FaaS. Function as a Service

José Manuel Bernabeu Aubán

Brian Valiente Ródenas.
Álvaro López Álvarez.

21 de febrero de 2025

Índice

1. Descripción de la Arquitectura de Microservicios

La arquitectura de nuestro sistema FaaS está basada en una serie de **microservicios** que interactúan entre sí para ofrecer un servicio flexible y escalable para la ejecución de funciones. Algunos de los microservicios usan variables situadas en el .env, que deberás crear donde se te indique. Los microservicios clave en la arquitectura son los siguientes:

1.1. APISIX (Reverse Proxy)

- **Función**: APISIX actúa como un proxy inverso que maneja la autenticación de los usuarios mediante JWT, además de gestionar el balanceo de carga entre instancias del **API Server**. Recibe las peticiones de los usuarios y las redirige a su endpoint correspondiente balanceando la carga.
- **Tecnología**: Apache APISIX (`https://apisix.apache.org/`).
- **Puertos expuestos**: 9080 (HTTP)
- **Configuración de Rutas**: Las rutas en APISIX están configuradas con un **balanceo de carga round-robin**, lo que permite distribuir las solicitudes de manera equitativa entre las instancias del API Server. Si se replicara el API server, se debería crear un nuevo "upstream" de Apisix para que el balanceador de carga decida qué servidor debe proporcionar al usuario.
- **Plugins de Seguridad**: APISIX utiliza los plugins `jwt-auth` y `key-auth` para gestionar la autenticación:
  - `jwt-auth`: Se utiliza para autenticar a los usuarios mediante tokens JWT. Los consumidores están definidos con claves específicas (`key`) y secretos (`secret`) configurados en APISIX, permitiendo validar los tokens generados por el servicio de autenticación.
  - `key-auth`: Ofrece un mecanismo adicional para autenticar solicitudes mediante claves API.
- **Configuración de Consumidores**: Se define un consumidor llamado `faas_jwt_consumer`, que está vinculado al plugin `jwt-auth`. Este consumidor utiliza un secreto 'SECRET' (localizado en el .env de la carpeta ApiServer) compartido con el servicio de autenticación del sistema para validar los tokens JWT y garantizar el acceso seguro a las rutas protegidas.
- **Plugins de Tráfico**: Para evitar que un usuario lance demasiadas peticiones en poco tiempo, utilizamos el plugin "limit-count" configurado de tal manera que un usuario no puede lanzar más de 2 peticiones a cualquier ruta del API en un lapso de 10 segundos, de lo contrario saltará un error HTTP 429 que indica que se ha superado el límite de peticiones seguidas.

1.2. API Server

- **Función**: El API Server expone los puntos finales de la API REST, que permiten la creación y gestión de usuarios, funciones, y la ejecución de las mismas.
- **Tecnologías**: Lenguaje Go.
- **Puertos expuestos**: 8080.
- **Dependencias**: Depende de NATS para la comunicación con los workers.
- **Escalabilidad**: Puede escalarse añadiendo más réplicas de instancias si es necesario para gestionar una mayor carga. Si se escala a otra máquina distinta, hay que ajustar el archivo de configuración de rutas ("apisix_confroutes.yaml") de apisix para que pueda hacer load balancing sobre el servidor API.

1.3. NATS (Message Queue and Key-Value Store)

- **Función**: NATS gestiona la cola de mensajes, permitiendo la comunicación entre los microservicios, especialmente entre el API Server y los Workers. Además, con su función de clave-valor registramos usuarios y funciones.
- **Tecnología**: NATS (`https://nats.io/`).
- **Puertos expuestos**: 4222.
- **Escalabilidad**: NATS puede escalarse horizontalmente añadiendo más instancias, lo que podría mejorar el rendimiento y la capacidad de gestionar un mayor volumen de mensajes (se podría hacer un cluster de NATS).

1.4. Workers

- **Función**: Los Workers son responsables de ejecutar las funciones en contenedores Docker y devolver los resultados al API Server.
- **Dependencias**: Depende de NATS y el ApiServer para llevar a cabo la ejecución de funciones solicitadas por los usuarios.
- **Tecnología**: Go, Docker.
- **Escalabilidad**: Se han configurado 7 réplicas de los Workers para manejar solicitudes concurrentes. El número de réplicas se puede ajustar manualmente según la carga del sistema.

1.5. Uso de Docker y Docker Compose

El sistema está **contenedorizado** usando **Docker** y gestionado mediante **Docker Compose**. Docker permite que cada componente del sistema (API Server, Workers, NATS, APISIX) se ejecute en contenedores aislados, lo que facilita la implementación y el escalado de los microservicios. Docker Compose se usa para orquestar los contenedores, definir redes entre ellos y gestionar los volúmenes persistentes (por ejemplo, el almacenamiento de NATS).

1.6. Grado de Replicación

Para ejecutar el FaaS, **docker compose up –build** y si deseamos especificar las réplicas de un servicio **docker compose up –scale worker=10**

- **API Server**: 2 réplicas por defecto, aunque podría escalarse manualmente agregando más instancias si es necesario y configurando en Apisix las rutas para poder redirigir el tráfico si se replica en varias máquinas distintas.
- **Workers**: 7 réplicas configuradas inicialmente (en la opción *replicas* de docker compose), con la capacidad de añadir más instancias según la carga del sistema manualmente desde –scale o modificando el archivo docker compose.
- **NATS**: Se podría replicar horizontalmente para mejorar la disponibilidad y el rendimiento si es necesario (formando clústeres NATS).

1.7. Elasticidad (Variación de la carga)

La elasticidad del sistema se gestiona principalmente mediante la **configuración de réplicas** de los microservicios, especialmente los **Workers**, que se pueden replicar para manejar una mayor cantidad de activaciones simultáneas de funciones.

Además, las rutas de **APISIX** utilizan **balanceo de carga round-robin** para distribuir las solicitudes de forma equitativa entre las instancias del **API Server**. Este balanceo asegura que las

solicitudes se gestionen de manera eficiente incluso cuando el número de solicitudes concurrentes aumenta.

En cuanto al escalado automático, aunque no se ha implementado una solución de escalado automático en esta arquitectura, el número de réplicas de los microservicios puede ajustarse manualmente según sea necesario, lo que permite al sistema adaptarse a cargas variables.

2. Uso del ApiServer

Esta API permite gestionar usuarios, autenticación y funciones, además de ejecutar acciones sobre las funciones registradas. Usaremos el puerto **9080**, usado por Apisix, para redirigir a la API las peticiones. Estos son los endpoints disponibles:

2.1. Registrar un usuario

- **Método**: POST
- **Ruta**: `/register`
- **Cuerpo**:

```
{
    "Username": "tuUsuario",
    "Password": "tuContraseña"
}
```

- **Descripción**: Este endpoint permite registrar un nuevo usuario enviando su nombre de usuario y contraseña.
- **Consideraciones**: El nombre de usuario NO debe utilizar los siguientes carácteres especiales: '.', '*', '¿', '$'
- **Ejemplo**:

```
curl -X POST "http://localhost:9080/register" \
  -H "Content-Type: application/json" \
  -d '{"Username": "testUser", "Password": "1234"}'
```

2.2. Iniciar sesión

- **Método**: POST
- **Ruta**: `/login`
- **Cuerpo**:

```
{
    "Username": "tuUsuario",
    "Password": "tuContraseña"
}
```

- **Descripción**: Este endpoint permite iniciar sesión y obtener un token JWT para autenticación en futuras solicitudes.
- **Ejemplo**:

```
curl -X POST "http://localhost:9080/login" \
  -H "Content-Type: application/json" \
  -d '{"Username": "testUser", "Password": "1234"}'
```

2.3. Verificar autenticación

- **Método**: GET
- **Ruta**: `/validate`
- **Encabezado o Cookie**: Incluir el token JWT obtenido al iniciar sesión es opcional, ya se incluye de forma automática.
- **Descripción**: Sirve para verificar que el token de autenticación funciona correctamente
- **Ejemplo**:

```
curl -X GET "http://localhost:9080/validate" \
  --cookie "Authorization=tuTokenJWT"
```

2.4. Registrar una función

- **Método**: POST
- **Ruta**: `/registerFunction`
- **Cuerpo**:

```
{
    "Name": "nameFunction",
    "Data": "dockerImage"
}
```

- **Descripción**: Este endpoint permite registrar una nueva función, proporcionando su nombre y la imagen docker que contiene esa función hecha por el usuario y publicada en Docker Hub `https://hub.docker.com/`.
- **Ejemplo**:

```
curl -X POST "http://localhost:9080/registerFunction" \
  -H "Content-Type: application/json" \
  --header 'Authorization: Bearer <jwt-token>\
  -d '{"Name": "cuenta_palabras", \
  "Data": "bvalrod/cuenta_palabras:v2"}'
```

2.5. Ejecutar una función

- **Método**: POST
- **Ruta**: `/execute`
- **Cuerpo**:

```
{
    "FuncID": "idDeLaFuncion",
    "Parameter": "valorDelParametro"
}
```

- **Descripción**: Permite ejecutar una función registrada previamente, enviando su ID y los parámetros necesarios para la ejecución.
- **Ejemplo**:

```
curl -X POST "http://localhost:9080/execute" \
  -H "Content-Type: application/json" \
  --header 'Authorization: Bearer <jwt-token>\
  -d '{"FuncID": "cuenta_palabras123456789", \
  "Parameter": "{"frase": "Hola que tal"}"}'
```

2.6. Eliminar una función

- **Método**: DELETE
- **Ruta**: `/deleteFunction/:id`
- **Descripción**: Este endpoint permite eliminar una función específica, identificada por su ID.
- **Ejemplo**:

```
curl -X DELETE "http://localhost:9080/deleteFunction/cuenta_palabras123456789"\
    --header 'Authorization: Bearer <jwt-token>
```

2.7. Obtener todas las funciones

- **Método**: GET
- **Ruta**: `/getFunctions`
- **Descripción**: Permite recuperar todas las funciones registradas por el usuario autenticado.
- **Ejemplo**:

```
curl -X GET "http://localhost:9080/getFunctions"\
    --header 'Authorization: Bearer <jwt-token>\
```

2.8. Obtener todos los usuarios (solo admin)

- **Método**: GET
- **Ruta**: `/users`
- **Descripción**: Solo los administradores pueden acceder a este endpoint. Cuando te registres por primera vez, si tu *username* coincide con el ADMIN_USER localizado en el .env de la carpeta ApiServer, se te asigna el rol de admin. Recupera la lista de todos los usuarios registrados.
- **Ejemplo**:

```
curl -X GET "http://localhost:9080/users"\
    --header 'Authorization: Bearer <jwt-token>\
```

Resumen de Endpoints

En los endpoints que requieren autenticación (todos excepto `/register` y `/login`), el token JWT se incluye de forma automática en las cookies bajo el nombre `Authorization` (si se utiliza una aplicación como puede ser Postman para enviar peticiones). Esto es para validar la identidad del usuario durante su sesión.

- **POST** `/register`: Registra un nuevo usuario.
- **POST** `/login`: Inicia sesión y obtiene un token JWT.
- **GET** `/validate`: Verifica que el usuario esté autenticado.
- **POST** `/registerFunction`: Registra una nueva función.
- **POST** `/execute`: Ejecutar una función.
- **DELETE** `/deleteFunction/:id`: Elimina una función específica.
- **GET** `/getFunctions`: Recupera todas las funciones registradas por el usuario.
- **GET** `/users`: Obtiene todos los usuarios registrados (solo para admins).

Cada uno de estos endpoints está protegido por autenticación JWT y cuando se inicia sesión el token se envía correctamente en las cookies para acceder a funciones restringidas.

3. Descripción del Sistema como un Archivo Docker-Compose

El archivo `docker-compose.yml` define la arquitectura de microservicios para un sistema Function-as-a-Service (FaaS). Utiliza contenedores para desplegar los componentes necesarios, asegurando modularidad y escalabilidad. En nuestro caso, tenemos los siguientes elementos:

Servicios

- **nats:**
  - Es el *broker de mensajes* que permite la comunicación entre los servicios.
  - Se activa JetStream (`--js`) para garantizar la persistencia de mensajes de nuestra cola (vista por los *workers*), y los datos se almacenan en un volumen compartido (`faas_data`).
- **apisix:**
  - Actúa como el *proxy inverso* y *gestor de API*.
  - Maneja las solicitudes entrantes, autentica usuarios y realiza balanceo de carga.
  - Las configuraciones de rutas y plugins se proporcionan mediante archivos de configuración mediante el modo "standalone" de apisix que no requiere del uso de etcd para guardar la configuración del servicio.
  - Mediante **port-forwarding** expone el puerto 9080 a la máquina host para ser el único punto de entrada del servicio FaaS.
- **apiServer:**
  - Implementa la API REST para gestionar funciones y usuarios.
  - Se comunica con `nats` para gestionar la cola de solicitudes y respuestas. Mediante un stream de tipo "workingQueue" se suben las peticiones realizadas al API para ser manejadas posteriormente por los "workers".
- **worker:**
  - Procesa las solicitudes de ejecución de funciones.
  - Se configuran *múltiples réplicas* (`deploy.replicas: 7`) para manejar cargas de trabajo en paralelo, asegurando escalabilidad.

Volúmenes y redes

- **Volúmenes:** El volumen `faas_data` almacena los datos del servicio, como el estado de JetStream, persistiendo la información incluso si los contenedores se reinician.
- **Redes:** Una red de tipo puente (`faas-network`) conecta todos los servicios, asegurando una comunicación segura y aislada.

Puertos y dependencias

- **Puertos:**
  - APISIX expone los puertos `9080` (HTTP) y `9443` (HTTPS) para acceso externo.
- **Dependencias:**
  - `apiServer` depende de `nats` para que las colas de mensajes estén listas antes de procesar solicitudes.
  - `worker` depende de `nats` y `apiServer` para operar correctamente.

4. Informe del Sistema

4.1. Cómo funciona el apiServer

El apiServer permite realizar funciones CRUD sobre usuarios y funciones de forma sencilla por medio de consultas API REST. Para acceder al API, el usuario primero debe registrarse, para a continuación, autenticarse (mediante la ruta /login).

Tras autenticarse, el usuario puede utilizar el token recibido al autenticarse para realizar las peticiones protegidas del API como son el registro de funciones (/registerFunction), la ejecución de funciones (/execute), el listado de funciones (/getFunctions) y más.

Cuando un usuario registra una función, por detrás el sistema se trae la imagen docker desde DockerHub (si no la encuentra, salta un error ErrFunctionNotPulled, del archivo /service/functions/errors.go).

Cuando un usuario ha registrado una función, puede utilizar su functionIdentifier obtenido al registrar la función para ejecutarla por medio de la ruta /execute. Esta ruta lanza un mensaje a un Stream de nats de nombre "jobs" para que sea manejado por cualquiera de las instancias de workers que hay en ejecución. Cuando un worker acaba de procesar la petición, devuelve un ACK al servidor para indicarle el éxito en el procesado de la función. El servidor se subscribe a un tema que comparten el worker y el apiServer para recibir la respuesta del worker.

4.2. Cómo funcionan los Workers

Los workers son instancias de Go que se conectan al servidor Nats y se quedan esperando mensajes del stream "jobs". Cuando aparece un mensaje, este contiene información sobre la imagen de la función a ejecutar, el parámetro que se debe pasar a esa función y un tema único que conocen el servidor api y el worker para que este último pueda comunicar la respuesta de la petición al API.

Tras recibir el mensaje, el worker lanza un contenedor docker con la imagen y el parámetro especificados y se queda a la espera de recibir el resultado. Cuando recibe el resultado de la ejecución lo envía al tema único recibido por el mensaje Nats y hace ACK para que el servidor Nats pueda eliminar ese mensaje de la cola de mensajes.

4.3. Ventajas y Desventajas

Las ventajas del uso del servicio FaaS son claras, el desarrollador puede abstraer los pasos de instalación en su ordenador en forma de imagen Docker para ser ejecutada por un servidor mucho más potente que su máquina personal. El servicio consigue la seguridad de las funciones registradas evitando cualquier acceso de otro usuario que no haya registrado la función y permite al usuario el manejo sencillo de las funciones por medio de la ruta /getFunctions para listar las funciones que tiene registradas.

El escalado del servicio es sencillo, se puede configurar el número de réplicas tanto de apiServer como de workers desde el archivo de despliegue de Docker, el docker-compose.

El parámetro de entrada para la ejecución de funciones es un string, lo cual lo hace flexible para introducir estructuras complejas con JSON por ejemplo para funciones que requieran varios parámetros o parámetros complejos.

Las desventajas aparecen cuando se requiere un escalado complejo. Si se requiere escalar el servidor a varias máquinas distintas, la arquitectura se volverá más compleja y se requerirán cambios en archivos de configuración de apisix y en el docker-compose.

Otra desventaja del sistema es que al almacenar los datos en un Key-Value store, hace más complejo buscar funciones y usuarios y es por eso que para ejecutar una función, se debe utilizar el "functionIdentifier" y el usuario debe copiar este identificador o utilizar la ruta /getFunctions para buscar la función que quiere ejecutar. Además, el usuario puede registrar varias funciones con la misma imagen, lo cual puede no tener mucho sentido.