DESARROLLO FAAS

**1. Arquitectura de Microservicios**

A continuación se detallan los microservicios que se emplean y sus funciones principales:

1. **auth**:
   * **Función**: Es el microservicio encargado de la autenticación.
   * **Puertos**: Expone el puerto 8000.
   * **Dependencias**: Depende de redis, lo que sugiere que puede usar Redis para almacenamiento en caché o como parte de su lógica de autenticación.
2. **faas**:
   * **Función**: Este microservicio parece estar relacionado con una arquitectura basada en funciones como servicio (FaaS, por sus siglas en inglés). Posiblemente maneje la ejecución de funciones específicas bajo demanda.
   * **Puertos**: Expone el puerto 8001.
   * **Dependencias**: Depende de redis (posiblemente para almacenamiento o caché) y de nats (probablemente para mensajería y comunicación entre microservicios).
3. **orquestador**:
   * **Función**: Este microservicio actúa como un orquestador, es decir, coordina la ejecución de otros servicios o tareas dentro del sistema.
   * **Puertos**: Expone el puerto 8002.
   * **Dependencias**: Depende de redis y nats, probablemente para comunicación eficiente entre los servicios y manejo de estado o caché.
4. **worker**:
   * **Función**: Este microservicio es un trabajador que probablemente realiza tareas en segundo plano o procesamiento de eventos/mensajes.
   * **Dependencias**: Depende de nats para la mensajería y probablemente reciba instrucciones o trabajos a través de este servicio.
5. **redis**:
   * **Función**: Este servicio utiliza Redis, una base de datos en memoria que se emplea comúnmente para almacenamiento en caché, gestión de sesiones o como broker de mensajes.
   * **Puertos**: Expone el puerto 6379, que es el puerto estándar de Redis.
   * **Configuración**: Se utiliza un archivo de configuración redis.conf.
6. **nats**:
   * **Función**: NATS es un sistema de mensajería ligero y de alto rendimiento. Este servicio se utiliza para facilitar la comunicación entre los otros microservicios.
   * **Puertos**: Expone el puerto 4222, que es el puerto por defecto para las conexiones NATS.

**Resumen de funciones:**

* **auth**: Gestión de autenticación.
* **faas**: Ejecución de funciones como servicio (FaaS).
* **orquestador**: Coordinación y orquestación de servicios.
* **worker**: Procesamiento de trabajos o tareas en segundo plano.
* **redis**: Almacenamiento en caché y posiblemente gestión de sesiones.
* **nats**: Comunicación eficiente entre microservicios mediante mensajería.

**2. Como trabajan los “workers”.**

1. **Dependencia de NATS**:
   * El servicio worker depende de **NATS**, lo que indica que la comunicación entre los **workers** y otros microservicios probablemente se realice a través de este sistema de mensajería.
   * **NATS** actúa como un broker de mensajes, permitiendo que los **workers** reciban tareas, eventos o mensajes de otros microservicios (como el **faas** o el **orquestador**) que necesitan ser procesados.
2. **Escucha de Mensajes**:
   * Los **workers** se suscriben a canales de mensajes o temas específicos en **NATS**.
   * Cuando un microservicio como **faas** o **orquestador** genera una tarea o evento que debe ser procesado, publica un mensaje en un canal específico de **NATS**.
   * Los **workers** escuchan estos canales y cuando reciben un mensaje (por ejemplo, una tarea a realizar), comienzan a procesar ese trabajo.
3. **Procesamiento de Tareas**:
   * Una vez que un **worker** recibe un mensaje con una tarea o trabajo a realizar, comienza a ejecutarlo. Este trabajo puede ser cualquier operación que esté definida para ser ejecutada en segundo plano, como el procesamiento de datos, la ejecución de cálculos o el manejo de eventos asíncronos.
   * Dependiendo de la lógica implementada, el **worker** puede interactuar con otros servicios (por ejemplo, usando **redis** para almacenar resultados temporales o acceder a información almacenada en caché).
4. **Finalización de la Tarea**:
   * Después de completar el procesamiento, el **worker** podría enviar un mensaje de confirmación o resultado a otro microservicio (por ejemplo, al **faas** o al **orquestador**), informando que la tarea ha sido completada o que el resultado está listo.
   * Es posible que los **workers** también se encarguen de manejar tareas recurrentes o se mantengan esperando nuevas instrucciones en **NATS**.

**Descripción del flujo de trabajo de los workers:**

1. **Conexión a NATS**:
   * El **orquestador** se conecta a **NATS**, un sistema de mensajería, mediante la librería nats.go.
   * Se suscribe a un canal llamado "Peticion" para recibir solicitudes o mensajes que indican la necesidad de crear y ejecutar un **worker**.
2. **Recibo de un mensaje**:
   * Cuando el **orquestador** recibe un mensaje en el canal "Peticion", este mensaje contiene datos en formato JSON. La estructura de ese mensaje está definida en la estructura Message, que tiene dos campos: Funcion y Parametro.
   * El orquestador deserializa el mensaje recibido y obtiene la información necesaria para ejecutar una tarea (función y parámetros).
3. **Creación de un Worker**:
   * El **orquestador** invoca la función createWorker con los parámetros extraídos del mensaje. En este caso, la función createWorker recibe un nombre de imagen Docker (imageName) y un comando a ejecutar dentro del contenedor Docker (command).
   * **El contenedor Docker es creado** con la imagen proporcionada. Si la imagen no está disponible localmente, el orquestador descarga la imagen desde un repositorio Docker (como Docker Hub).
   * El contenedor se inicia y ejecuta el comando especificado. El contenedor es aislado, y se le asigna un nombre único basado en la hora actual (containerName), lo que asegura que cada contenedor creado sea único.
4. **Ejecución de la tarea (Worker)**:
   * El contenedor ejecuta el comando proporcionado (en el ejemplo de código, el comando es simplemente echo "Hola desde Docker con Go!").
   * El **orquestador** espera a que el contenedor termine de ejecutarse. Esto se logra usando cli.ContainerWait, que monitorea el estado del contenedor y espera a que termine su ejecución.
5. **Obtención de los logs del contenedor**:
   * Una vez que el contenedor ha terminado de ejecutarse, el **orquestador** captura los logs de la ejecución del contenedor usando cli.ContainerLogs. Esto permite obtener la salida estándar (stdout) y los errores (stderr) del contenedor que se está ejecutando.
   * Los logs se leen y se almacenan en un buffer (stdoutBuf), para luego enviarlos como respuesta al canal de **NATS**.
6. **Respuesta al canal de NATS**:
   * El **orquestador** envía los logs de salida del contenedor de vuelta al canal de NATS en respuesta al mensaje original. Esto se hace mediante el método m.Respond([]byte(stdout)), donde stdout es el resultado del trabajo realizado por el worker.
7. **Limpieza del contenedor**:
   * Después de la ejecución del contenedor y la captura de los logs, el **orquestador** limpia el contenedor utilizando cli.ContainerRemove. Esto asegura que los contenedores se eliminen después de su uso, evitando así el consumo innecesario de recursos.

**Resumen de cómo trabajan los workers:**

* El **orquestador** actúa como un intermediario que recibe solicitudes (mensajes) a través de **NATS** para crear **workers**.
* Cada **worker** es un contenedor Docker que se crea y se ejecuta con una imagen específica y un comando definido.
* Una vez que el contenedor termina de ejecutarse, el **orquestador** captura los logs de la ejecución y los envía de vuelta como respuesta al canal de **NATS**.
* Finalmente, el contenedor se elimina para liberar recursos.

**Relación con los workers:**

* Los **workers** en este caso son contenedores Docker que se crean dinámicamente para ejecutar trabajos específicos definidos por los mensajes recibidos en **NATS**.
* Cada **worker** se ejecuta de manera aislada, lo que le permite realizar tareas en paralelo o de manera independiente según las necesidades del sistema.
* El proceso de creación y ejecución de los **workers** es gestionado completamente por el **orquestador**, quien se encarga de gestionar la infraestructura de contenedores y de coordinar la comunicación mediante **NATS**.

**3. Cómo se escala a los workers?**

Actualmente la escalabilidad de los workers no esta definia explicitamente.

**4 Cómo se escala la base de datos?**

Actualmente la base de datos no escala, pues no está implementada ninguna estrategia para de escalabilidad de las bases de datos.