UNIVERSITAT POLITÉCNICA DE VALENCIA

**Departamento**

**de**

**Sistemas Informáticos y Computación**



CLOUD COMPUTING

Trabajo Final de Asignatura

“Proyecto FaaS”

Alumnos:

Juan Amilcar Ttito Ttito

Chunlu Hu

Carlos Caramagna

Profesor:

José Manuel Bernabeu Aubán

Valencia, 2025

ÍNDICE

1. Introducción…………………………………………………………………………………………..…………… Pag 1

1.1. Descripción general del proyecto………………………………………………………………..………………. Pag 2

1.2. Objetivos del Sistema………………………………………………………………………………..………………… Pag 3

1.3. Beneficios de usar FaaS con Go, NATS, Redis y JWT………………………………..…………………… Pag 4

2. Arquitectura del Sistema……………………………………………………………………….…………………… Pag 6

2.1. Diagrama de arquitectura……………………………………………………………………..……………………. Pag 8

2.2. Componentes principales…………………………………………………………………..………………………. Pag 9

2.2.1. Funciones como servicio (FaaS)……………………………………………………..……………………….. Pag 9

2.2.2. Sistema de mensajería (NATS)………………………………………………………………………………. Pag 12

2.2.3. Almacenamiento en memoria (Redis)…………………………………………………………………… Pag 13

2.2.4. Autenticación y autorización (JWT)………………………………………………………………………. Pag 14

2.3. Flujo de datos entre components……………………………………………………………………………. Pag 17

3. Instalación y Configuración……………………………………………………………………………... Pag 17

3.1. Prerrequisitos…………………………………………………………………………………………………………… Pag 17

3.1.1. Entorno de desarrollo (Go, Docker, etc.)……………………………………………………………….. Pag 18

3.1.2. Instalación de NATS y Redis…………………………………………………………………………………… Pag 19

3.2. Configuración del proyecto………………………………………………………………………………………. Pag 20

3.2.1. Archivo de configuración (config.yaml)…………………………………………………………………. Pag 20

4. Estructura del Proyecto……………………….…………………………………………………………… Pag 21

4.1 Descripción de carpetas y archivos……………………………………………………………………………. Pag 21

4.1.1. internal/auth: Gestión de JWT………………………………………………………………………………. Pag 21

4.1.2. internal/messaging: Conexión y manejo de NATS………………………………………………….. Pag 23

4.1.3. internal/storage: Integración con Redis…………………………………………………………………. Pag 23

5. Implementación de Funciones………………………………………………………………………….. Pag 23

5.1. Cómo manejar eventos de NATS……………………………………………………………………………….. Pag 23

5.2. Uso de Redis para almacenamiento temporal…………………………………………………………… Pag 23

6. Despliegue……………………………………………………………………………………………………… Pag 23

6.1. Entorno de producción…………………………………………………………………………………………… Pag 23

6.2. Uso de Docker………………………………………………………………………………………………………… Pag 23

6.3. Estrategias de despliegue continuo (CI/CD)……………………………………………………………. Pag 24

6.4. Escalabilidad de las funciones………………………………………………………………………………… Pag 25

7. Documentación oficial de herramientas utilizadas…………………………………………. Pag 27

1. Introducción

El servicio FaaS (Function as a Service) forma parte de la evolución de la computación en la nube; un modelo que ha transformado profundamente la manera en que las organizaciones gestionan sus aplicaciones y recursos tecnológicos.

FaaS se engloba dentro del concepto más amplio de la arquitectura sin servidor (serverless), que se comenzó a popularizar a mediados de la década de 2010. Aunque el concepto de la computación en la nube ya existía desde los años 2000, fue durante esa década cuando se consolidaron nuevas formas de interactuar con los recursos informáticos sin necesidad de gestionar servidores físicos, permitiendo a las empresas centrarse más en el desarrollo de sus aplicaciones que en la infraestructura subyacente.

FaaS, en particular, permite a los desarrolladores ejecutar funciones individuales en la nube sin tener que preocuparse por el manejo de servidores o la infraestructura. Este modelo se distingue por su capacidad para ejecutar código en respuesta a eventos, escalando automáticamente según la demanda y sin necesidad de que el usuario gestione la infraestructura. El primer servicio de FaaS de gran adopción fue AWS Lambda, lanzado por Amazon Web Services en 2014, que permitió a los desarrolladores ejecutar funciones específicas en la nube bajo demanda, de manera eficiente y rentable.

A diferencia de otros modelos de computación en la nube como IaaS (Infrastructure as a Service) o PaaS (Platform as a Service), FaaS no requiere que los usuarios gestionen máquinas virtuales o contenedores. En cambio, se basa en un enfoque más granular donde cada función es autónoma, y se ejecuta solo cuando es invocada, optimizando recursos y reduciendo costos operativos.

Las bondades de utilizar FaaS son diversas. En primer lugar, su modelo de pago por uso permite a las empresas reducir costos significativamente, ya que solo se paga por el tiempo en que las funciones están activas. Además, el escalado automático es una ventaja crucial, ya que permite que las aplicaciones gestionen picos de demanda sin intervención manual o previsión de recursos. Por otro lado, FaaS promueve una mayor agilidad en el desarrollo de software, ya que los equipos de desarrollo pueden concentrarse en crear funcionalidades específicas sin preocuparse por la infraestructura. Esta simplificación no solo mejora la eficiencia operativa, sino que también acelera el tiempo de comercialización de nuevas aplicaciones.

Por último, FaaS contribuye a una mayor resiliencia y fiabilidad de las aplicaciones. Las funciones pueden ser distribuidas por múltiples ubicaciones geográficas y los sistemas se vuelven más robustos frente a fallos. Combinado esto con la capacidad de actualizar y desplegar funciones sin afectar otras partes del sistema, otorga una gran flexibilidad y control.

FaaS representa un avance significativo en la computación en la nube, permitiendo a las empresas y desarrolladores una experiencia más eficiente, escalable y económica.

Al integrar este modelo, las organizaciones pueden optimizar recursos y mejorar su capacidad de respuesta ante las demandas del mercado, posicionándose a la vanguardia de la tecnología moderna.

* 1. Descripción general del proyecto

El proyecto presentado es una aplicación en Go que implementa un sistema para gestionar funciones registradas en una base de datos Redis, a través de una serie de APIs RESTful. Este sistema permite registrar, desregistrar y llamar a funciones almacenadas en Redis, lo cual puede ser útil en contextos donde se necesite almacenar y ejecutar código o funcionalidades dinámicas de manera eficiente.

El proyecto consta de tres endpoints principales:

1. **Registro de función (/api/registrafuncion)**: Este endpoint permite registrar una nueva función en la base de datos Redis. La función se representa como un objeto JSON que contiene tres campos: usuario, funcion y codigo. Al recibir la solicitud, el servidor decodifica el JSON y utiliza la librería de Redis para almacenar el código de la función bajo una clave única correspondiente al nombre de la función. Si la función se registra con éxito, Redis devuelve un resultado indicando si se añadió como una nueva entrada o si sobrescribió una existente.
2. **Desregistro de función (/api/desregistrafuncion)**: Este endpoint permite eliminar una función previamente registrada en Redis. Similar al endpoint anterior, recibe un objeto JSON con los detalles de la función a eliminar y, mediante el comando HDel, la remueve de la base de datos. Si la función no existe en Redis, se devuelve un mensaje que indica que no se encontró ninguna función para desregistrar.
3. **Llamado de función (/api/llamarfuncion)**: El último endpoint permite "llamar" a una función registrada, lo que implica recuperar su código de Redis y devolverlo como respuesta. Primero, el sistema verifica si la función solicitada existe en la base de datos Redis. Si la función no está registrada, se envía un mensaje de error; si está disponible, se retorna el código asociado a esa función, lo que simula la ejecución de la misma.

Para interactuar con la base de datos Redis, se utiliza la librería go-redis, que proporciona una API sencilla y eficiente para interactuar con Redis desde Go. El servidor HTTP se ejecuta en el puerto 8001 y maneja las solicitudes utilizando las funciones registerFunction, unregisterFunction y llamarFunction para los respectivos endpoints.

Las principales bondades de este sistema incluyen la capacidad de almacenar y gestionar funciones dinámicamente sin la necesidad de persistirlas en archivos o bases de datos tradicionales. Esto hace que el sistema sea altamente flexible, permitiendo ejecutar funciones bajo demanda. Además, al utilizar Redis, el proyecto se beneficia de la rapidez y eficiencia de este sistema de almacenamiento en memoria, lo que asegura un acceso rápido y una gestión de datos eficiente. Este enfoque también facilita el desarrollo de aplicaciones modulares, donde las funciones pueden ser fácilmente registradas, desregistradas y ejecutadas en función de la necesidad del usuario, todo en tiempo real.

En términos de escalabilidad y mantenimiento, la solución es flexible, ya que el servidor de Redis puede ser distribuido y replicado para mejorar la disponibilidad y rendimiento. Además, el uso de Go para el desarrollo del servidor garantiza una ejecución eficiente y un manejo adecuado de concurrencia.

Este proyecto proporciona una interfaz sencilla para gestionar funciones registradas en Redis mediante una API RESTful, aprovechando las ventajas de la arquitectura sin servidor y el uso de Redis como sistema de almacenamiento rápido. Este enfoque puede aplicarse en una amplia variedad de aplicaciones que necesiten ejecutar código de manera dinámica y eficiente.

* 1. Objetivos del Sistema

El sistema presentado tiene como objetivo principal proporcionar una plataforma sencilla y eficiente para gestionar funciones registradas en una base de datos Redis mediante una API RESTful. Los objetivos específicos del sistema incluyen:

1. **Registrar Funciones de Manera Dinámica**: Permitir a los usuarios registrar nuevas funciones en el sistema, almacenando el código correspondiente en Redis. Esto facilita la gestión de funciones que pueden ser ejecutadas bajo demanda sin la necesidad de codificarlas directamente en el servidor.
2. **Desregistrar Funciones Existentes**: Proporcionar una interfaz para eliminar funciones previamente registradas en el sistema. Esto asegura que el sistema se mantenga actualizado y limpio, eliminando funciones obsoletas o innecesarias.
3. **Ejecutar Funciones Registradas**: Permitir que los usuarios llamen a funciones previamente registradas, recuperando el código almacenado en Redis. Esto simula la ejecución de la función solicitada, proporcionando acceso a las funcionalidades de manera dinámica.
4. **Facilitar la Integración con Redis**: Integrar el sistema con Redis como un almacenamiento de datos rápido y eficiente, aprovechando sus capacidades de manejo de datos en memoria para garantizar un acceso rápido y eficiente a las funciones registradas.
5. **Proveer una API RESTful Simple**: Ofrecer una API RESTful sencilla y bien definida para interactuar con el sistema, permitiendo que los usuarios puedan registrar, desregistrar y llamar funciones de manera clara y estructurada a través de solicitudes HTTP.
6. **Mejorar la Flexibilidad y Escalabilidad**: Proporcionar un sistema flexible y escalable que permita a los usuarios gestionar funciones de forma independiente, optimizando el rendimiento del sistema mediante el uso de Redis para manejar grandes cantidades de datos de manera eficiente.
7. **Optimizar el Rendimiento de la Ejecución de Funciones**: Al utilizar Redis como backend, el sistema asegura tiempos de respuesta rápidos para las operaciones de registro, eliminación y ejecución de funciones, lo que mejora la experiencia del usuario al interactuar con el sistema.

Proporciona una solución ágil y eficiente para la gestión de funciones dinámicas, permitiendo a los usuarios registrar, desregistrar y ejecutar funciones con facilidad, todo respaldado por la rapidez y eficiencia de Redis como sistema de almacenamiento.

* 1. Beneficios de usar FaaS con GO, NATS, Redis y JWT

El uso de **FaaS** (Function as a Service) combinado con tecnologías como **Go**, **NATS**, **Redis** y **JWT** en el proyecto presentado aporta una serie de beneficios clave que optimizan el rendimiento, la escalabilidad y la seguridad del sistema.

A continuación, se detallan los beneficios específicos de cada tecnología utilizada:

1. **Go (Lenguaje de Programación)**:
   * **Eficiencia y Rendimiento**: Go es conocido por su alta eficiencia y rapidez en la ejecución, lo cual es ideal para la creación de aplicaciones escalables y de alto rendimiento como las basadas en FaaS. La concurrencia nativa de Go (a través de goroutines) facilita la gestión eficiente de múltiples solicitudes simultáneas, lo que es crucial en un entorno de funciones como servicio.
   * **Simplicidad en el Desarrollo**: Go es un lenguaje sencillo y fácil de aprender, lo que permite a los desarrolladores centrarse en la lógica de negocio en lugar de lidiar con complejidades del lenguaje. Esto acelera el tiempo de desarrollo y reduce errores.
   * **Desempeño Optimizado**: Gracias a su compilación en código nativo y a su eficiencia en el manejo de recursos, Go es ideal para el desarrollo de aplicaciones con alta demanda de rendimiento, como las que se ejecutan en arquitecturas sin servidor.
2. **NATS (Mensajería en Tiempo Real)**:
   * **Alta Escalabilidad y Desempeño**: NATS es una solución de mensajería ligera y de alto rendimiento que facilita la comunicación en tiempo real entre microservicios o funciones en una arquitectura FaaS. Al integrar NATS, el sistema puede escalar de manera eficiente, manejando grandes volúmenes de tráfico de manera simultánea sin comprometer el rendimiento.
   * **Desacoplamiento de Componentes**: NATS permite desacoplar los diferentes servicios dentro de la arquitectura, lo que mejora la flexibilidad y facilita la administración del sistema. Las funciones pueden operar de manera independiente, mejorando la capacidad de respuesta y la resiliencia.
   * **Facilidad de Integración**: NATS facilita la integración con otros servicios de mensajería y con sistemas distribuidos, lo que es crucial en aplicaciones modernas que requieren alta disponibilidad y baja latencia.
3. **Redis (Almacenamiento en Memoria Rápido)**:
   * **Velocidad de Acceso a Datos**: Redis, al ser una base de datos en memoria, proporciona un acceso extremadamente rápido a los datos. Esto es fundamental en el contexto de FaaS, donde la rapidez en la ejecución de funciones y la recuperación de datos son cruciales para ofrecer una experiencia de usuario óptima.
   * **Escalabilidad**: Redis es altamente escalable, lo que permite manejar grandes volúmenes de datos distribuidos de manera eficiente. En el proyecto, se utiliza para almacenar funciones registradas y sus respectivos códigos, lo que garantiza un rendimiento constante incluso a medida que el sistema crece.
   * **Simplicidad y Flexibilidad**: Redis ofrece una variedad de estructuras de datos que permiten almacenar de manera flexible la información, y su fácil integración con Go hace que sea una opción ideal para manejar datos temporales en sistemas dinámicos.
4. **JWT (JSON Web Token) para Autenticación y Autorización**:
   * **Seguridad**: JWT se utiliza para asegurar la autenticación y autorización de los usuarios en el sistema. Permite emitir tokens firmados que verifican la identidad del usuario, asegurando que solo los usuarios autorizados puedan registrar, desregistrar o ejecutar funciones.
   * **Escalabilidad y Descentralización**: JWT no requiere mantener sesiones en el servidor, lo que lo convierte en una excelente opción para arquitecturas distribuidas y sistemas sin servidor. Esto reduce la carga en los servidores y mejora la escalabilidad del sistema, ya que cada solicitud puede ser validada independientemente sin necesidad de acceder a una base de datos central para verificar las credenciales.
   * **Interoperabilidad**: Al ser un estándar abierto, JWT es fácilmente integrable con otras aplicaciones y servicios, lo que lo convierte en una solución ideal para sistemas que necesitan comunicarse de manera segura con diferentes servicios o aplicaciones.

La combinación de **Go**, **NATS**, **Redis** y **JWT** en un sistema basado en **FaaS** proporciona una serie de ventajas, incluyendo eficiencia, escalabilidad, rapidez en el acceso a datos y seguridad. El uso de **Go** asegura un alto rendimiento en la ejecución de las funciones, mientras que **NATS** facilita la mensajería eficiente entre servicios. **Redis** garantiza un acceso rápido a los datos almacenados, y **JWT** proporciona una capa de seguridad robusta para la autenticación y autorización. Juntas, estas tecnologías permiten desarrollar un sistema flexible, escalable y altamente eficiente que puede satisfacer las necesidades de aplicaciones dinámicas y de alto rendimiento.

2. Arquitectura del Sistema

El sistema presentado está basado en una arquitectura distribuida que utiliza varios servicios interconectados para proporcionar una solución escalable, eficiente y modular. Esta arquitectura está compuesta por varios microservicios, que operan en contenedores Docker, y se comunican entre sí utilizando herramientas como **Redis** para almacenamiento en memoria, **NATS** para mensajería en tiempo real, y varios otros componentes que ejecutan funciones dinámicas. A continuación se describen los componentes claves y su interacción.

#### **a. Contenedores y Servicios**

El sistema se implementa utilizando **Docker** y está compuesto por cinco servicios principales, que son gestionados a través de un archivo docker-compose.yml. Los servicios están diseñados para funcionar de manera independiente, pero también están interconectados para cumplir con la funcionalidad general del sistema. Los servicios son:

1. **auth**: Este servicio es responsable de la autenticación y autorización de los usuarios. Utiliza JSON Web Tokens (JWT) para gestionar la autenticación, lo que garantiza que solo los usuarios autorizados puedan acceder a las funcionalidades del sistema. El servicio auth genera tokens JWT cuando un usuario se registra o inicia sesión, y estos tokens se utilizan para validar las solicitudes en los demás servicios. El servicio escucha en el puerto 8000 y depende de Redis para almacenar los usuarios registrados y sus contraseñas cifradas.
2. **faas**: Este servicio es el núcleo del sistema, gestionando las funciones como servicio (FaaS). Permite registrar, desregistrar y ejecutar funciones dinámicas que pueden ser invocadas por otros componentes del sistema. El servicio faas también interactúa con Redis, donde las funciones y sus respectivos códigos son almacenados, y con NATS para la comunicación en tiempo real y la gestión de las invocaciones de funciones. Este servicio escucha en el puerto 8001.
3. **orquestador**: El servicio orquestador se encarga de coordinar y gestionar las invocaciones de funciones a través de los diferentes servicios. Sirve como un controlador central que orquesta el flujo de trabajo entre los servicios y asegura que las solicitudes sean procesadas de manera adecuada. Este servicio también se comunica con Redis y NATS, y depende de ambos para acceder a datos y para la comunicación en tiempo real entre servicios. El orquestador está expuesto en el puerto 8002.
4. **worker**: El servicio worker ejecuta las tareas específicas relacionadas con la ejecución de las funciones registradas. Este servicio es responsable de escuchar los mensajes enviados por el servicio faas a través de NATS, ejecutar las funciones requeridas y devolver los resultados. La arquitectura basada en workers permite que el sistema maneje de manera eficiente la ejecución de múltiples tareas en paralelo, mejorando el rendimiento y la escalabilidad del sistema. Este servicio depende exclusivamente de NATS para la mensajería y no tiene un puerto expuesto directamente, ya que su función es interna.
5. **redis**: Redis es una base de datos en memoria de alto rendimiento que se utiliza para almacenar datos temporales, como las funciones registradas y los usuarios autenticados. Redis es utilizado por todos los servicios (auth, faas, orquestador) para acceder y almacenar datos de manera rápida. Este servicio escucha en el puerto 6379 y tiene una configuración personalizada a través de un archivo de configuración montado desde el host.
6. **nats**: NATS es un sistema de mensajería de alto rendimiento que permite la comunicación en tiempo real entre los diferentes servicios. En esta arquitectura, NATS se utiliza para enviar mensajes y coordinar la ejecución de las funciones entre el servicio faas y el worker. Esto permite que el sistema escale y maneje múltiples invocaciones de funciones de manera eficiente. El servicio NATS escucha en el puerto 4222.

#### **b. Ventajas de la Arquitectura**

* **Escalabilidad**: La arquitectura basada en microservicios permite escalar los componentes de manera independiente. Los servicios pueden ser replicados según la demanda sin afectar a otros servicios del sistema. Esto se logra aprovechando herramientas como NATS y Redis, que manejan la comunicación y el almacenamiento de manera eficiente.
* **Desacoplamiento de Componentes**: Cada servicio está desacoplado, lo que facilita el mantenimiento y la actualización de los diferentes componentes sin afectar el funcionamiento general del sistema. La modularidad también permite que nuevos servicios sean añadidos fácilmente sin una reestructuración significativa.
* **Alta Disponibilidad**: La comunicación a través de NATS y la persistencia de datos en Redis permiten que el sistema sea altamente disponible y tolerante a fallos. Si un servicio falla, los demás pueden continuar funcionando de manera independiente, y los mensajes pueden ser reintentados sin pérdida de datos.
* **Rendimiento**: Redis, como base de datos en memoria, y NATS, como sistema de mensajería de alta velocidad, contribuyen significativamente al alto rendimiento del sistema, permitiendo una rápida respuesta y ejecución de funciones.

La arquitectura del sistema está diseñada para ser flexible, escalable y eficiente, utilizando tecnologías modernas que permiten manejar solicitudes dinámicas y funciones bajo demanda de manera óptima. La separación de responsabilidades entre los diferentes servicios y el uso de herramientas como Redis y NATS asegura que el sistema pueda manejar altos volúmenes de tráfico y ofrecer una experiencia de usuario fluida y rápida.

2.1 Diagramas





2.2. Componentes principales

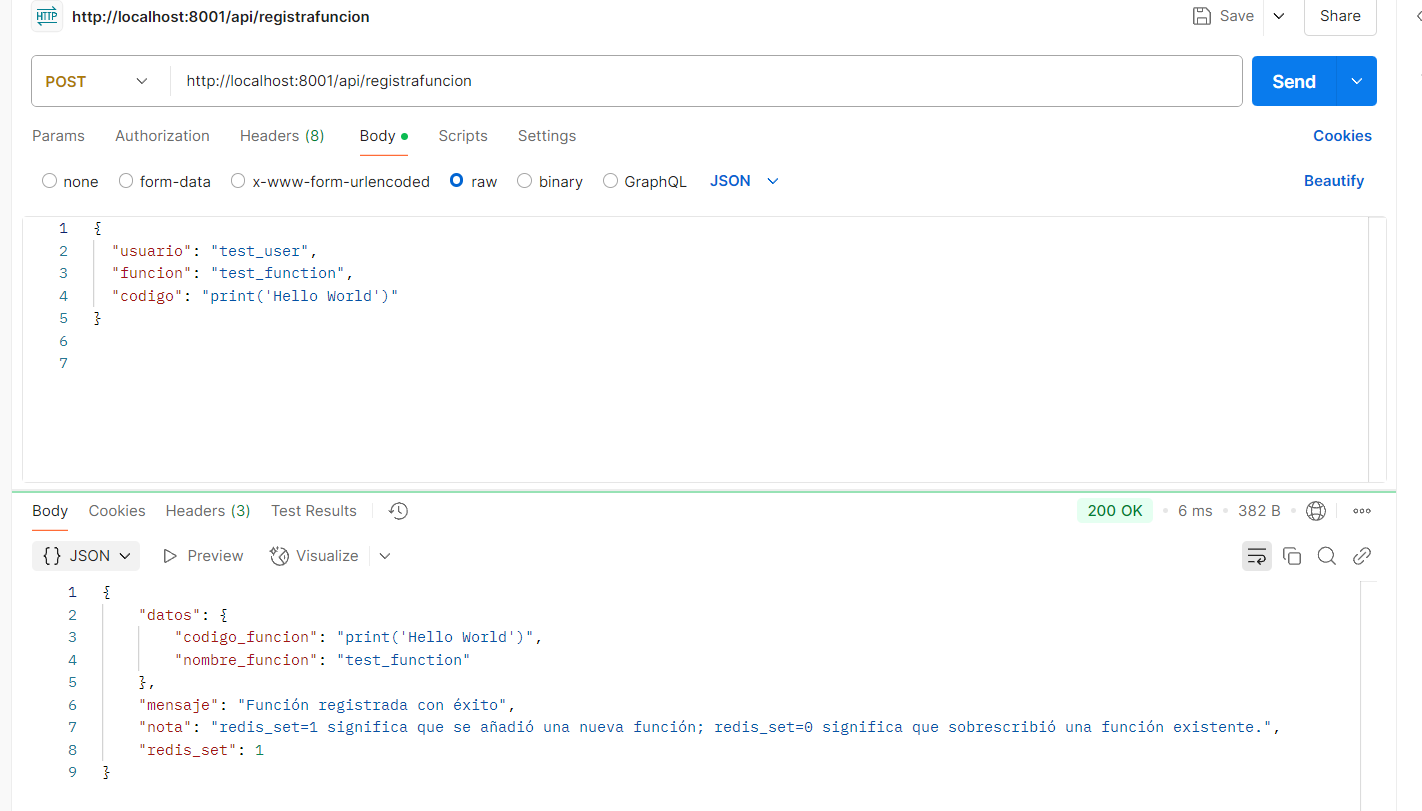
2.2.1. Funciones como servicio (FaaS)

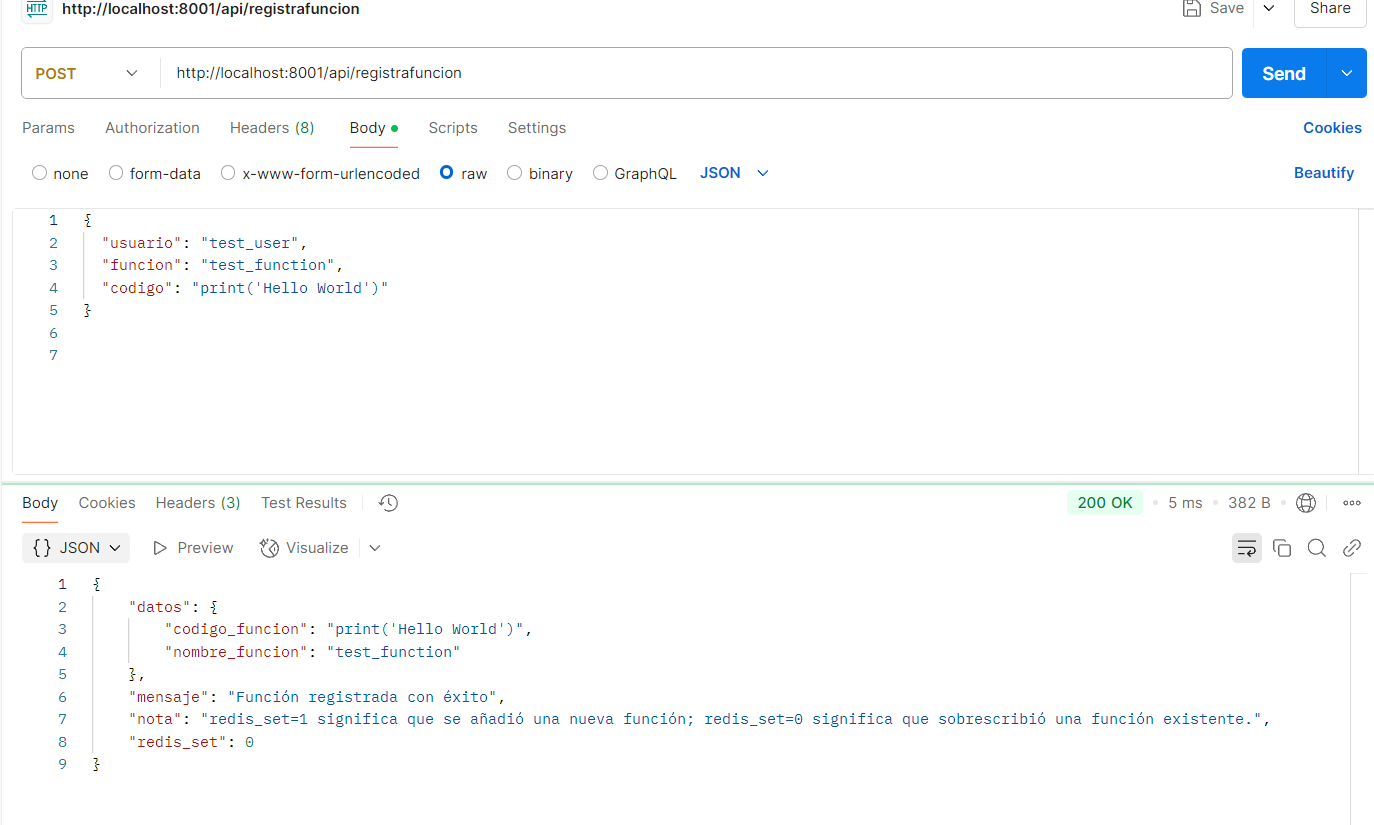
Este modulo gestiona funciones registradas y su ejecución a través de un conjunto de servicios que permiten a los usuarios registrar, desregistrar y ejecutar funciones. Las funciones son almacenadas en una base de datos en memoria, **Redis**, lo que asegura un acceso rápido y eficiente a los códigos de las funciones registradas.

En el sistema, cada función tiene un nombre único y un código asociado. Los usuarios pueden registrar nuevas funciones a través de la API, proporcionándoles un nombre y un bloque de código (en formato de texto). Esta información se almacena en **Redis**, que sirve como almacenamiento en memoria, y permite el acceso rápido a las funciones registradas cuando se necesitan ejecutar.

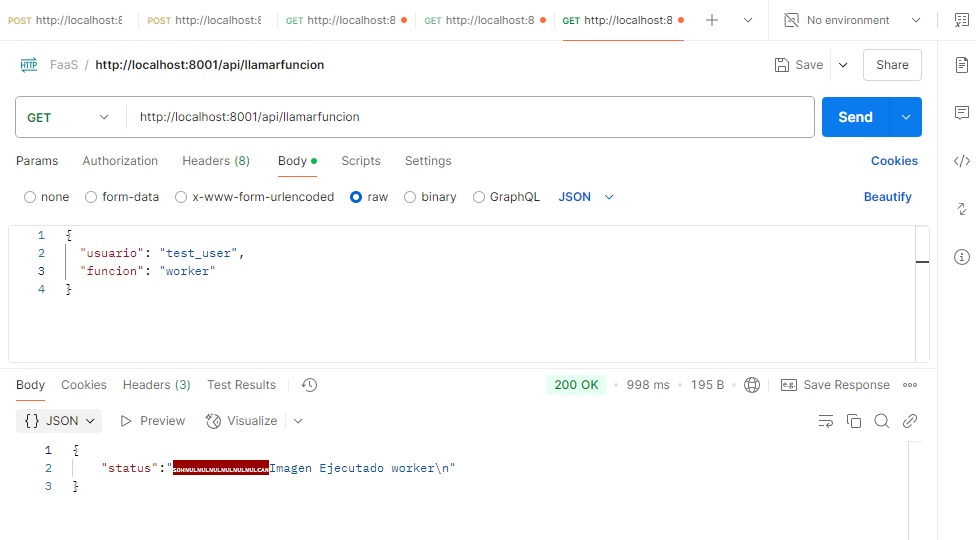
Componentes Clave del modulo FAAS

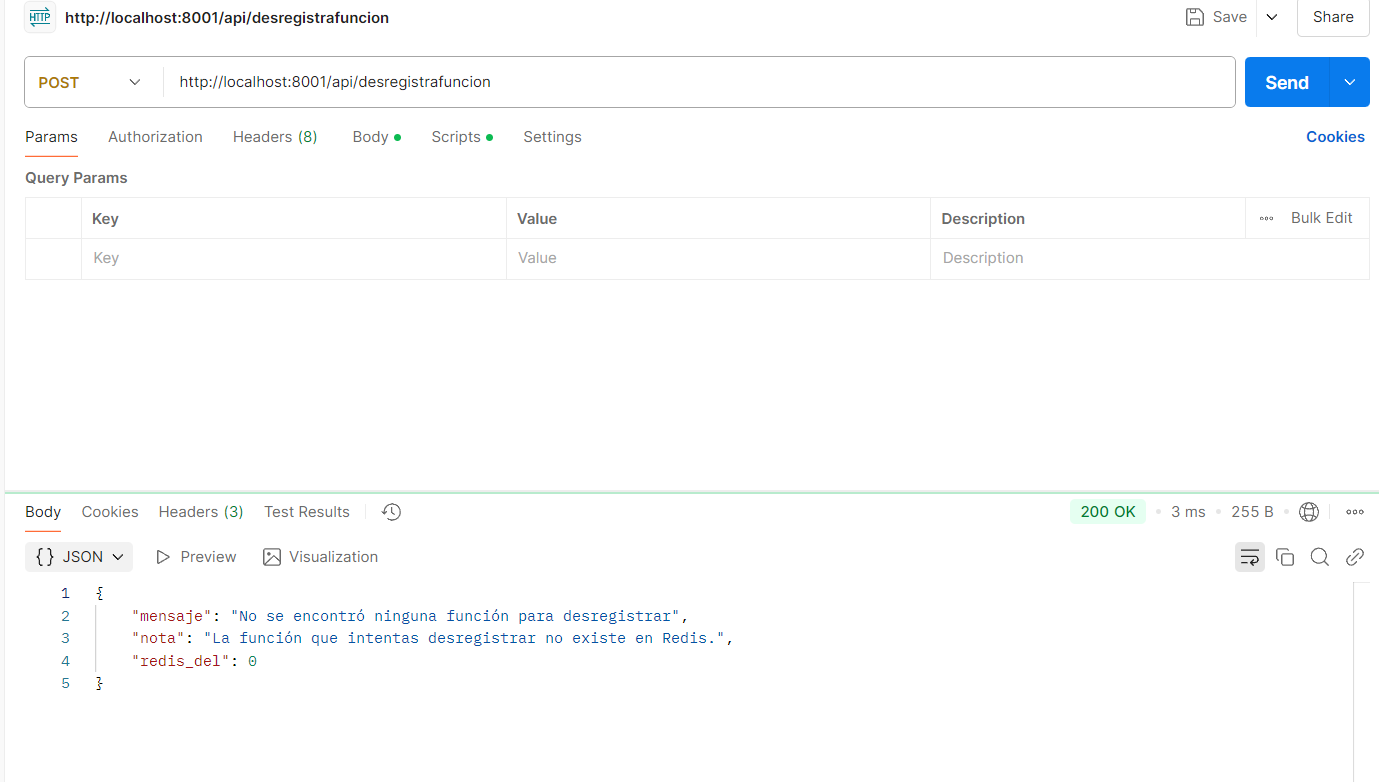
1. **Registrar Funciones**: El servicio permite a los usuarios registrar nuevas funciones o sobrescribir funciones existentes. Cuando un usuario envía una solicitud de registro, se recibe un objeto JSON que contiene el nombre de la función y su código. Este objeto es procesado y almacenado en Redis usando la operación HSet, que guarda el código de la función bajo una clave basada en el nombre de la función. El servidor responde con un mensaje que confirma si la función fue registrada exitosamente.



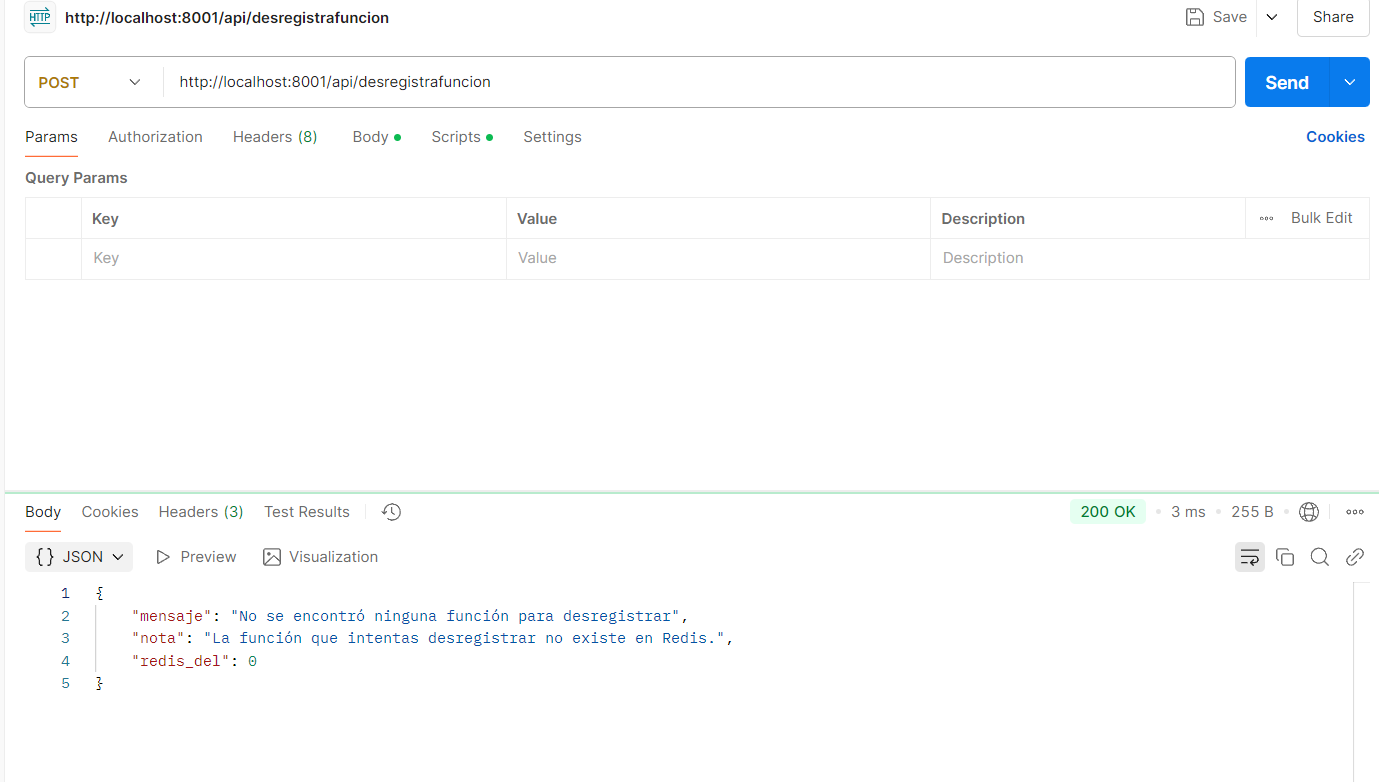


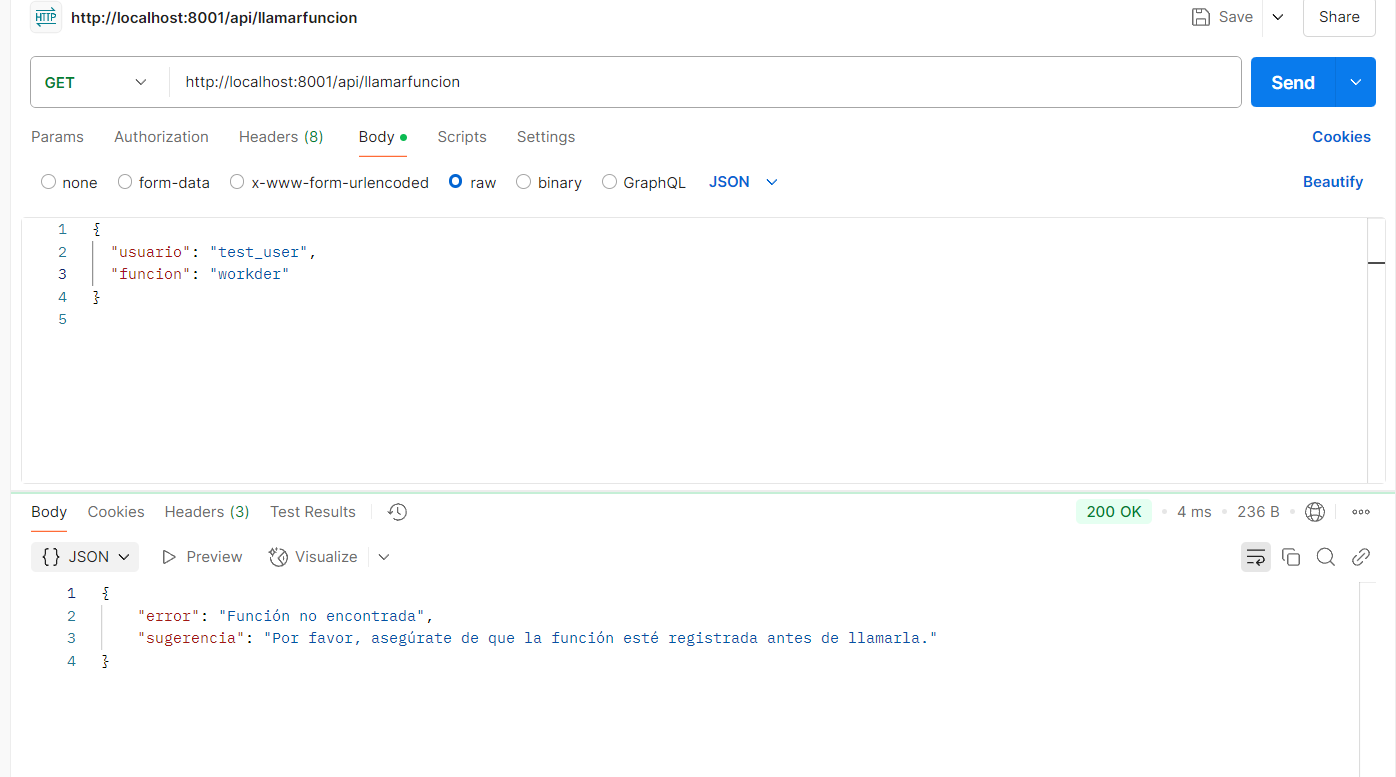
1. **Desregistrar Funciones**: Para eliminar una función previamente registrada, el sistema permite desregistrarla mediante la API. Al recibir una solicitud para desregistrar una función, el sistema verifica su existencia en Redis y, si está registrada, la elimina utilizando el comando HDel. Si no se encuentra la función en Redis, el sistema informa al usuario mediante una respuesta de error.





1. **Ejecutar Funciones**: Una vez que las funciones están registradas, el sistema permite su ejecución. Para invocar una función, el usuario debe enviar una solicitud que incluya el nombre de la función. El sistema consulta Redis para verificar si la función existe. Si la función está registrada, se recupera su código asociado y se devuelve al usuario. En caso contrario, se informa que la función no se encuentra disponible.





El código recuperado es el bloque de instrucciones que el sistema está dispuesto a ejecutar, lo cual en este contexto es un paso esencial para cumplir con el principio de **FaaS**, donde las funciones son ejecutadas bajo demanda.

El uso de **Funciones como Servicio (FaaS)** en este sistema permite una arquitectura flexible, escalable y eficiente. Los usuarios pueden registrar, desregistrar y ejecutar funciones de manera dinámica, lo que facilita la integración de nuevas funcionalidades sin necesidad de reestructurar el sistema. El almacenamiento rápido y eficiente en **Redis**, combinado con la ejecución bajo demanda de funciones, proporciona un rendimiento óptimo y una gestión simplificada de recursos. Además, este enfoque basado en microservicios y funciones desacopladas permite una gran flexibilidad y escalabilidad, siendo ideal para aplicaciones que requieren agilidad y capacidad de adaptación rápida.

2.2.2. Sistema de mensajería (NATS)

En este desarrollo, **NATS** permite la interacción entre diversos componentes del sistema de manera eficiente y desacoplada. A continuación, se describe cómo se integra y funciona NATS en este sistema.

Arquitectura Basada en NATS

El sistema se compone de varios microservicios que interactúan entre sí, y NATS se utiliza como intermediario para enviar y recibir mensajes entre estos servicios. NATS proporciona un canal de comunicación rápido y fiable, asegurando que los mensajes lleguen a su destino sin necesidad de que los servicios dependan directamente unos de otros.

Los microservicios que interactúan con NATS son:

1. **Servicio FaaS (Funciones como Servicio)**: Este servicio se encarga de la gestión de las funciones registradas y su ejecución bajo demanda. Cuando un cliente solicita la ejecución de una función registrada, el servicio de **FaaS** necesita coordinarse con otros servicios, como el **Orquestador** o el **Worker**, para llevar a cabo el trabajo. Para ello, utiliza **NATS** para enviar mensajes entre los servicios que componen el flujo de ejecución.
2. **Orquestador**: El servicio **Orquestador** es responsable de coordinar el flujo general del sistema, como la invocación de funciones o la gestión de procesos que necesitan ser ejecutados en paralelo o de manera secuencial. Utiliza NATS para recibir solicitudes del servicio **FaaS** y enviar las órdenes de ejecución a los servicios correspondientes, como el **Worker**.
3. **Worker**: El **Worker** es el servicio que ejecuta las funciones registradas. Este servicio está suscrito a un canal de mensajes de **NATS**, esperando recibir mensajes que le indiquen qué función debe ejecutar. Al recibir un mensaje del **Orquestador** a través de **NATS**, el Worker procesa la solicitud y ejecuta la función correspondiente.

Comunicación Síncrona

Una de las principales características de **NATS** es su modelo de comunicación **pub/sub** (publicador/suscriptor). Esto significa que los servicios pueden publicar mensajes en un canal específico, y los servicios interesados pueden suscribirse a esos canales para recibir los mensajes. En este caso:

* **Publicación**: El servicio **FaaS** o **Orquestador** publica mensajes sobre qué funciones ejecutar o coordinar, informando a otros servicios.
* **Suscripción**: El servicio **Worker** se suscribe a los canales de **NATS** y espera recibir mensajes con detalles sobre las funciones que debe ejecutar.

Este enfoque **síncrono** permite que los servicios dependan directamente de otros para realizar sus tareas. Los servicios pueden operar de manera independiente, mejorando la escalabilidad y robustez del sistema. Si un servicio está inactivo o no disponible temporalmente, los mensajes pueden almacenarse en el sistema de NATS y ser procesados cuando el servicio vuelva a estar disponible.

2.2.3. Almacenamiento en memoria (Redis)

En este sistema, Redis se utiliza para almacenar las funciones que los usuarios pueden registrar y ejecutar a través del servicio **FaaS** (Funciones como Servicio). Cada vez que un usuario registra una nueva función mediante la API /api/registrafuncion, el sistema guarda el código de la función en Redis bajo un conjunto de claves específicas.

* **Clave**: El nombre de la función (fn.Funcion).
* **Valor**: El código de la función (fn.Codigo).

Este almacenamiento en Redis permite verificar rápidamente las credenciales del usuario cuando este intenta autenticarse, sin necesidad de acceder a bases de datos más lentas o externas, lo que mejora la eficiencia en el proceso de inicio de sesión.

go

Copy

client.Set(ctx, user.Correo, hashedPassword, 0)

Ventajas del Uso de Redis en este Desarrollo

1. **Alto Rendimiento**: Al ser una base de datos en memoria, Redis ofrece tiempos de acceso extremadamente rápidos, lo que es esencial en un sistema que necesita manejar grandes volúmenes de solicitudes en tiempo real, como el sistema de **FaaS**.
2. **Escalabilidad**: Redis es altamente escalable, lo que permite que el sistema crezca sin afectar el rendimiento. A medida que el número de funciones y usuarios aumenta, Redis puede manejar eficazmente el almacenamiento en memoria de datos sin comprometer la velocidad de acceso.
3. **Persistencia Opcional**: Aunque Redis es principalmente una base de datos en memoria, también ofrece opciones para persistir datos a disco. En este caso, se podría optar por configurar Redis para guardar los datos de funciones o contraseñas en caso de reinicios del sistema, aunque en este desarrollo se prioriza la rapidez y eficiencia.
4. **Simplicidad y Eficiencia en el Modelado de Datos**: Redis permite modelar de forma eficiente las relaciones clave-valor y estructuras como hashes, listas o conjuntos, que se adaptan bien a las necesidades del sistema. Al utilizar hashes para almacenar funciones y contraseñas, Redis simplifica las operaciones de lectura y escritura sin necesidad de realizar consultas complejas.
5. **Desempeño en Alta Concurrencia**: El uso de Redis facilita la gestión de múltiples solicitudes concurrentes, ya que es capaz de manejar miles de operaciones por segundo con una latencia muy baja. Esto es esencial en un sistema que debe interactuar con varios microservicios, como en el caso de la comunicación entre el servicio **FaaS**, **Orquestador**, y **Worker**.

2.2.4. Autenticación y autorización (JWT)

El servicio de **autenticación** implementado está diseñado para manejar el registro y la validación de usuarios utilizando **JWT (JSON Web Tokens)** y un mecanismo de cifrado de contraseñas mediante **bcrypt**. Este enfoque asegura que solo los usuarios autenticados puedan acceder a los recursos del sistema, proporcionando un acceso seguro y controlado.

El servicio de autenticación está dividido en dos funcionalidades principales: **registro de usuario** y **validación de usuario**.

**Registro de Usuario (API: /api/registro)**

El proceso de registro de usuario consiste en crear una nueva cuenta para el usuario, almacenando su información de forma segura y generando un token JWT que el usuario podrá usar para autenticar futuras solicitudes.

* **Recibo de datos**: Cuando un usuario envía una solicitud para registrarse, el servicio recibe un objeto JSON que contiene el correo electrónico y la contraseña del usuario. El servicio deserializa estos datos en una estructura User que contiene los campos Correo y Pswd.

go

Copy

var user User

json.NewDecoder(r.Body).Decode(&user)

* **Cifrado de la Contraseña**: La contraseña proporcionada por el usuario se cifra utilizando el algoritmo **bcrypt** para asegurar que no se almacene en texto plano en la base de datos. Esto mejora la seguridad al proteger la información sensible del usuario.

go

Copy

hashedPassword, \_ := bcrypt.GenerateFromPassword([]byte(user.Pswd), bcrypt.DefaultCost)

* **Almacenamiento de la Contraseña Cifrada**: La contraseña cifrada se guarda en **Redis**, utilizando el correo del usuario como clave. Redis es una base de datos en memoria que ofrece acceso rápido a los datos.

go

Copy

client.Set(ctx, user.Correo, hashedPassword, 0)

* **Generación de JWT**: Una vez que el usuario se ha registrado exitosamente, el sistema genera un **JSON Web Token (JWT)**. Este token es un identificador único que se utiliza para autenticar las solicitudes del usuario en el futuro. El token contiene el correo del usuario como una reclamación (claim) y es firmado usando una clave secreta (jwtKey).

go

Copy

token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{"correo": user.Correo})

tokenString, \_ := token.SignedString(jwtKey)

* **Respuesta con el Token**: Finalmente, el servicio responde al usuario enviando el token JWT generado. Este token puede ser almacenado por el cliente y utilizado en solicitudes subsecuentes para verificar su identidad.

go

Copy

json.NewEncoder(w).Encode(map[string]string{"token": tokenString})

**Validación de Usuario (API: /api/validarusuario)**

La segunda funcionalidad del servicio de autenticación es la **validación de usuarios**. Esta operación se utiliza para verificar si un usuario está autenticado y autorizado para acceder a ciertos recursos del sistema.

* **Extracción del Token**: El usuario debe enviar el **JWT** generado anteriormente en el encabezado de autorización de sus solicitudes. El servicio extrae este token de los encabezados HTTP.

go

Copy

tokenString := r.Header.Get("Authorization")

* **Validación del Token**: El token recibido es analizado y verificado usando la clave secreta (jwtKey). Si el token es válido y contiene las reclamaciones necesarias (en este caso, el correo del usuario), el servicio permite el acceso. Si el token es inválido o no está presente, el servicio responde con un código de estado HTTP 401 Unauthorized.

go

Copy

token, \_ := jwt.Parse(tokenString, func(token \*jwt.Token) (interface{}, error) {

return jwtKey, nil

})

* **Respuesta de Validación**: Si el token es válido, el servicio responde con un código HTTP 200 OK, lo que indica que el usuario está autorizado. Si el token es inválido, se responde con un código HTTP 401 Unauthorized.

go

Copy

if \_, ok := token.Claims.(jwt.MapClaims); !ok || !token.Valid {

w.WriteHeader(http.StatusUnauthorized)

return

}

w.WriteHeader(http.StatusOK)

**Ventajas del Sistema de Autenticación Implementado**

* **Seguridad Mejorada**: El uso de **bcrypt** para el cifrado de contraseñas asegura que las contraseñas no se almacenen en texto claro en la base de datos. Además, **JWT** garantiza que las solicitudes sean autenticadas de forma segura sin la necesidad de mantener sesiones de usuario en el servidor.
* **Escalabilidad**: El sistema basado en **JWT** es altamente escalable, ya que el token puede ser verificado en cualquier servicio sin necesidad de almacenar el estado de la sesión en el servidor. Esto es especialmente útil en sistemas distribuidos y microservicios.
* **Simplicidad y Flexibilidad**: La autenticación basada en JWT permite que los usuarios inicien sesión una vez y utilicen el token en todas sus solicitudes posteriores, sin la necesidad de volver a ingresar sus credenciales cada vez. Además, el sistema es fácil de integrar con otros servicios que necesiten autenticación.
* **Manejo Eficiente de Usuarios**: Al usar **Redis** para almacenar la contraseña cifrada, el sistema puede acceder rápidamente a los datos de los usuarios, lo que mejora la eficiencia y permite un acceso rápido a la información sin comprometer la seguridad.

2.3. Flujo de datos entre componentes

1. **Registro de Usuario**: El usuario se registra a través del servicio auth, que cifra la contraseña utilizando bcrypt y la almacena en Redis. El servicio auth genera un JWT que se devuelve al usuario y que puede ser utilizado en solicitudes posteriores para acceder a otros servicios.
2. **Registro de Función**: El usuario, una vez autenticado, interactúa con el servicio faas para registrar funciones. Estas funciones se almacenan en Redis bajo un identificador único. El servicio faas también puede desregistrar o invocar funciones en función de las solicitudes que recibe.
3. **Orquestación de Funciones**: El servicio orquestador gestiona las solicitudes y coordina las invocaciones de funciones entre los diferentes servicios. Al recibir una solicitud para ejecutar una función, el orquestador se comunica con Redis para verificar la existencia de la función y con NATS para enviar la tarea al servicio worker.
4. **Ejecución de Funciones**: El servicio worker escucha los mensajes de NATS enviados por el servicio faas o el orquestador. Una vez que recibe una solicitud, ejecuta la función correspondiente y devuelve el resultado.
5. **Mensajería en Tiempo Real**: NATS actúa como el sistema de mensajería central entre los servicios. Permite que las funciones sean invocadas de manera asincrónica, enviando mensajes a los workers para la ejecución de tareas. Este enfoque reduce la latencia y mejora la eficiencia al manejar múltiples solicitudes simultáneamente.

3. Instalación y Configuración

3.1. Prerrequisitos

Para desarrollar un proyecto basado en la arquitectura **FaaS** (Función como Servicio), se deben cumplir ciertos requisitos clave que aseguren la correcta implementación y operación del sistema:

1. **Plataforma de FaaS**: Se necesita una plataforma FaaS que gestione la ejecución de las funciones. Esto puede ser un servicio en la nube como AWS Lambda, Google Cloud Functions, o una solución autohospedada como OpenFaaS. La plataforma debe permitir la carga, despliegue y escalado automático de las funciones.
2. **Lenguaje de Programación**: FaaS es compatible con varios lenguajes, Se seleccionó Go, debido a su eficiencia y facilidad de integración.
3. **Contenedores (Docker)**: Para aplicaciones FaaS autohospedadas o cuando se desea mayor control sobre el entorno de ejecución, es útil usar contenedores (como Docker). Esto permite empaquetar las funciones con sus dependencias, facilitando su despliegue en diferentes entornos.
4. **Gestión de Estado**: En FaaS, las funciones son sin estado por diseño, por lo que se necesita una solución externa para manejar el estado persistente, como bases de datos (ej. Redis, MongoDB) o sistemas de almacenamiento en la nube.
5. **Autenticación y Seguridad**: Implementar medidas de seguridad, como el uso de JWT (JSON Web Tokens) para autenticar y autorizar las solicitudes a las funciones, es fundamental. Además, se deben gestionar adecuadamente las credenciales y las políticas de acceso.
6. **API Gateway**: Un API Gateway es necesario para gestionar las solicitudes de entrada hacia las funciones, controlar el tráfico, manejar las autenticaciones y aplicar políticas de enrutamiento y balanceo de carga.
7. **Monitorización y Logging**: Es fundamental contar con herramientas para la monitorización y el registro de eventos (logs) en tiempo real para poder identificar problemas, hacer un seguimiento de las ejecuciones y mejorar la eficiencia del sistema.

Estos requisitos básicos permiten que un desarrollo FaaS funcione de manera eficiente, escalable y segura, cumpliendo con las necesidades del negocio.

3.1.1. Entorno de desarrollo (Go, Docker, etc.)

El entorno de desarrollo para este proyecto implica varias configuraciones y servicios interdependientes que permiten crear una infraestructura de microservicios escalable y distribuida. A continuación, se detallan los componentes clave del entorno de desarrollo y su interacción:

Docker Compose:

La estructura proporcionada utiliza **Docker Compose**, que define y configura varios servicios en contenedores Docker. Docker Compose es una herramienta que facilita la orquestación de contenedores, permitiendo que múltiples servicios (contenedores) se levanten de forma conjunta y se configuren de manera simple mediante un archivo YAML.

El entorno se compone de los siguientes servicios:

* **auth**: Este servicio gestiona la autenticación y probablemente la generación y validación de tokens JWT. Está construido desde el directorio ./services/auth y expone el puerto 8000. Depende de Redis, lo que sugiere que podría utilizar Redis para gestionar sesiones o almacenar tokens.
* **faas**: Este servicio es el corazón del proyecto FaaS y es responsable de ejecutar las funciones. También está construido desde el directorio ./services/faas y expone el puerto 8001. Depende de Redis y NATS, lo que indica que Redis puede ser usado para almacenar datos temporales, mientras que NATS se usa para la mensajería entre servicios (probablemente para manejar las invocaciones de funciones).
* **orquestador**: Este servicio parece ser el encargado de orquestar el flujo entre las funciones, gestionando las invocaciones y el enrutamiento. El servicio se construye desde ./services/orquestador y expone el puerto 8002. Además, tiene acceso al socket de Docker (mediante el volumen /var/run/docker.sock:/var/run/docker.sock), lo que sugiere que podría gestionar contenedores o interactuar con otros servicios de Docker de manera dinámica.
* **worker**: El servicio de trabajo está diseñado para ejecutar tareas específicas o procesar funciones. Se construye desde ./services/worker y depende de NATS, lo que indica que los trabajadores reciben tareas a través de NATS.
* **redis**: Este servicio proporciona la base de datos en memoria clave-valor Redis, que es fundamental para la persistencia de datos temporales, como tokens o estados. Utiliza la imagen redis:latest y mapea el puerto 6379 para que sea accesible desde otros contenedores. Además, se monta un archivo de configuración personalizado redis.conf.
* **nats**: El servicio NATS proporciona un sistema de mensajería ligero y rápido para la comunicación entre los diferentes servicios. Utiliza la imagen nats:latest y expone el puerto 4222, lo que permite la interacción de los otros servicios a través de este sistema de mensajería.

3.1.2. Instalación de NATS y Redis

La instalación de **NATS** y **Redis** en un proyecto basado en **Docker Compose** como en este caso, se gestiona principalmente a través de la configuración de servicios en el archivo docker-compose.yml. Ambos servicios son componentes clave para el manejo de la mensajería y el almacenamiento en memoria del proyecto FaaS. A continuación, se describe el proceso de instalación y configuración de cada uno dentro de este entorno específico.

### Redis:

En este proyecto, Redis se instala utilizando una imagen oficial de Docker (redis:latest), lo que simplifica considerablemente su despliegue y configuración. Al incluir el servicio redis en el archivo docker-compose.yml, el contenedor se inicia automáticamente cuando se levanta el entorno de desarrollo. El servicio Redis se configura para exponer el puerto 6379, lo que permite que otros servicios del contenedor (como faas, auth, y orquestador) se conecten y utilicen Redis para almacenar o recuperar datos.

Una característica adicional en este entorno es la inclusión de un archivo de configuración personalizado (redis.conf). Este archivo se monta dentro del contenedor para permitir configuraciones específicas de Redis, como la gestión de la persistencia de datos, la configuración de la memoria y otras optimizaciones de rendimiento. La flexibilidad de Docker permite modificar este archivo fácilmente para ajustar la configuración sin necesidad de acceder directamente al contenedor.

### Nats:

En este caso, NATS se configura también como un servicio dentro del archivo docker-compose.yml, utilizando la imagen oficial de Docker (nats:latest). Al igual que Redis, NATS se expone a través de un puerto, en este caso el 4222, lo que permite que los otros servicios en el proyecto puedan conectarse y transmitir mensajes a través de este bus de mensajes.

La principal función de NATS en este proyecto es gestionar la comunicación asincrónica entre los servicios, como el envío de tareas a los trabajadores (worker) o la orquestación de la ejecución de funciones en el servicio faas. NATS permite un alto rendimiento en la transmisión de mensajes y es adecuado para sistemas distribuidos como este, donde la escalabilidad y la rapidez de la comunicación son esenciales.

La integración de **Redis** y **NATS** en este proyecto no solo facilita la persistencia de datos y la mensajería, sino que también establece un entorno de servicios distribuidos y escalables. La clave para que ambos servicios funcionen correctamente radica en la forma en que están definidos en el archivo docker-compose.yml y cómo los otros servicios del proyecto (como faas, auth, worker, y orquestador) los consumen. Docker Compose gestiona automáticamente las dependencias entre los servicios a través de la directiva depends\_on, asegurando que Redis y NATS estén disponibles antes de iniciar los servicios que dependen de ellos.

El servicio **faas**, por ejemplo, requiere de Redis para almacenar datos temporales y de NATS para la mensajería y la invocación de funciones. Del mismo modo, el servicio **worker** depende de NATS para recibir mensajes y procesar las tareas de manera eficiente. La estructura modular de Docker y la forma en que se gestionan estos contenedores facilita la instalación y el escalado de estos servicios sin la necesidad de intervención manual, ya que Docker se encarga de todo el proceso de despliegue.

3.2. Configuración del proyecto

3.2.1. Archivo de configuración (docker-compose.yml)

El archivo docker-compose.yml describe la configuración de varios servicios que componen un entorno de desarrollo basado en contenedores Docker. En este archivo, se definen los siguientes servicios:

1. **auth**: Este servicio maneja la autenticación, probablemente generando y validando tokens JWT. Está construido desde el directorio ./services/auth, expone el puerto 8000, y depende del servicio redis, lo que sugiere que utiliza Redis para almacenar información relacionada con las sesiones o tokens.
2. **faas**: Este servicio es el encargado de ejecutar las funciones dentro de la arquitectura FaaS. Se construye desde ./services/faas, expone el puerto 8001 y depende de redis y nats. Redis probablemente se utiliza para almacenar datos temporales, mientras que NATS facilita la mensajería entre servicios.
3. **orquestador**: Este servicio coordina y gestiona el flujo de trabajo entre las funciones. Se construye desde ./services/orquestador, expone el puerto 8002, y también depende de redis y nats. Además, se monta un volumen que permite que el contenedor interactúe con el socket de Docker (/var/run/docker.sock), lo que podría implicar que el orquestador gestiona otros contenedores o recursos del sistema.
4. **worker**: El servicio worker es responsable de procesar las tareas que se le asignan, probablemente de acuerdo con las invocaciones de funciones o el enrutamiento gestionado por el orquestador. Este servicio también depende de nats, que se usa para la mensajería entre servicios.
5. **redis**: Este servicio utiliza la imagen oficial de Redis (redis:latest) y expone el puerto 6379. Además, se monta un archivo de configuración (redis.conf), que permite personalizar la configuración de Redis según las necesidades del proyecto.
6. **nats**: Este servicio usa la imagen oficial de NATS (nats:latest) y expone el puerto 4222, lo que habilita la comunicación entre los diferentes servicios del proyecto a través de un sistema de mensajería eficiente y ligero.

El archivo docker-compose.yml define una infraestructura distribuida y escalable, donde varios servicios interactúan entre sí, con Redis para almacenamiento y NATS para mensajería, permitiendo un entorno de desarrollo para funciones en la arquitectura FaaS.

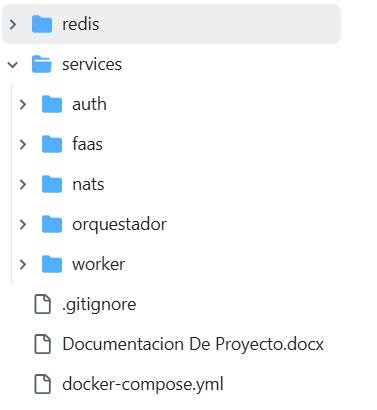
4. Estructura del Proyecto

4.1. Descripción de carpetas y archivos

El proyecto se encuentra almacenado y centralizado en Github de la siguiente manera:

Carpeta Raiz Redis

Carpeta raiz Services con las subcarpetas Auth, Faas, Nats, Orquestador, Worker.



4.1.1. internal/auth: Gestión de JWT

Este código implementa un sistema de autenticación utilizando JSON Web Tokens (JWT) en Go, combinando el uso de Redis para almacenar contraseñas y el paquete jwt para gestionar tokens. A continuación, se explica la gestión del proceso de autenticación paso a paso:

a. Estructura del Usuario (User)

El tipo User contiene dos campos:

* Correo: Representa el correo electrónico del usuario, que es utilizado para identificar al usuario.
* Pswd: Almacena la contraseña del usuario.

Estos datos se reciben en formato JSON en las solicitudes HTTP para registrar o validar usuarios.

b.Generación de Token en el Registro

La función registerUser es responsable de registrar un nuevo usuario. Este proceso incluye:

* **Recepción de Datos**: Se decodifica el JSON de la solicitud HTTP para obtener el correo y la contraseña del usuario.
* **Cifrado de Contraseña**: Utiliza la biblioteca bcrypt para generar un hash de la contraseña del usuario. Este hash se guarda en una base de datos Redis, donde la clave es el correo del usuario.
* **Generación de Token**: Usando la librería jwt, se crea un JWT con el correo del usuario como una "claim" (reclamación). El token se firma utilizando una clave secreta (jwtKey) con el algoritmo HS256.
* **Envío del Token**: Se devuelve al usuario el JWT generado en la respuesta HTTP, bajo el encabezado "Authorization", para que lo utilice en futuras solicitudes.

c. Validación del Token

La función validateUser se encarga de verificar la validez de un token JWT recibido en la cabecera de la solicitud HTTP (Authorization):

* **Extracción del Token**: Se extrae el token de la cabecera Authorization.
* **Verificación del Token**: Se valida el token utilizando la misma clave secreta (jwtKey) y el algoritmo de firma HS256. Si el token es válido, se confirma que la reclamación (claim) contenida en el token es correcta y que el token no ha expirado.
* **Respuesta**: Si el token es válido, se responde con un estado HTTP 200 (OK). Si no es válido, se responde con un estado HTTP 401 (Unauthorized).

d. Flujo General

El flujo de interacción es el siguiente:

1. El usuario envía su correo y contraseña a través de una solicitud HTTP POST al endpoint /api/registro.
2. El servidor genera un token JWT y lo devuelve al usuario en la respuesta.
3. Para acceder a rutas protegidas o validar su sesión, el usuario debe enviar el token en la cabecera de sus solicitudes HTTP.
4. El servidor verifica el token en cada solicitud que requiera autenticación, permitiendo el acceso si el token es válido.

e. Servidor HTTP

El servidor escucha en el puerto 8000 y tiene dos rutas principales:

* /api/registro: Registra a un usuario nuevo y le proporciona un JWT.
* /api/validarusuario: Valida si el JWT enviado en la cabecera de la solicitud es válido.

4.1.2. internal/messaging: Conexión y manejo de NATS

4.1.3. internal/storage: Integración con Redis

5. Implementación de Funciones

5.1. Cómo manejar eventos de NATS

1. **Conexión Exitosa**: El servidor se conecta a NATS. Manejo implícito mediante la ausencia de errores en la conexión.
2. **Publicación de Mensaje**: Se publica un mensaje a NATS, pero no se define explícitamente en el código actual.
3. **Solicitud a NATS**: El código envía una solicitud a NATS mediante nc.Request() y espera una respuesta.
4. **Respuesta de NATS**: El código maneja la respuesta de NATS y la devuelve al cliente.
5. **Desconexión de NATS**: Se maneja de forma implícita con defer nc.Close().
6. **Error de Conexión**: Se maneja al verificar si nats.Connect() devuelve un error.
7. **Error en la Solicitud**: Se maneja al verificar si nc.Request() devuelve un error.

5.2. Uso de Redis para almacenamiento temporal

6. Despliegue

6.1. Entorno de producción

Diagrama insertar

6.2. Uso de Docker

6.3. Estrategias de despliegue continuo (CI/CD)

El uso de **Docker Compose** junto con **GitHub** permite automatizar el proceso de despliegue de aplicaciones en contenedores de manera eficiente, facilitando la configuración y orquestación de múltiples contenedores. A continuación, se describe cómo se puede desplegar una aplicación utilizando Docker Compose a través de un repositorio en GitHub.

**Preparación del Repositorio en GitHub**

Primero, debes tener un repositorio en GitHub que contenga los archivos necesarios para construir y ejecutar tu aplicación. Los pasos iniciales incluyen:

* **Crear un Repositorio en GitHub**: Si aún no tienes un repositorio, crea uno en GitHub. Puedes hacer esto desde el panel principal de GitHub, haciendo clic en "New Repository".
* **Subir los Archivos de la Aplicación**: Los archivos que subas deben incluir el código de tu aplicación y un archivo docker-compose.yml que define cómo deben ser desplegados los contenedores.

**Configuración de un Archivo de Acción de GitHub (GitHub Actions)**

GitHub Actions te permite automatizar flujos de trabajo (workflows) directamente desde tu repositorio. Para desplegar tu aplicación usando Docker Compose, debes configurar un archivo de acción que defina los pasos a seguir.

* **Crear un Directorio de GitHub Actions**: En tu repositorio de GitHub, crea un directorio .github/workflows/ donde se almacenarán los archivos de configuración de las acciones.
* **Definir un Workflow para el Despliegue**: Dentro del directorio .github/workflows/, crea un archivo YAML, por ejemplo docker-deploy.yml, que contendrá la definición del flujo de trabajo.

**Acción del Workflow**

Cuando se ejecuta un push a la rama main del repositorio, GitHub ejecutará automáticamente el flujo de trabajo definido. Los pasos son los siguientes:

1. **GitHub Clona el Repositorio**: El flujo de trabajo comienza con la acción actions/checkout, que descarga el código del repositorio.
2. **Configura Docker**: La acción docker/setup-buildx-action asegura que Docker esté configurado adecuadamente en el entorno de GitHub Actions.
3. **Construcción y Despliegue de Contenedores**: Utilizando el archivo docker-compose.yml, el comando docker-compose up -d construye las imágenes necesarias y arranca los contenedores en segundo plano.

**Despliegue en un Servidor**

Aunque en el ejemplo anterior se utiliza GitHub Actions para construir y ejecutar contenedores en la misma plataforma de GitHub, es común que los despliegues se realicen en un servidor remoto (por ejemplo, un servidor en la nube como AWS, DigitalOcean, o un servidor local).

**Monitoreo y Mantenimiento**

Después de que se haya desplegado la aplicación, es importante monitorear los contenedores. Puedes agregar pasos adicionales en el archivo de GitHub Actions para verificar el estado de los contenedores o para hacer un docker-compose down cuando sea necesario. También es recomendable configurar alertas para recibir notificaciones si algún contenedor falla.

6.4. Escalabilidad de las funciones

La **escalabilidad** es clave para garantizar que el sistema pueda manejar un aumento en la carga de trabajo sin comprometer su rendimiento. La arquitectura está diseñada de tal manera que cada componente pueda escalar de manera independiente según las necesidades, utilizando contenedores Docker y herramientas de mensajería como NATS para facilitar la interacción entre los servicios. Los diferentes components interactuan para gestionarla.

La escalabilidad horizontal de microservicios en el sistema está contenido dentro de un contenedor Docker, lo que permite que los servicios escalen de manera independiente. Los microservicios en este proyecto incluyen:

* **Servicio de Autenticación (auth)**: Encargado del registro y validación de usuarios mediante JWT. Si el tráfico de autenticación aumenta, se pueden desplegar más instancias de este servicio.
* **Servicio FaaS (faas)**: Gestiona las funciones registradas. Este servicio puede escalarse horizontalmente para manejar un mayor número de solicitudes para registrar, ejecutar y desregistrar funciones.
* **Orquestador (orquestador)**: Coordina la comunicación entre los otros servicios. La escalabilidad de este componente se maneja iniciando más instancias si es necesario para gestionar mayores cargas de trabajo.
* **Worker (worker)**: Ejecuta las funciones registradas en un contenedor Docker aislado. Este servicio es crucial para la escalabilidad del sistema, ya que cada vez que una función es invocada, un nuevo contenedor puede ser creado dinámicamente para ejecutarla.

Adicionalmente Docker proporciona una base sólida para la escalabilidad del sistema. Los contenedores pueden ser replicados y distribuidos sin necesidad de modificar el código, lo que facilita la respuesta ante picos de tráfico. El uso de Docker permite gestionar y escalar múltiples instancias de un servicio automáticamente.

* **Escalabilidad de Workers**: El componente **worker** es el núcleo de la ejecución de funciones. Cada vez que un usuario invoca una función, el orquestador puede disparar la creación de un nuevo contenedor para ejecutar esa función en particular. Si la demanda de funciones aumenta, el número de instancias de workers puede crecer de manera dinámica, asegurando que las funciones sean procesadas sin retrasos significativos.

Los workers interactúan con el sistema de mensajería NATS para recibir solicitudes de ejecución de funciones, lo que permite distribuir las tareas de manera eficiente.

La interacción entre los componentes es esencial para lograr una escalabilidad efectiva. Los microservicios se comunican entre sí utilizando una arquitectura distribuida basada en mensajes. A continuación, se detalla cómo interactúan los componentes para escalar de manera eficiente:

* **Orquestador y Workers**: El **orquestador** es responsable de gestionar la carga de trabajo y distribuir las solicitudes de ejecución de funciones a los **workers**. El orquestador se comunica con los workers a través de **NATS**, un sistema de mensajería ligera que permite una comunicación eficiente en tiempo real. Cuando un worker recibe una solicitud, crea un nuevo contenedor Docker para ejecutar la función solicitada. Si la carga aumenta, el orquestador puede enviar más solicitudes para iniciar nuevos workers, garantizando que se puedan ejecutar más funciones de manera simultánea.
* **Redis**: Actúa como una base de datos clave-valor que almacena las funciones registradas y sus respectivos códigos. El sistema de workers consulta Redis para verificar si una función está registrada y obtener su código para ejecutarlo. Redis también ayuda a mantener el estado entre diferentes instancias del sistema, lo que garantiza la coherencia en la ejecución de funciones.
* **NATS para Comunicación en Tiempo Real**: La escalabilidad también depende de una mensajería eficiente. **NATS** facilita la comunicación entre los servicios, permitiendo que el orquestador, los workers y otros componentes se comuniquen en tiempo real. La interacción basada en mensajes permite que los workers puedan ser iniciados dinámicamente y que el sistema maneje un mayor volumen de tareas sin cuello de botella en la comunicación.

La capacidad de escalar los servicios horizontalmente se ajusta automáticamente en función de la demanda o carga de trabajo. A medida que los usuarios invocan más funciones, el orquestador puede activar más instancias de workers. Al estar orquestados en contenedores Docker, los servicios pueden ser replicados sin grandes esfuerzos, permitiendo que el sistema pueda responder ante un incremento en la cantidad de funciones que se deben ejecutar sin que se afecte el rendimiento global del sistema.

* **Redis**: El uso de Redis permite almacenar y acceder rápidamente a la información de las funciones, lo que es esencial para la escalabilidad. En una implementación de producción, Redis puede ser replicado y distribuido para manejar mayores volúmenes de datos y solicitudes concurrentes sin degradar el rendimiento.
* **NATS**: En un entorno con alta demanda, **NATS** puede ser escalado para soportar una mayor cantidad de mensajes entre los microservicios. Esto asegura que la mensajería sea eficiente y pueda manejar un alto volumen de solicitudes de ejecución de funciones sin latencia excesiva.

7. Documentación oficial de herramientas utilizadas

Docker – <https://www.docker.com>

Go – <https://go.dev>

Github – <https://github.com>

VS2022 - <https://visualstudio.microsoft.com/es/downloads/>

Postman - <https://www.postman.com/>