UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS DE VERA

FaaS
Function as a Service

Cloud Computing

HÉCTOR LÓPEZ TERRADES
GERARD ROIG CLÉRIES

Índice

1. Introducción

En este proyecto se ha diseñado e implementado una arquitectura FaaS (Functions as a Service) basada en tecnologías de código abierto como Apache APISIX, Node.js y NATS. Este sistema permite a los usuarios gestionar funciones de manera dinámica, proporcionando funcionalidades de registro, ejecución y eliminación de funciones mediante una API RESTful expuesta a través de un gateway configurado con Apache APISIX.

La idea principal detrás de este trabajo es crear una solución escalable y extensible que pueda ser utilizada como base para futuras aplicaciones basadas en el paradigma serverless. Además, el proyecto implementa un sistema de mensajería mediante NATS, que facilita la ejecución de funciones de manera eficiente y asíncrona.

Este documento describe detalladamente el diseño del sistema, su implementación, los pasos necesarios para su despliegue, y las pruebas realizadas para validar su correcto funcionamiento.

2. Arquitectura del Sistema

El sistema implementado sigue un enfoque basado en microservicios, donde los componentes interactúan a través de interfaces bien definidas, asegurando modularidad, escalabilidad y facilidad de mantenimiento. A continuación, se presenta una descripción detallada de la arquitectura.

2.1 Descripción General

El sistema está compuesto por tres componentes principales:

2.1.1 Apache APISIX:

- **Función principal:** Gateway API que actúa como punto de entrada para todas las solicitudes del cliente. Su principal objetivo es enrutar las solicitudes hacia los servicios backend adecuados.
- Configuración clave:
  - Define rutas específicas para operaciones como registro de usuarios, inicio de sesión, gestión y ejecución de funciones.
  - Balanceo de carga entre múltiples instancias backend (si se necesitan en el futuro).
  - Habilitación de autenticación mediante tokens JWT para garantizar la seguridad.

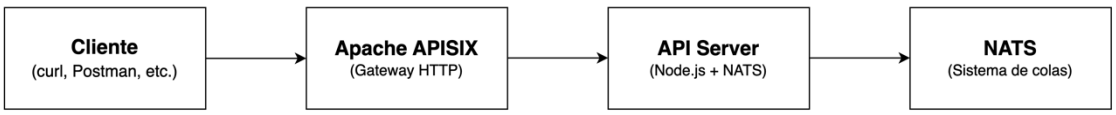
2.1.2 API Server:

- **Función principal:** Servicio backend desarrollado en Node.js que gestiona toda la lógica de negocio del sistema.
- Principales responsabilidades:
  - Registro de usuarios con almacenamiento seguro de contraseñas.
  - Autenticación y generación de tokens JWT.
  - Registro, ejecución y eliminación de funciones en un entorno seguro.
  - Comunicación con NATS para gestionar las solicitudes de ejecución de funciones de forma asíncrona.

2.1.3 NATS:

- Sistema de mensajería ligero y de alto rendimiento.
- Permite la comunicación eficiente entre servicios mediante tópicos (canales de mensajes).
- Garantiza la separación entre las solicitudes de los clientes y la lógica de ejecución de funciones.

2.2 Diagrama de Arquitectura



2.3 Apache APISIX

Apache APISIX es un componente crítico que sirve como punto de entrada al sistema. Se encarga de recibir solicitudes HTTP, aplicar políticas de enrutamiento y delegar las solicitudes al backend adecuado.

2.3.1 Funciones clave

2.3.1.1 Enrutamiento:

- Define rutas HTTP que redirigen las solicitudes del cliente hacia el API Server.
- Soporta patrones de URI como /auth/* para manejar múltiples operaciones dentro de un módulo.

2.3.1.2 Balanceo de carga:

- Si se despliegan múltiples instancias del API Server, APISIX distribuye las solicitudes entre ellas.

2.3.1.3 Seguridad:

- Implementación de autenticación mediante validación de tokens JWT en las rutas.
- Restricción de acceso a la API de administración para direcciones IP específicas.

2.3.1.4 Monitorización y métricas:

- APISIX tiene soporte nativo para integrar con herramientas como Prometheus y Grafana, facilitando la observación del sistema.

2.3.2 Rutas configuradas:

- [POST]/auth/register: Registro de usuarios.
- [POST]/auth/login: Inicio de sesión.
- [POST]/functions: Registro de funciones.
- [POST]/functions/{id}/execute: Ejecución de funciones.
- [DELETE]/functions/{id}: Eliminación de funciones.

2.4 API Server: Lógica de Negocio

El API Server es el núcleo del sistema y se implementa en Node.js utilizando bibliotecas como express, jsonwebtoken y nats. Es responsable de manejar todas las operaciones principales del sistema.

Funciones principales del API Server:

2.4.1 Gestión de Usuarios:

- **Registro:** Permite registrar nuevos usuarios almacenando las contraseñas de forma segura (encriptadas con bcrypt).
- **Inicio de sesión:** Valida las credenciales de usuario y genera un token JWT que permite a los clientes realizar operaciones autenticadas.

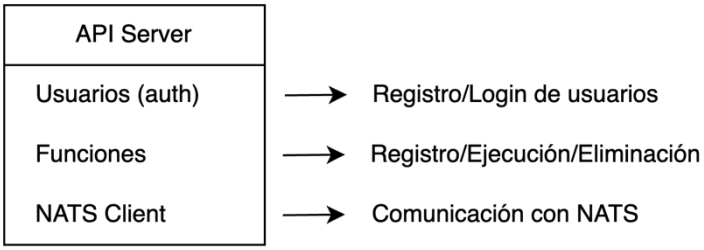
2.4.2 Gestión de Funciones:

- **Registro:** Almacena el código fuente de una función asociada a un usuario.
- **Ejecución:** Publica una solicitud en NATS para que la función sea ejecutada de forma asíncrona.
- **Eliminación:** Borra una función registrada.

2.4.3 Comunicación con NATS:

- Publica mensajes en el canal functions.execute cuando se solicita la ejecución de una función.
- Recibe resultados a través del canal functions.result y los envía de vuelta al cliente.

Diagrama interno del API Server:



2.5 NATS

NATS es un sistema de mensajería que permite el intercambio de información entre los servicios. Es especialmente útil para implementar sistemas desacoplados y escalables.

2.5.1 Tópicos utilizados:

2.5.1.1 functions.execute:

- Publicado por el API Server cuando un cliente solicita la ejecución de una función.
- Contiene los datos necesarios para ejecutar la función, como el código fuente y los parámetros de entrada.

2.5.1.2 functions.result:

- Publicado por el servicio que ejecuta la función con el resultado de la operación.
- Escuchado por el API Server para devolver el resultado al cliente.

2.5.2 Ventajas de usar NATS:

- Alta velocidad: Procesa millones de mensajes por segundo con baja latencia.
- Escalabilidad: Permite añadir más servicios sin modificar la lógica de los componentes existentes.
- Simplicidad: La API de NATS es sencilla de usar y facilita la implementación de patrones de comunicación como pub/sub y request/reply.

2.5.3. Ejemplo de flujo en NATS:

1. El cliente solicita la ejecución de una función enviando una solicitud HTTP al API Server
2. El API Server publica un mensaje en el canal functions.execute.
3. Un servicio escucha en ese canal, ejecuta la función y publica el resultado en functions.result.
4. El API Server recibe el resultado y lo devuelve al cliente.

3. Implementación

La implementación del sistema se ha llevado a cabo utilizando tecnologías modernas y herramientas de contenedores para garantizar escalabilidad, modularidad y facilidad de despliegue. A continuación, se describen las decisiones clave de implementación, los pasos realizados y los componentes desarrollados.

3.1 Organización del Proyecto

El proyecto está organizado en tres carpetas principales:

3.1.1. api-server:

- Contiene el código fuente del servicio backend, implementado en Node.js.
- Gestiona la autenticación, el registro y la ejecución de funciones.
- Incluye el archivo Dockerfile para crear su imagen de contenedor.

3.1.2. apisix_conf:

- Contiene la configuración de Apache APISIX.
- El archivo clave es config.yaml, que define las conexiones con el API Server.

3.1.3. nats_conf:

- Incluye la configuración para desplegar el sistema de mensajería NATS.
- Define el uso de JetStream y las configuraciones básicas en el archivo nats.conf.

Además, en el directorio raíz se encuentran los siguientes archivos clave:

- docker-compose.yaml: Orquesta el despliegue de todos los servicios.
- init_routes.sh: Script para inicializar las rutas de APISIX.

3.2 Implementación del API Server

El API Server es el núcleo del sistema y sigue una estructura modular para facilitar su escalabilidad y mantenibilidad.

3.2.1 Estructura del API Server:

```
api-server/
├── routes/                 # Define las rutas principales
│   ├── auth.js             # Gestión de usuarios (registro y login)
│   ├── functions.js        # Registro, ejecución y eliminación de funciones
├── services/               # Servicios de soporte para el API Server
│   ├── nats.js             # Comunicación con NATS
│   ├── db.js               # Interacción con la base de datos en memoria
├── server.js               # Punto de entrada del servidor
├── Dockerfile              # Archivo para construir la imagen del contenedor
├── package.json            # Dependencias y scripts del proyecto
└── .env                    # Variables de entorno (como la URL de NATS)
```

3.2.2 Descripción de Módulos

3.2.2.1 server.js:

- Punto de entrada principal del API Server.
- Carga las rutas desde la carpeta routes/.
- Configura el servidor Express y las dependencias.

3.2.2.2 routes/:

Contiene los controladores que definen las rutas HTTP.

- auth.js:
  - **POST** /auth/register: Registra nuevos usuarios.
  - **POST** /auth/login: Genera tokens JWT tras validar credenciales.
- functions.js:
  - **POST** /functions: Registra una nueva función.
  - **POST** /functions/:id/execute: Solicita la ejecución de una función.
  - **DELETE** /functions/:id: Elimina una función.

3.2.2.3 services/

- nats.js:
  - Define la lógica para publicar y suscribirse a mensajes en NATS.
  - Publica solicitudes de ejecución en el canal functions.execute.
  - Escucha resultados de ejecución en el canal functions.result.
- db.js:
  - Implementa una base de datos en memoria para almacenar usuarios y funciones.
  - Proporciona métodos como registerUser, authenticateUser, y registerFunction.

3.2.2.4 Dockerfile:

Contenediza el API Server para su despliegue con Docker.

Ejemplo del contenido:

```
# Usar una imagen base de Node.js
FROM node:18

# Crear y movernos al directorio del contenedor
WORKDIR /app

COPY package*.json ./

# Instalar dependencias
RUN npm install

# Copia el resto de los archivos del proyecto
COPY . .

# Exponer el puerto
EXPOSE 3000

# Comando para iniciar el servidor
CMD ["node", "server.js"]
```

3.3 Configuración de Apache APISIX

Apache APISIX actúa como puerta de enlace de API (API Gateway) para enrutar las solicitudes entrantes hacia el api-server según las rutas configuradas dinámicamente. Este subsistema utiliza el archivo de configuración apisix_conf/config.yaml y permite la gestión de rutas mediante la API administrativa de APISIX.

Estructura del Archivo apisix_conf/config.yaml

El archivo config.yaml contiene los ajustes necesarios para inicializar y gestionar el funcionamiento de APISIX. A continuación, se detalla su contenido y propósito:

3.3.1 Configuración General

```
apisix:
  node_listen: 9080  # Puerto donde APISIX escucha las solicitudes externas
  enable_ipv6: false # Desactiva el soporte para IPv6
  enable_control: true # Habilita el panel de control de APISIX
  control:
    ip: "0.0.0.0"    # El panel de control estará accesible en todas las
interfaces de red
    port: 9092       # Puerto del panel de control
```

- Puerto de Escucha (node_listen):
  APISIX escucha en el puerto 9080 para manejar solicitudes externas, permitiendo que los clientes accedan a los servicios configurados.
- Panel de Control (control):
  El panel de control se habilita en el puerto 9092, accesible desde cualquier interfaz IP.

3.3.2 Configuración Administrativa

```
deployment:
  admin:
    allow_admin:
      - 0.0.0.0/0  # Permite acceso administrativo desde cualquier IP (solo
para desarrollo)
    admin_key:
      - name: "admin"
        key: edd1c9f034335f136f87ad84b625c8f1
        role: admin
      - name: "viewer"
        key: 4054f7cf07e344346cd3f287985e76a2
        role: viewer
```

- Acceso Administrativo (allow_admin):
  Permite el acceso a la API administrativa de APISIX desde cualquier dirección IP.
- Llaves de Administración (admin_key):
  Define dos usuarios:
  - admin: Tiene permisos completos para gestionar configuraciones.
  - viewer: Solo puede ver las configuraciones, sin modificarlas.

3.3.3 Configuración de ETCD

```
etcd:
  host:
    - "http://etcd:2379"  # Dirección del servidor ETCD utilizado por APISIX
  prefix: "/apisix"       # Prefijo de las claves en ETCD para almacenar
configuraciones
  timeout: 30             # Tiempo de espera para conexiones a ETCD (en
segundos)
```

ETCD es un sistema de almacenamiento clave-valor distribuido que APISIX utiliza para guardar configuraciones como rutas, upstreams y plugins, así mismo garantiza la interacción de APISIX con el servidor ETCD disponible en la red.

3.3.4 Gestión Dinámica de Rutas

Las rutas se crean dinámicamente mediante la API administrativa de APISIX utilizando el script init_routes.sh, en lugar de preconfigurarse en el archivo config.yaml. Esto permite modificarlas sin reiniciar APISIX.

Ejemplo de Creación de Rutas

El script init_routes.sh utiliza curl para interactuar con la API administrativa de APISIX y configurar las rutas dinámicamente. Ejemplo:

```
#!/bin/bash

# Ruta para autenticar usuarios
curl -i -X PUT http://localhost:9180/apisix/admin/routes/1 \
-H "X-Api-Key: edd1c9f034335f136f87ad84b625c8f1" \
-H "Content-Type: application/json" \
-d '{
  "uri": "/auth/login",
  "methods": ["POST"],
  "upstream": {
    "type": "roundrobin",
    "nodes": {
      "api-server:3000": 1
    }
  }
}'

# Ruta para gestionar funciones
curl -i -X PUT http://localhost:9180/apisix/admin/routes/2 \
-H "X-Api-Key: edd1c9f034335f136f87ad84b625c8f1" \
-H "Content-Type: application/json" \
-d '{
  "uri": "/functions/*",
  "methods": ["POST"],
  "upstream": {
    "type": "roundrobin",
    "nodes": {
      "api-server:3000": 1
    }
  }
}'
```

3.4 Configuración de NATS

La configuración de NATS se encuentra definida en el archivo nats_conf/nats.conf, que asegura el correcto funcionamiento del sistema de mensajería entre los distintos componentes del proyecto. NATS, como middleware de mensajería, es crucial para permitir la comunicación asíncrona y la ejecución distribuida de funciones.

3.4.1. Monitoreo

El parámetro http_port: 8222 habilita una interfaz de monitoreo HTTP en el puerto 8222. Proporciona métricas y estadísticas en tiempo real sobre el servidor NATS, como conexiones activas, tópicos publicados y recursos utilizados. Facilita el diagnóstico de problemas de rendimiento o la detección de cuellos de botella en el sistema de mensajería.

3.4.2 JetStream

La sección jetstream activa JetStream, la extensión avanzada de NATS para la gestión de streams y almacenamiento persistente. El atributo store_dir: “/data/jetstream” especifica el directorio para los datos persistentes, permitiendo almacenar mensajes en disco y garantizando la durabilidad en tareas críticas y la confiabilidad del sistema ante fallos.

3.4.3 Límites de Mensajes

El parámetro max_payload: 10MB define el tamaño máximo de los mensajes que pueden publicarse en el servidor NATS. Esto evita la saturación de la red y del sistema de almacenamiento por mensajes demasiado grandes. Un límite de 10MB resulta adecuado, dado que los mensajes habituales, como código de funciones y datos de entrada, son pequeños.

3.4.4 Autorización

El bloque authorization define permisos globales predeterminados para todos los clientes que interactúan con el servidor NATS. Con la configuración:

```
default_permissions {
  publish = ">"
  subscribe = ">"
}
```

- publish = ">": Permite a los clientes publicar mensajes en cualquier tema.
- subscribe = ">": Autoriza a los clientes a suscribirse a cualquier tema.

Aunque es permisiva y facilita el desarrollo inicial, en un entorno de producción conviene restringir los permisos para garantizar la seguridad y prevenir accesos no autorizados a tópicos sensibles.

Asegura que el servidor NATS esté optimizado para los requerimientos del proyecto, permitiendo una comunicación efectiva y confiable entre los servicios.

4. Pruebas

Se han diseñado y ejecutado pruebas para validar las funcionalidades del sistema:

4.1 Registro de Usuarios

Registra un usuario para comenzar a utilizar el sistema:

```
curl -X POST http://localhost:9080/auth/register \
-H "Content-Type: application/json" \
-d '{"username": "testuser", "password": "password123"}'
```

Respuesta esperada:

```
{"message": "Usuario registrado exitosamente"}
```

4.2. Inicio de sesión

Obtén un token JWT para el usuario registrado:

```
curl -X POST http://localhost:9080/auth/register \
-H "Content-Type: application/json" \
-d '{"username": "testuser", "password": "password123"}'
```

Respuesta esperada:

```
{"token": "<jwt-token>"}
```

4.3. Registrar una función

Usa el token obtenido para registrar una función:

```
curl -X POST http://localhost:9080/functions \
-H "Content-Type: application/json" \
-H "Authorization: Bearer <jwt-token>" \
-d '{"user": "testuser", "code": "return input + 1;"}'
```

Respuesta esperada:

```
{"id": 1, "code": "return input + 1;"}
```

4.4. Ejecutar una función

Ejecuta la función registrada proporcionando un valor de entrada:

```
curl -X POST http://localhost:9080/functions/1/execute \
-H "Content-Type: application/json" \
-H "Authorization: Bearer <jwt-token>" \
-d '{"input": 10}'
```

Respuesta esperada:

```
{"result": 11}
```

4.5. Eliminar una función

Elimina la función registrada proporcionando el id de la misma:

```
curl -X DELETE http://localhost:9080/functions/1 \
-H "Authorization: Bearer <jwt-token>"
```

Respuesta esperada:

```
{"message": "Función eliminada"}
```

5. Diseño de Bases de Datos

NATS JetStream se utiliza como base de datos para almacenar información de usuarios y funciones, aprovechando su sistema de almacenamiento clave-valor integrado, que ofrece una forma sencilla y distribuida para guardar y recuperar datos.

5.1 Modelo de Datos

El modelo de datos en NATS JetStream se organiza mediante claves únicas vinculadas a colecciones o "buckets". Cada bucket representa una categoría de datos, como usuarios o funciones. Este diseño es flexible y facilita la escalabilidad en aplicaciones distribuidas.

5.1.1.Usuarios (users)

- Bucket: users
- Claves: username
- Valor: Un objeto JSON que almacena el nombre de usuario y la contraseña hasheada.

  Ejemplo:

```
Bucket: users
Key: testuser
Value: {
    "username": "testuser",
    "password": "$2b$10$Xv3zT9huT1V.4B2O1jiLoOU3wPQUtXeRBB0A2AnZNE.4iEN/dn3qW"
}
```

5.1.2 Funciones (functions)

- Bucket: functions
- Claves: functionId
- Valor: Un objeto JSON que almacena el código y metadatos asociados a la función.

  Ejemplo:

```
Bucket: functions
Key: 1
Value: {
    "id": 1,
    "user": "testuser",
    "code": "return input + 1;"
}
```

5.2 Operaciones Soportadas

5.2.1.Usuarios:

- Registro de usuario:
  - Al registrar un usuario, se utiliza el bucket users y se crea una clave con el nombre de usuario. El valor asociado incluye el nombre de usuario y la contraseña hasheada.
- Autenticación de usuario:
  - Se busca en el bucket users la clave correspondiente al nombre de usuario. Si existe, se compara la contraseña ingresada con la almacenada utilizando bcrypt.compare.

5.2.2.Funciones:

- Registro de función:
  - Se genera un identificador único (functionId) para la clave y se almacena en el bucket functions. El valor asociado incluye el usuario y el código de la función.
- Listado de funciones:
  - Se filtran las claves en el bucket functions que pertenecen a un usuario específico.
- Ejecución de función:
  - Se busca la clave correspondiente en el bucket functions para recuperar el código de la función y ejecutarlo.
- Eliminación de función:
  - Se elimina la clave correspondiente en el bucket functions.

5.3 Ventajas de Usar NATS JetStream como Almacén

5.3.1 Distribución y Escalabilidad:

- NATS JetStream permite distribuir los datos en múltiples nodos, lo que garantiza la escalabilidad del sistema.

5.3.2 Velocidad:

- La estructura de clave-valor de NATS JetStream ofrece lecturas y escrituras rápidas, ideales para aplicaciones en tiempo real.

5.3.3 Persistencia Ligera:

- JetStream almacena los datos en buckets con persistencia configurable, asegurando que los datos sobrevivan a reinicios.

5.3.4 Simplicidad:

- La API de clave-valor de JetStream es fácil de usar y se integra directamente con el sistema de mensajería de NATS.

5.3.5 Compatibilidad con Mensajería:

- Al estar basado en NATS, el sistema puede aprovechar la infraestructura de mensajería para sincronizar y distribuir datos.

6. Tópicos de NATS

En este sistema FaaS, NATS actúa como middleware de mensajería asíncrona para coordinar las solicitudes de ejecución de funciones, la gestión de errores y el monitoreo. Los tópicos de NATS facilitan la comunicación eficiente y desacoplada entre los diferentes componentes del sistema.

6.1 Tópico Principal: functions.execute

El tópico centraliza la interacción con NATS. El API Server lo utiliza para publicar solicitudes de ejecución de funciones. Los mensajes en el tópico contienen el código de la función a ejecutar y los datos de entrada proporcionados por el usuario. Un componente suscrito recibe las solicitudes, ejecuta el código y devuelve el resultado.

6.2 Tópico de Errores: functions.error

El tópico se encarga del manejo de errores. Si ocurre un problema durante la ejecución de una función o al interactuar con NATS, el API Server publica un mensaje con detalles sobre el error. Esto facilita el monitoreo de problemas y el análisis posterior.

6.3 Tópico Opcional: functions.logs

El monitoreo del sistema recopila logs de funciones ejecutadas, incluyendo detalles sobre el usuario, la función, los datos de entrada y el resultado. Este enfoque permite generar estadísticas y facilita la depuración.

El uso de los anteriores tópicos en el sistema NATS asegura un flujo de comunicación ágil y escalable, esencial para el rendimiento del sistema FaaS en entornos distribuidos.

7. Conclusión

El proyecto ha abordado el diseño e implementación de una arquitectura FaaS (Function-as-a-Service) mediante la integración de Apache APISIX (gestión de rutas), NATS (middleware de mensajería asíncrona) y un API Server modular (lógica de negocio). El objetivo central ha sido validar la viabilidad de una solución FaaS eficiente, escalable y robusta basada en software de código abierto.

El desarrollo ha implicado la segregación de responsabilidades entre componentes para facilitar la construcción, prueba y depuración.

La integración de NATS ha permitido explorar el desacoplamiento entre invocación y ejecución de funciones, posibilitando la adición de workers paralelos para incrementar el throughput y tolerar fallos sin afectar la disponibilidad del API Server.

La implementación de Apache APISIX como gateway ha proporcionado una abstracción para la gestión de rutas, seguridad y observabilidad. Se ha configurado APISIX para gestionar el enrutamiento dinámico y la implementación de políticas de autenticación.

Finalmente, se ha empleado NATS JetStream como almacenamiento clave-valor para funciones y usuarios, evaluando el equilibrio entre rendimiento y durabilidad que ofrece esta solución.

Este trabajo confirma que se puede construir un sistema FaaS mediante la combinación de diferentes tecnologías para la creación de aplicaciones serverless, destacando el valor del desacoplamiento, la gestión centralizada de APIs y el uso de sistemas de mensajería.