-compose.yml, integramos todos estos servicios para facilitar su despliegue uniforme en cualquier entorno, independientemente del dispositivo utilizado para ejecutar el proyecto. Esta estrategia nos permite gestionar eficientemente los componentes de nuestra aplicación de manera modular y escalable, garantizando una consistencia en el despliegue que optimiza nuestro proceso de desarrollo y operaciones.

3. IMPLEMENTACIÓN DEL PROBLEMA

El diseño de la arquitectura que se ha llevado a cabo es el siguiente:

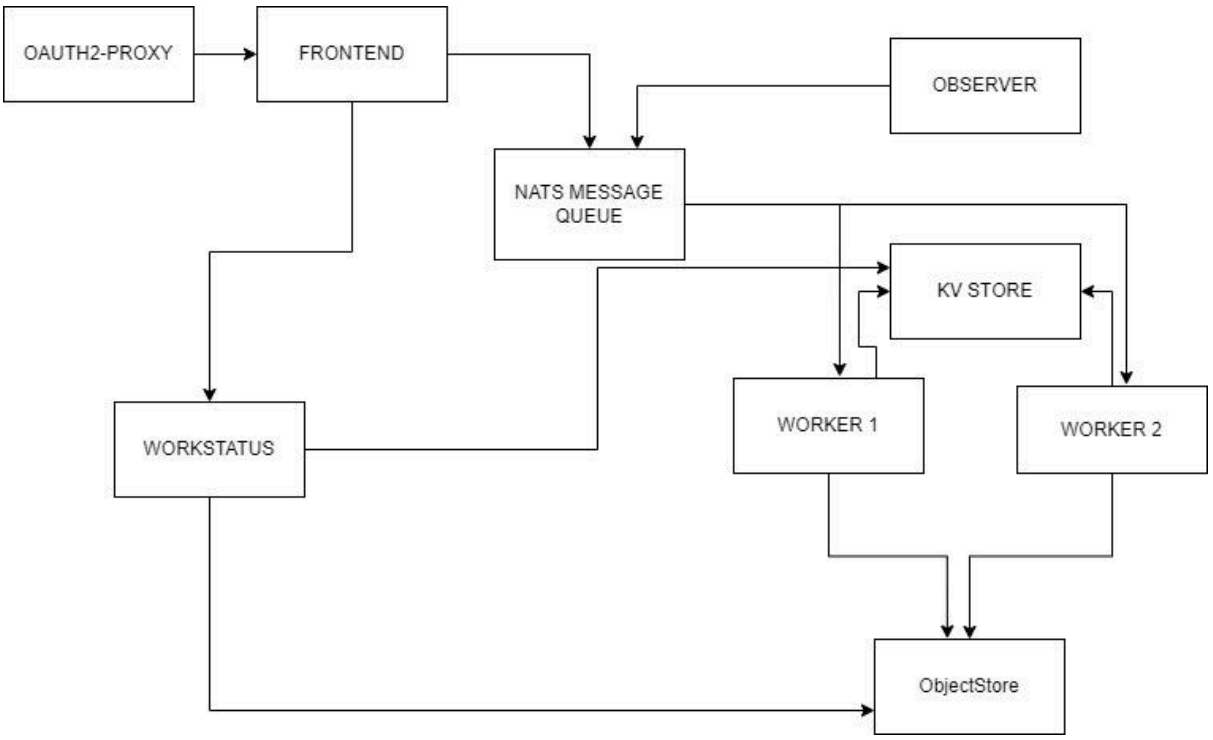


Figura 3.1. Diseño del problema

3.1. CASOS DE USO

El escenario real es el siguiente :

1. **Usuario se conecta a través del OAuth2-Proxy**: El usuario auténtica su acceso al sistema a través del OAuth2-Proxy, lo que garantiza la seguridad y la gestión adecuada de las credenciales de usuario.
2. **Usuario envía un URL de GitHub:** Después de autenticarse, el usuario proporciona un URL de un repositorio de GitHub que contiene un script que desea ejecutar como un trabajo en el sistema.
3. **Frontend genera un ID para el trabajo**: El frontend genera un identificador único (ID) para el trabajo que se va a ejecutar. Este ID se utilizará para identificar y rastrear el trabajo a lo largo del sistema.
4. **Frontend envía un mensaje JSON al servidor NATS**: El frontend construye un mensaje JSON que contiene el ID generado y el URL de GitHub proporcionado por el usuario. Este mensaje se envía al servidor NATS para ser distribuido a los workers que están suscritos al tema "work".
5. **NATS envía el mensaje a los workers suscritos al tema "work"**:El servidor NATS recibe el mensaje del frontend y lo distribuye a los workers que están suscritos al tema "work".
6. **El Worker recibe el mensaje y ejecuta el script desde el repositorio de GitHub**: El worker recibe un mensaje del servidor NATS que contiene el ID del trabajo y la URL del repositorio de GitHub. Utilizando la URL proporcionada en el mensaje, el worker descarga el script desde el repositorio de GitHub. Antes de ejecutar el trabajo, el worker registra el estado del trabajo como "en ejecución" en un almacén de clave-valor (KV store). El worker ejecuta el script descargado localmente. Después de la ejecución del trabajo, el worker actualiza el estado del trabajo según

el resultado de la ejecución. Posteriormente, el worker registra el estado actualizado del trabajo en el almacén de clave-valor (KV store). Una vez que se ha completado la ejecución del trabajo y se ha obtenido un resultado, el worker guarda este resultado en un almacén de objetos (object store), y también avisa al observador que ha terminado su trabajo.

Además, el sistema proporciona una API para que los usuarios puedan obtener el estado y el resultado de sus trabajos.

4. CONCLUSIÓN

Este trabajo presenta una exitosa implementación de un sistema Function as a Service (FaaS), utilizando Node.js como lenguaje principal de programación. Se ejecutó un proyecto de GitHub que incluía archivos Python y se empleó NATS Object Store para proporcionar un sistema de almacenamiento altamente escalable y distribuido, garantizando así una alta disponibilidad y durabilidad de los datos. Además, se implementó un KV Store para gestionar de manera eficiente la persistencia de datos clave, como los estados intermedios de procesamiento, lo que facilitó la gestión y el acceso rápido a los datos.

La adopción de un Sistema de Función como Servicio, empleando Node.js, NATS Object Store y KV, demuestra una solución efectiva y escalable. Este enfoque ofrece una arquitectura flexible y rentable para el desarrollo y despliegue de aplicaciones modernas basadas en microservicios.

5. TRABAJO FUTURO

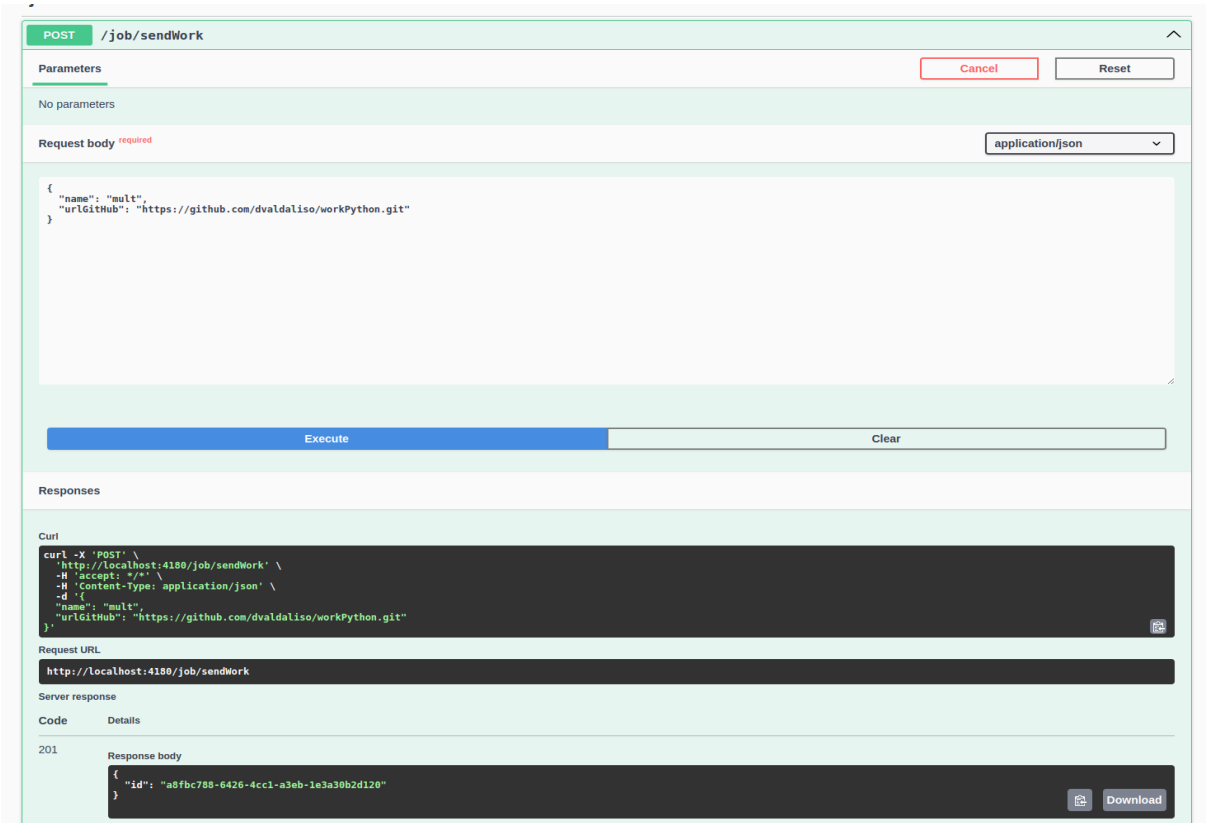
Para futuros proyectos, podemos mejorar al integrar una gama más amplia de métricas para aumentar la capacidad de los trabajadores. Sería beneficioso contemplar la activación automática de un nuevo trabajador al recibir una notificación del observador.

6. BIBLIOGRAFÍA

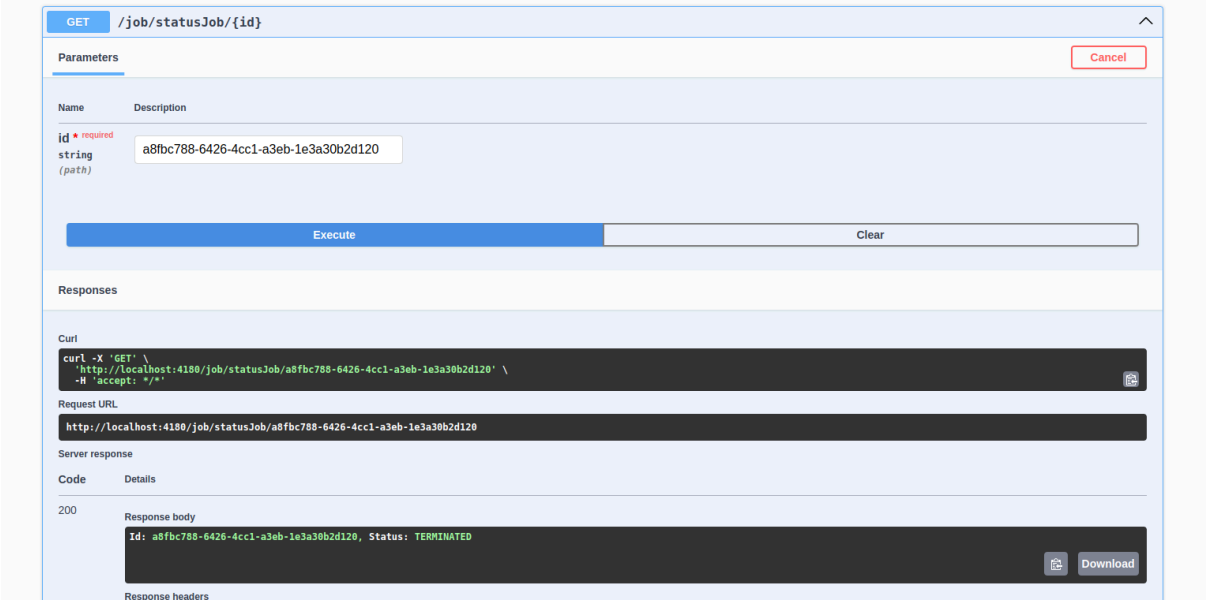
[1] https://www.redhat.com/es/topics/cloud-native-apps/what-is-faas
[2] https://www.ibm.com/es-es/topics/faas
[3] https://kinsta.com/es/blog/funcion-como-servicio/
[4] https://blog.hubspot.es/website/que-es-api-rest
[5] https://gfourmis.co/nestjs-que-es-y-por-que-empezar-a-usarlo/
[6] https://memphis.dev/blog/comparing-nats-and-kafka-understanding-the-differences/
[7] https://benjagarrido.com/nats-mensajeria-de-codigo-abierto/
[8] https://expressjs.com/
[9] https://docs.nats.io/nats-concepts/jetstream

7. ANEXOS

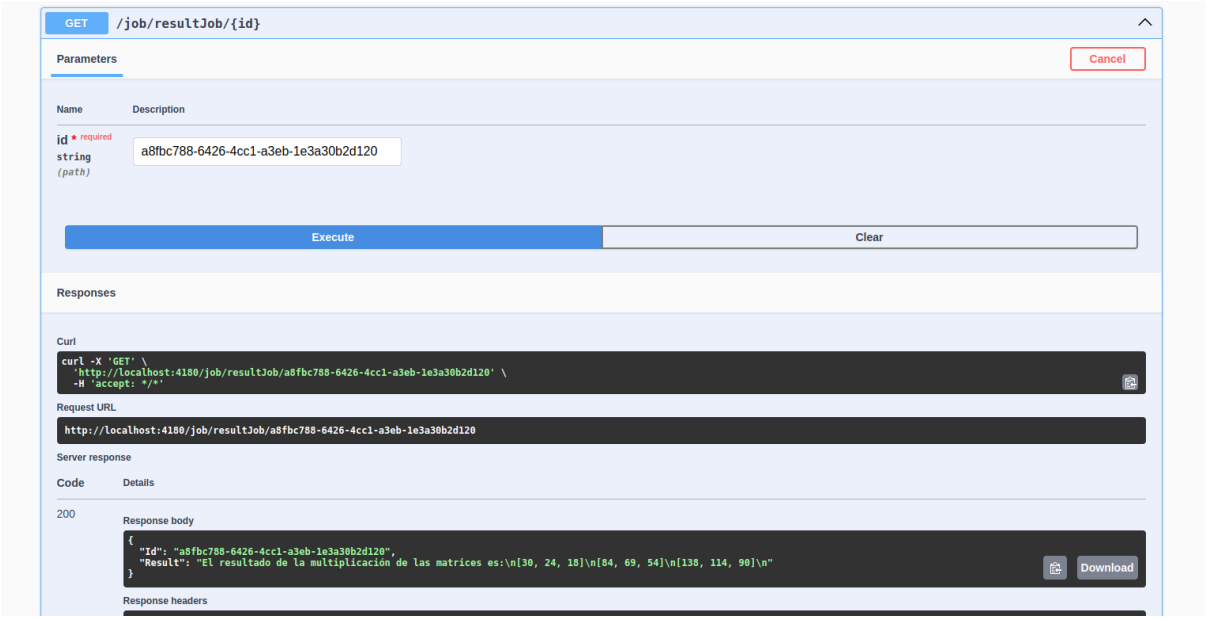
Envio de trabajo, responde con el id del trabajo



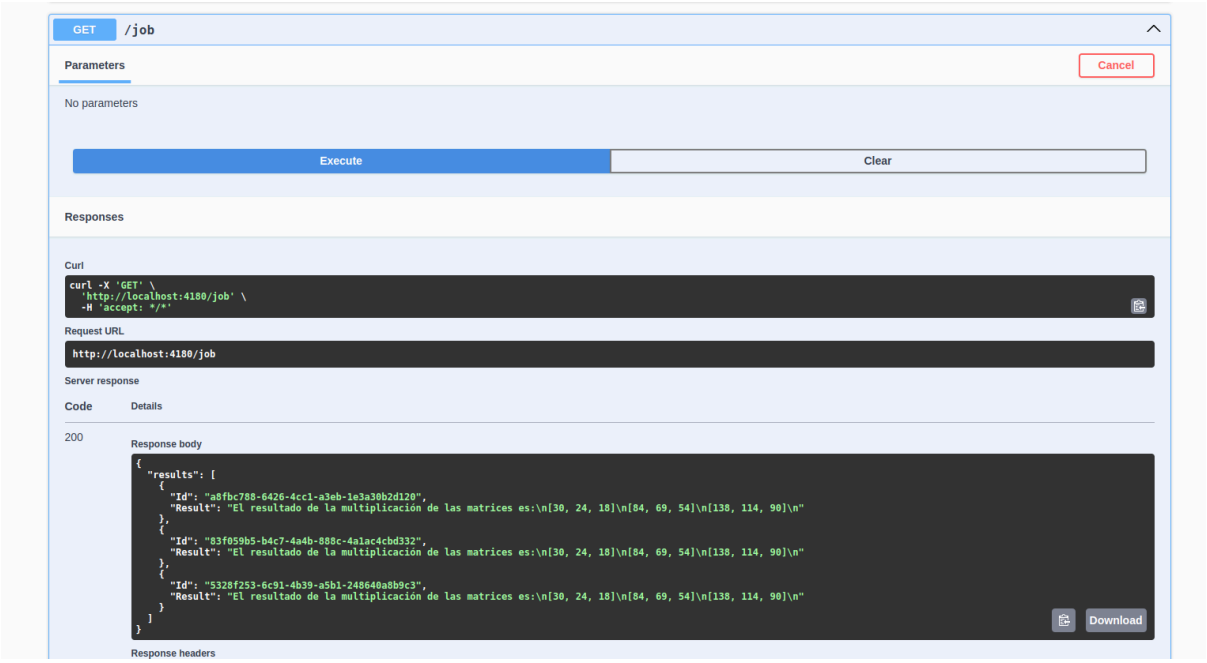
Estado de un trabajo



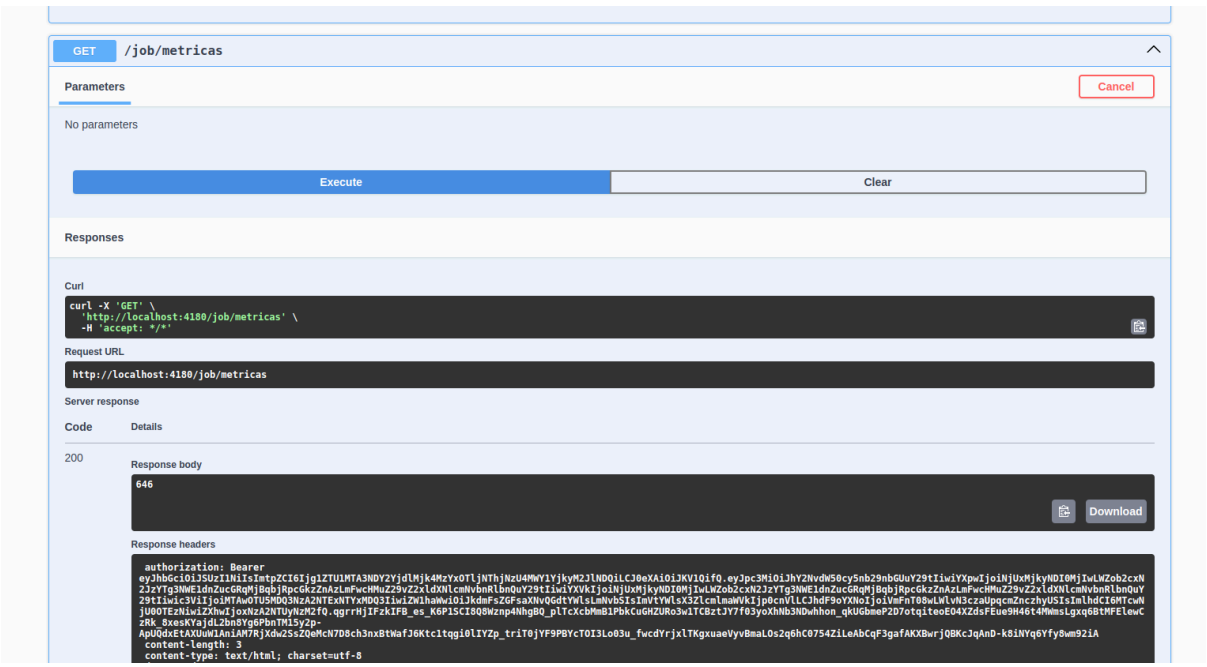
Resultado de un trabajo:



Listado de trabajos hechos por el usuario



Metricas


GET
/job/metricas
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://localhost:4180/job/metricas' \
-H 'accept: */*'
Request URL
http://localhost:4180/job/metricas
Server response
Code
Details
200
Response body
646
Download
Response headers
authorization: Bearer
eyJhbGciOiJSUzI1NiIsImtpZCI6Ijg1ZTU1MTA3NDY2YjdlMjk4MzYxOTljNThjNzU4MWY1YjkyM2JlNDQiLCJ0eXAiOiJKV1QifQ.eyJpc3MiOiJhY2NvdW50cy5nb29nbGUuY29tIiwiYXpwIjoiNjUxMjkyNDI0MjIwLWZob2cxN
2JzYTg3NWE1dnZucGRqMjBqbjRpcGkzZnAzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiwiYXVkIjoiNjUxMjkyNDI0MjIwLWZob2cxN2JzYTg3NWE1dnZucGRqMjBqbjRpcGkzZnAzLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY
29tIiwic3ViIjoiMTAwOTU5MDQ3NzA2NTExNTYxMDQ3IiwiZW1haWwiOiJkdmFsZGFsaXNvQGdtYWlsLmNvbSIsImVtYWlsX3ZlcmlmaWVkIjp0cnVlLCJhdF9oYXNoIjoiVmFnT08wLWlvN3czaUpqcmZnczhyUSIsImlhdCI6MTcwN
jU0OTEzNiwiZXhwIjoxNzA2NTUyNzM2fQ.qgrrHjIFzkIFB_es_K6P1SCI8Q8Wznp4NhgBQ_plTcXcbMmB1PbkCuGHZURo3w1TCBztJY7f03yoXhNb3NDwhhon_qkUGbmeP2D7otqiteoEO4XZdsFEue9H46t4MWmsLgxq6BtMFElewC
zRk_8xesKYajdL2bn8Yg6PbnTM15y2p-
ApUQdxEtAXUuW1AniAM7RjXdw2SsZQeMcN7D8ch3nxBtWafJ6Ktc1tqgi0lIYZp_triT0jYF9PBYcTOI3Lo03u_fwcdYrjxlTKgxuaeVyvBmaLOs2q6hC0754ZiLeAbCqF3gafAKXBwrjQBKcJqAnD-k8iNYq6Yfy8wm92iA
content-length: 3
content-type: text/html; charset=utf-8