Diseño e Implementación de un Sistema FaaS: Una Arquitectura Basada en Contenedores Docker

Trabajo Final Cloud Computing

Daniel Herrero Pardo, Jorge Sánchez Lorente, Ismael Illán García

23 de enero de 2025

Índice

Índice de figuras

Índice de tablas

1. Objetivo del trabajo

El propósito de este trabajo es crear e implementar un sistema FaaS (Function as a Service) que habilite a los usuarios para registrar y ejecutar funciones bajo demanda a través de una arquitectura basada en contenedores Docker. Dicho sistema debe cumplir varios requisitos para asegurar la seguridad, modularidad y escalabilidad del servicio. Entre las características esenciales se encuentra la implementación de una API REST (HTTPS) que centralice todas las operaciones del sistema, incluyendo el registro y autenticación de usuarios, así como la gestión y ejecución de funciones. El diseño del sistema FaaS debe incorporar funciones clave, como la ejecución de funciones con parámetros y resultados en formato de cadena (string), el soporte para la eliminación de funciones por sus creadores, y la capacidad de manejar múltiples activaciones en paralelo con un límite ajustable. Además, se requiere una arquitectura que integre un proxy inverso para autenticación y balanceo de carga, un servidor de API para la lógica principal, una base de datos que almacene los usuarios y funciones registradas, y trabajadores responsables de ejecutar las funciones usando contenedores Docker. Esta solución busca ofrecer una plataforma robusta y flexible para ejecutar funciones en un entorno distribuido.

2. Arquitectura de los microservicios

La arquitectura que se propone emula un entorno de Function as a Service (FaaS) utilizando una estrategia de microservicios. En su centro, reside una API desarrollada con Express.js, que facilita la interacción de los usuarios con el servicio. A través de esta API, los usuarios administran sus funciones almacenadas como imágenes y solicitan su ejecución. La seguridad y autenticación se refuerzan mediante un proxy inverso configurado con Apache APISIX. Los datos de funciones y usuarios se concentran en una base de datos MongoDB, lo que proporciona un almacenamiento seguro y escalable.

Para la gestión de ejecuciones, se emplea una cola de mensajes basada en NATS, responsable de gestionar las solicitudes de los usuarios. Las peticiones son procesadas de manera asíncrona por un controlador de workers, que consume solicitudes de la cola y crea contenedores para ejecutar las funciones según lo solicitado. Este método garantiza disponibilidad alta, escalabilidad y capacidad para manejar diferentes cargas de trabajo, al realizar cada procesamiento en un contenedor separado. La integración de estos componentes da lugar a un entorno modular y eficaz que captura la flexibilidad y elementos esenciales de un servicio FaaS.

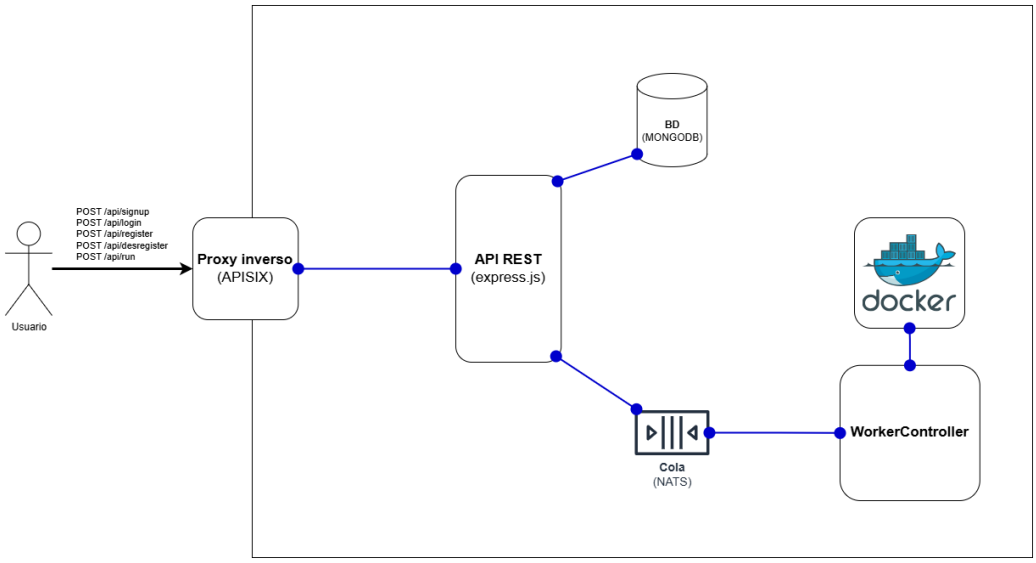

Figura 1: Arquitectura de los microservicios del sistema.

3. Desarrollo

3.1. Workers

En la solución propuesta, no se lleva a cabo la implementación directa de los Workers. En lugar de eso, se emplea un WorkerController que se encarga de leer los mensajes de la cola de ejecución de NATS, crear un contenedor para realizar la ejecución solicitada, extraer el resultado del contenedor y enviar la respuesta al usuario que requirió la ejecución. La estructura general se muestra en el diagrama 2. Basándonos en el diseño presentado, las ejecuciones de las imágenes pueden realizarse y procesarse de forma asincrónica, lo que permite manejar varios mensajes de la cola simultáneamente. Para gestionar la cola, dentro del controlador se ha creado la clase NatsQueue, que lleva a cabo las operaciones sobre la cola utilizando el módulo nats, facilitando el acceso a NATS mediante la implementación de objetos, funciones y métodos. En cuanto al manejo de los contenedores, el WorkerController utiliza Dockerode, un módulo de Node.js que a su vez hace uso de docker-modem para interactuar con la API de Docker. Para obtener el resultado, el WorkerController enlaza la salida estándar y la salida de error (stdout y stderr) a un stream. Cuando el contenedor termina su ejecución, el WorkerController lee el resultado obtenido y lo emplea como respuesta para el API Server.

4. Base de datos

Una configuración de MongoDB que incluye dos colecciones puede organizarse para manejar usuarios y sus roles de manera eficaz. La primera colección, llamada ‘users‘, guarda detalles de los usuarios, como sus nombres y contraseñas cifradas, lo que impide que estas se almacenen en texto plano, mejorando así la seguridad. Cada documento en esta colección contiene campos como ‘username‘, que es único, y ‘password‘. El cifrado de la contraseña puede realizarse utilizando bibliotecas como bcrypt antes de almacenar la información. La segunda colección, denominada ‘functions‘, asocia a cada usuario

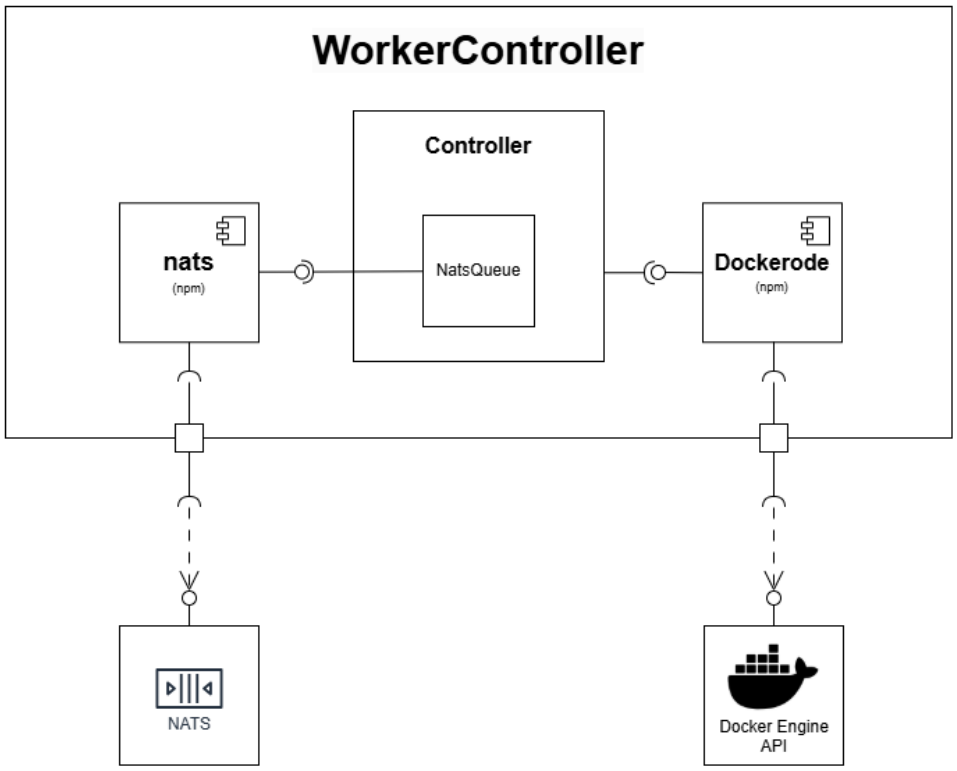

Figura 2: Diagrama del funcionamiento de los workers.

con sus roles específicos. Cada documento incluye referencias al usuario a través del ‘username‘ y el ID de una imagen Docker. Se crea un índice compuesto único que combina ‘username‘ e ‘image‘ para asegurar que un usuario no pueda registrar la misma función más de una vez. Esto permite realizar consultas de manera eficiente y evita la duplicación. Las dos colecciones funcionan conjuntamente: la colección ‘users‘ se encarga de la autenticación del usuario, mientras que ‘functions‘ gestiona las asignaciones de roles o permisos en el sistema.

5. Escalabilidad del sistema

5.1. Workers

Los contenedores generados por el controlador se sitúan en la instancia de Docker del servidor host y no en el contenedor del WorkerController. Esto asegura que sus recursos sean independientes de las ejecuciones que inicia, lo que permite tanto el escalamiento horizontal como vertical sin inconvenientes. Asimismo, al destinar todos los recursos del WorkerController únicamente a la recepción y procesamiento de mensajes, evitamos que ejecuciones demandantes lo saturen, asegurando que las nuevas solicitudes puedan ingresar sin problemas.

5.2. Base de datos

Para ampliar la base de datos implementada, se debe establecer un clúster de réplicas que incluya una principal para las escrituras y una o varias réplicas que sincronizan los datos de la principal. La base de datos está implementada con MongoDB, que permite el escalado horizontal a través del sharding. Este enfoque distribuye los datos entre varios nodos para manejar mayores cargas mediante la distribución de datos y operaciones. El sharding en MongoDB emplea tres componentes clave: shards, config servers, y mongos. Los shards son los nodos que almacenan los datos distribuidos, y generalmente cada uno es un conjunto de réplicas para asegurar alta disponibilidad. Los config servers

administran los metadatos del clúster, como la ubicación de los datos en los shards, y son fundamentales para el sharding. Finalmente, los mongos funcionan como encaminadores de consultas, dirigiendo las operaciones del cliente al shard apropiado según la clave de shard. Esta arquitectura facilita una escalabilidad horizontal efectiva y equilibrada.

6. Configuración para diferentes cargas de trabajo

La capacidad de responder dinámicamente a las variaciones de carga es crucial para garantizar el rendimiento y la escalabilidad en un sistema FaaS. Por ello, nuestro sistema prioriza esta capacidad de adaptación, permitiendo una respuesta eficaz ante picos de carga. Actualmente, el sistema se configura para desplegarse con docker-compose, lo que implica que la escalabilidad no se gestiona dinámicamente. Con esta consideración, el sistema requiere escalado manual. Para activar el escalado automático, se debería utilizar un orquestador de contenedores como Kubernetes o Docker Swarm e incluir algún tipo de Autoscaler para gestionar los recursos y el número de réplicas de cada componente. Gracias al diseño de nuestro sistema, no se necesita un balanceador de carga ni modificar configuraciones si se incrementan los recursos en los WorkerControllers, ya que estos reciben los mensajes directamente de la cola y ejecutan las funciones, permitiendo así desplegar múltiples réplicas sin problemas ni conflictos. Por otro lado, el API Server recibe mensajes del proxy inverso (APISIX), que está configurado para distribuir las solicitudes mediante el método round robin. Así que, para aumentar el número de réplicas, basta con añadir nuevos nodos a la configuración de APISIX. Alternativamente, se podría emplear un balanceador de carga entre APISIX y el API Server, siendo más sencillo con el uso de un orquestador de contenedores como se mencionó anteriormente.

7. Discusión sobre el acceso a servicios externos

En un entorno FaaS (Function as a Service), las funciones suelen depender de servicios externos para realizar tareas esenciales, como acceder a bases de datos, interactuar con APIs de terceros o comunicarse con sistemas de mensajería. Esta integración mejora significativamente las capacidades de las funciones, pero también introduce desafíos técnicos, de rendimiento y de seguridad que deben ser gestionados de manera adecuada.

Los servicios externos aportan varios beneficios importantes. En primer lugar, permiten simplificar las funciones al delegar tareas complejas a esos servicios, lo que reduce la cantidad de lógica que debe incluirse directamente en las funciones. Esto también contribuye a disminuir la carga de ejecución, ya que parte del procesamiento se desplaza hacia el servicio externo. Además, fomenta la modularidad al separar responsabilidades, lo que hace que las funciones sean más ligeras y más fáciles de mantener.

Sin embargo, el uso de servicios externos conlleva ciertos retos. Uno de los principales es la latencia, ya que cada interacción con un servicio externo introduce un retraso que puede afectar negativamente el tiempo de respuesta. Otro desafío es la fiabilidad: depender de servicios externos significa que un fallo en esos servicios puede dejar inactivas las funciones. También hay riesgos de seguridad, especialmente en el manejo de claves API, tokens de autenticación y otros datos sensibles, que deben gestionarse cuidadosamente para prevenir accesos no autorizados.

Para emplear servicios externos, es necesario realizar configuraciones específicas. Esto incluye proporcionar a la función los datos necesarios, como claves API o tokens de autenticación. Estos datos suelen incorporarse en la imagen desde la que se ejecuta la función, y la solicitud de ejecución debe incluirlos en el campo de parámetros como una cadena de texto.

Otro aspecto significativo del uso de servicios externos es el riesgo de fallos ajenos al control del sistema. Existen dos posibles enfoques para abordar este problema. Uno es implementar mecanismos automáticos de gestión de errores y reintentos. El otro consiste en permitir que el usuario gestione los reintentos manualmente en caso de error. Considerando que el usuario tiene visibilidad sobre los resultados y con el objetivo de evitar llamadas innecesarias a servicios externos, hemos optado por la segunda solución, dejando al usuario la responsabilidad de decidir cuándo realizar un nuevo intento.

8. Comparación con otras soluciones FaaS

Para optimizar la arquitectura y alinearla con otros servicios de FaaS, sería ventajoso incluir la capacidad de importar y almacenar funciones directamente como código fuente, en vez de limitarse al uso exclusivo de imágenes de contenedores. Esto permitiría a los usuarios escribir y subir sus funciones en diversos lenguajes de programación y entornos, como Node.js, Java, Python, o incluso Go, mejorando así la experiencia del usuario. Además, este almacenamiento podría integrarse con sistemas de control de versiones para posibilitar modificaciones y el seguimiento del historial de cambios en el código.

Desarrollar una interfaz gráfica o aplicación web también sería ventajoso, simplificando la interacción con la API del servicio. Tal interfaz debería facilitar la gestión de funciones, visualización de métricas—como tiempos de ejecución y uso de recursos—y supervisión de ejecuciones en tiempo real. Una centralización en el panel de control reforzaría el uso del servicio, actuando como un recurso esencial para que los usuarios administren y supervisen sus funciones con mayor facilidad.

Finalmente, implementar un sistema de disparadores basados en eventos permitiría la ejecución automática de funciones en respuesta a eventos particulares, como actualizaciones en bases de datos, mensajes en sistemas de colas como RabbitMQ o Kafka, o cambios en almacenamiento en la nube. Este enfoque emularía características esenciales de servicios como AWS Lambda y Google Cloud Functions, facilitando al mismo tiempo una integración fluida con otros sistemas.

Asimismo, proporcionar soporte para configuraciones avanzadas, como trabajos cron para ejecuciones programadas, ampliaría significativamente las capacidades del sistema y su atractivo para aplicaciones empresariales y de automatización.

El aspecto que diferencia notablemente a nuestro FaaS de los demás es la capacidad de ejecutar imágenes Docker completas, en vez de limitarse solo al código. Por esta razón, podemos realizar la comparación de la tabla 1.

Aspecto	**Nuestro FaaS**	**Otros FaaS**
Flexibilidad del entorno	Total (completo control sobre la imagen)	Limitado a los entornos soportados por el proveedor
Lenguajes soportados	Todos los lenguajes compatibles con Docker	Lista predefinida de lenguajes
Portabilidad	Alta, gracias a estándares Docker	Limitada al ecosistema del proveedor
Integración con DevOps	Excelente (integración con CI/CD basada en contenedores)	Limitada a herramientas específicas del proveedor
Tiempos de arranque (cold start)	Potencialmente más lentos debido al tamaño de las imágenes	Optimizado para funciones ligeras
Facilidad de adopción	Requiere conocimientos de Docker	Más simple para proyectos pequeños
Uso de recursos	Configurable mediante Dockerfile	Basado en configuraciones del proveedor

Tabla 1: Comparación entre nuestro FaaS y otros FaaS.