

Máster Universitario en Computación en la Nube y
de Altas Prestaciones

Cloud Computing

Construcción de un servicio complejo basado en microservicios

Blanca Mellado Pinto
Ismael Mira Hernández

Índice

1. Introducción	3
2. Microservicios implementados	3
2.1. Frontend	3
2.2. Worker	4
2.3. Observer: estrategia de elasticidad	5
3. Imágenes y contenedores Docker	5
3.1. Kafka y Zookeeper	5
3.2. PostgreSQL	6
4. Keycloak	¡Error! Marcador no definido.
5. Desarrollo del trabajo	9
6. Resultados	11
7. Conclusiones	12

1. Introducción

En este trabajo se va a implementar un servicio en la nube basándonos en una arquitectura de microservicios. El programa se ha desarrollado con Node.js, por lo que el lenguaje de programación utilizado es JavaScript.

La funcionalidad del servicio se basa en mandar trabajos al mismo, y tras este realizar la tarea indicada, mostrar un resultado. Como ejemplo, mandar un trabajo significa enviar al servicio una URL con un repositorio de Github. El servicio clonará y ejecutará el código del repositorio, y almacenará el resultado del trabajo en una base de datos.

Para lanzar todos los componentes se usan contenedores Docker. De esta manera los diferentes microservicios se pueden comunicar entre ellos, pero son independientes de la configuración del host y aislados de este. Además, utilizamos docker-compose para organizar la arquitectura del servicio y manejar el despliegue del servicio completo.

Finalmente, para asegurar la seguridad del sistema, se usa Keycloak, gestionando el acceso al servicio.

2. Microservicios implementados

2.1. Frontend

En primer lugar, el servicio cuenta con un frontend desarrollado con Express. Este microservicio permite acceder la dirección localhost:80, que está mapeada dentro del contenedor a la 3000 (la default para express), donde a través de parámetros se pueden realizar diferentes peticiones. A cada petición se le asigna un código único (UUID). Para la asignación de códigos hemos utilizado la biblioteca *uuidv4*.

Además, se usa *keycloak* como mecanismo de autorización para decidir si un usuario tiene permiso para enviar un trabajo. Este servicio se comentará en profundidad después.

El frontend utiliza Kafka, que es una plataforma para la interconexión de servicios mediante colas, la cual un esquema de productores y consumidores. Para ello, se configura el servicio con los parámetros que hemos elegido. A continuación, creamos una cola de Kafka para trabajos si esta no existe previamente e inicializamos un productor, que será el encargado de añadir trabajos a la cola. Hemos decidido que cada servicio que utilice una cola la intente crear al principio para asegurarnos de que esta existe.

Por otra parte, el frontend se conecta a una base de datos PostgreSQL, que es un sistema de bases de datos de código abierto. En esta base de datos se escribirá el resultado de los trabajos y podremos mostrarla al cliente del servicio cuando la sondee. Por tanto, el servicio es asíncrono, y es el cliente el que sondea si el resultado ya ha aparecido.

A continuación, se describen los métodos del frontend:

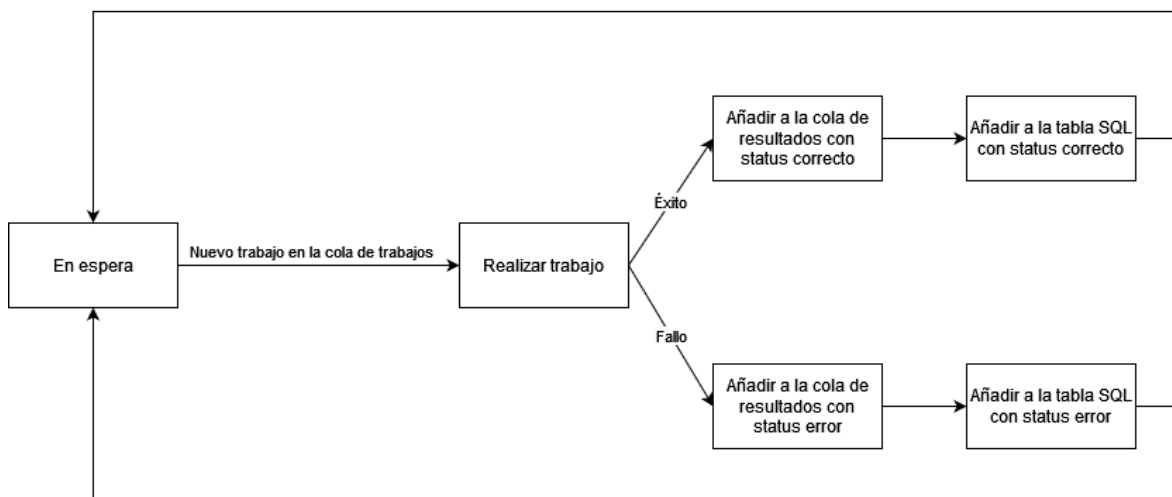
- **ADD:** este método recibe como parámetro una URL y añade el trabajo a la cola de Kafka mediante el productor.
- **ALL:** este método busca en la base de datos todos los trabajos del usuario y le presenta una lista con los mismos.
- **STATUS:** este método recibe como parámetro una clave de un trabajo sirve para obtener el estado de este trabajo añadido anteriormente, recogiendo el mismo de la base de datos.
- **HEALTHCHECK:** este método sirve para comprobar que el frontend está activo y todo funciona correctamente. Este método lo utiliza docker-compose para confirmar que el contenedor está funcionando mediante un healthcheck que hemos definido.

2.2. Worker

El worker es el microservicio encargado de realizar los trabajos presentes en la cola de trabajos de Kafka. Por lo tanto, tras conectarse a la cola y crear un consumidor, queda a la espera (asíncrono) de que haya un trabajo nuevo en la cola.

Puede haber varios workers recibiendo y ejecutando trabajos, ya que, mediante el mecanismo de particiones de Kafka, si todos ellos están asignados al mismo grupo de consumidores, los mensajes se repartirán entre ellos.

Cuando recibe una petición, obtiene de ella el timestamp del mensaje, la clave (única para cada trabajo) y el valor, que es la URL del trabajo, y ejecuta el trabajo, tomando nota del tiempo que tarda la ejecución. Los valores de tiempo serán importantes cuando veamos la estrategia de elasticidad. Tras la finalización del mismo, se guarda el resultado (correcto: “Finished” o error: “Failed”) en otra cola Kafka, en este caso de resultados, y además escribe el resultado en la base de datos, haciendo una inserción en la tabla de resultados. Este esquema muestra el proceso:



2.3. Observer: estrategia de elasticidad

Para la estrategia de elasticidad decidimos basarnos en el tiempo que esperan los trabajos a ser ejecutados, información que podemos obtener observando la cola de resultados.

Para ello, el worker introduce dos parámetros adicionales en la cola de resultados cuando termina un trabajo: el tiempo en el cual se introdujo el trabajo realizado en la cola de trabajos, el cual el worker conoce mediante el timestamp del mensaje, y el tiempo que ha tardado en ejecutar el trabajo.

El observador observa el tiempo de llegada del mensaje a resultados y le resta ambos valores, obteniendo el tiempo de espera. A continuación, inicializa un consumidor y se conecta a la cola de resultados de Kafka, quedando a la espera. Cuando un trabajo finalizado (sea correcto o fallido) se añade a esa cola, el observer muestra la información del mismo.

Si el tiempo que un trabajo ha estado esperando en la cola ha sido muy alto, podrían añadirse workers al sistema. Igualmente, si hay varios workers y los trabajos se realizan instantáneamente puede reducirse el número de workers.

3. Imágenes y contenedores Docker

Se utiliza Compose, que sirve para definir y lanzar una aplicación con varios contenedores Docker. Para ello se crea el fichero docker-compose.yml, donde se definen los parámetros de los contenedores.

Todos los componentes están dentro de una misma red “default”, y se conocen entre ellos por el nombre del microservicio como nombre de host.

Por ello sólo exponemos al exterior los puertos del frontend y de keycloak (para poder realizar la configuración desde un navegador)

Hemos añadido la opción de relanzar a menos que se haya parado a todos los contenedores para que si se produce un fallo y finaliza la ejecución del contenedor, este se vuelva a lanzar.

3.1. Kafka y Zookeeper

En primer lugar, se añade el servicio Zookeeper, que es necesario para que Kafka funcione, ya que permite coordinar los brokers de Kafka. Este expone el puerto 2181. A continuación, se añade Kafka. Con la cláusula depends podemos hacer que espere a que Zookeeper esté lanzado, ya que, si no puede haber error, y de esta forma realizamos un despliegue ordenado.

También definimos las variables de entorno que se utilizan dentro de la imagen de Kafka para la conexión a Zookeeper, según se indica en la documentación de la imagen.

Hemos definido un healthcheck que nos permite saber si Kafka está preparado para la conexión de productores y consumidores. Utilizamos un comando que lista los topics disponibles, ya que, si Kafka no está preparada, este comando fallará.

Para depurar errores en Kafka entramos en el contenedor y utilizamos los scripts de Kafka para observar la correcta creación de los topics y mensajes.

```
docker exec -it docker-compose_kafka_1 /bin/bash
kafka-topics.sh --list --bootstrap-server localhost:9092
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic trabajos --from-
beginning
```

```
zookeeper:
  image: ubuntu/zookeeper
  expose:
    - "2181"
  restart: "unless-stopped"

kafka:
  image: ubuntu/kafka
  expose:
    - "9092"
  depends_on:
    - zookeeper
  environment:
    ZOOKEEPER_HOST: zookeeper
    ZOOKEEPER_PORT: 2181
  healthcheck:
    test: ["CMD", "kafka-topics.sh", "--list", "--bootstrap-server", "localhost:9092"]
    interval: 5s
    timeout: 10s
    retries: 5
    restart: "unless-stopped"
```

3.2. PostgreSQL

A continuación, se añade la base de datos PostgreSQL, indicando mediante variables de entorno la base de datos, el usuario y la contraseña.

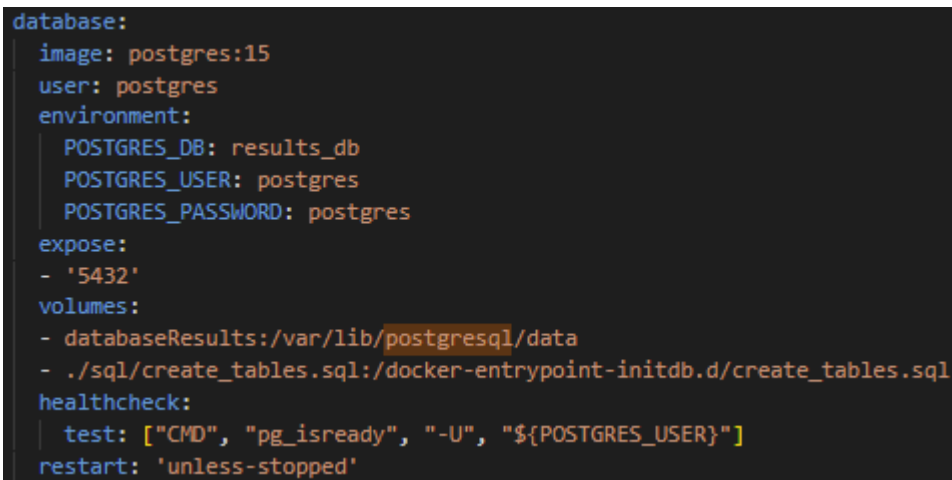
La base de datos hace uso de un volumen para almacenar de forma persistente los datos y poder recuperarlos si se relanza el sistema. Para ello hemos definido un volumen databaseResults y lo hemos mapeado dentro del contenedor con la ruta indicada, en la que se crean las bases de datos dentro del contenedor.

También se utiliza un script SQL para crear la tabla de resultados si la base de datos se inicia por primera vez. La tabla de resultados contiene los campos usuario, key y status. Para ello, en volúmenes indicamos que se introduzca el script create_tables.sql en la

carpeta docker-entrypoint-initdb.d. Postgres ejecuta al iniciar por primera vez el volumen los scripts introducidos en esta carpeta.

Para comprobar la creación de las tablas hemos entrado en el contenedor y ejecutado sentencias SQL:

```
docker exec -it docker-compose_database_1 /bin/bash
psql -U postgres
\c results_db //(\c <nombre_base_datos>)
\d <tabla>
select * from resultados;
exit //sale de psql
exit //sale del contenedor
```



```
database:
  image: postgres:15
  user: postgres
  environment:
    POSTGRES_DB: results_db
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  expose:
    - '5432'
  volumes:
    - databaseResults:/var/lib/postgresql/data
    - ./sql/create_tables.sql:/docker-entrypoint-initdb.d/create_tables.sql
  healthcheck:
    test: ["CMD", "pg_isready", "-U", "${POSTGRES_USER}"]
    restart: 'unless-stopped'
```

Por último, se añaden los contenedores con los microservicios que hemos creado (frontend, worker y observer). Cada uno de ellos se crea mediante un dockerfile, donde a partir de la imagen base (en este caso node), se copian los archivos source y package.json y se ejecuta el “npm install”. Este dockerfile es el que se le pasa en build en la plantilla del YAML.

Para el frontend, mapeamos el puerto 80 al 3000 del contenedor, para poder acceder al mismo por cualquier navegador accediendo a localhost:80. También se indican las variables de configuración de Kafka y de la base de datos para que puedan ser utilizadas en el código, ya que en el código no hemos introducido ningún parámetro de configuración de manera explícita, esta se obtiene mediante la variables de entorno.

Tanto para el frontend como para el worker indicamos que espere a que Kafka esté funcionando, y pase el healthcheck para ser lanzados. El frontend también tiene un healthcheck ya que es importante saber si los clientes pueden acceder al servicio.

Para el worker no es necesario exponer puertos, igual que para el observer. Ninguno de ellos va a ser accedido por otro servicio

En el caso del observer, con la cláusula depends le decimos que espere a que el frontend y el worker estén lanzados, de esta manera nos aseguramos de que la cola de resultados de Kafka está lanzada.

```
frontend:
  build:
    context: .
    dockerfile: dockerfile_frontend
  depends_on:
    kafka:
      condition: service_healthy
  ports:
    - "80:3000"
  environment:
    KAFKA_BOOTSTRAP: "kafka:9092"
    POSTGRES_HOST: database
    POSTGRES_PORT: 5432
    POSTGRES_DB: results_db
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  healthcheck:
    test: ["CMD", "curl", "localhost:3000/healthcheck"]
  restart: "unless-stopped"

worker:
  build:
    context: .
    dockerfile: dockerfile_worker
  depends_on:
    kafka:
      condition: service_healthy
  environment:
    KAFKA_BOOTSTRAP: "kafka:9092"
    POSTGRES_HOST: database
    POSTGRES_PORT: 5432
    POSTGRES_DB: results_db
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  restart: "unless-stopped"

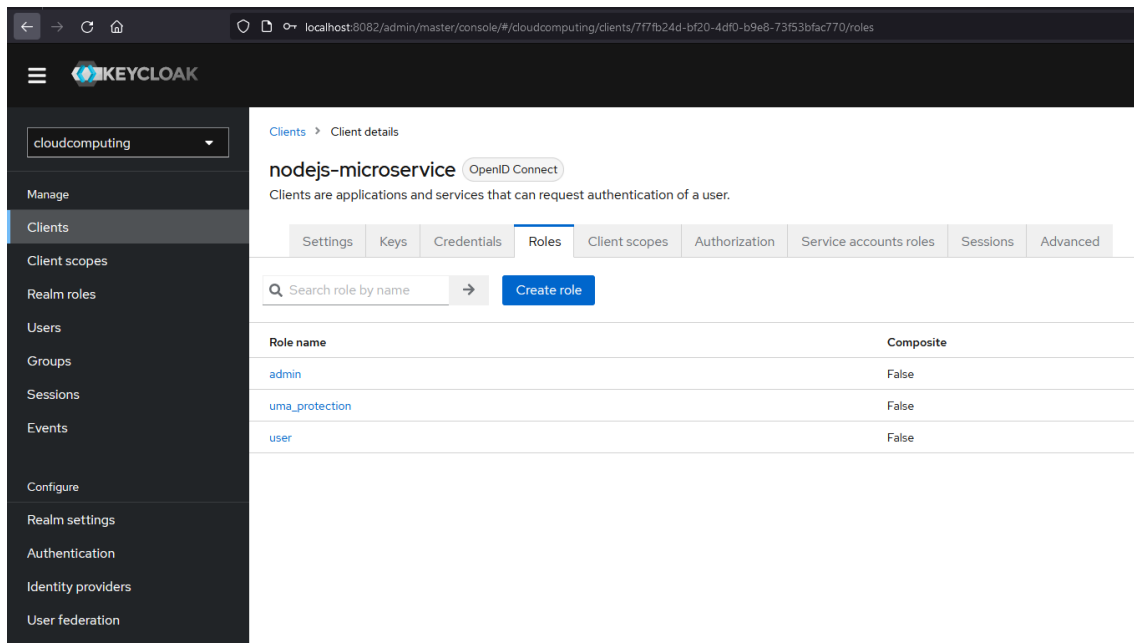
observer:
  build:
    context: .
    dockerfile: dockerfile_observer
  depends_on:
    - frontend
    - worker
  environment:
    KAFKA_BOOTSTRAP: "kafka:9092"
  restart: "unless-stopped"
```


4. Keycloak

Se utiliza para la autenticación del frontend. Existe otro YAML en la carpeta docker-compose-keycloak que lanza otra base de datos PostgreSQL (no confundir con la anterior), necesaria para el servicio, y lanza Keycloak.

Podemos acceder a la configuración del mismo mapeando el puerto 8080 del contenedor, en este caso lo mapeamos al 8082.

A continuación, podemos configurar un Realm, con los clientes y roles necesarios.



Para ello creamos un Realm (llamado cloudcomputing) y un cliente, que será nuestra aplicación en Node. Por eso lo hemos llamado nodejs-microservice. A continuación, añadimos al cliente los usuarios user y admin, con los permisos adecuados para cada uno.

5. Desarrollo del trabajo

Para la realización de este trabajo, debido al desconocimiento de las tecnologías a utilizar como Node.js o Docker, en primer lugar, fue necesario descargar algunos ejemplos de la web y leer documentación para conocer los primeros pasos a seguir.

Por ejemplo, nos basamos en ejemplos en Node.js que, utilizando Kafka, creaba un sistema simple de consumidor-productor (<https://www.sohamkamani.com/nodejs/working-with-kafka/>). Posteriormente, probamos a pasar este ejemplo a Docker, siguiendo la guía presente en <https://towardsdatascience.com/how-to-install-apache-kafka-using-docker-the-easy-way-4ceb00817d8b>.

Seguidamente, como tampoco teníamos experiencia usando sistemas de repositorios tipo GitHub, creamos algunos de ejemplo y aprendimos a utilizar la herramienta Git para hacer clone, commit, push, pull...

Llegado este momento, comenzamos a desarrollar el frontend. Contábamos con algo de experiencia creando APIs REST en Python, por lo que el funcionamiento de Express fue fácil de aprender. Creamos una API simple que recibía un parámetro y lo imprimía por pantalla. Teniendo eso, introducimos Kafka y creamos un productor. Tuvimos algo de problemas con los “topic”, ya que había veces que no mandaba los trabajos bien al no existir, por lo que introducimos una comprobación para crearlo si no existía. Después, creamos el método que leía los parámetros en la URL y lo metía en la cola de trabajos.

El paso siguiente fue añadir una base de datos con PostgreSQL. Basándonos en el ejemplo presente en <https://ed.team/blog/como-usar-bases-de-datos-postgres-con-nodejs>, añadimos los métodos para acceder a la base de datos y leer.

A continuación, empezamos a desarrollar el worker. El comienzo fue similar, conectándonos a Kafka y creando un rol de consumidor en vez de productor. Buscamos una forma de hacer clone desde el código en Node y al recoger un trabajo de la cola hacíamos que clonara el repositorio. Investigamos la elasticidad de las colas de Kafka y encontramos el mecanismo de los grupos de consumidores.

Implementamos después la comunicación de los resultados mediante la base de datos, cuya mayor dificultad fue su configuración y algunos problemas con el usuario.

A continuación elaboramos una estrategia de elasticidad. Respecto al código del observer, sabiendo ya cómo funcionaban el frontend y el worker fue sencillo crearlo, ya que solamente añadimos una función asíncrona para leer de la cola de resultados.

Mientras implementamos los componentes, realizamos también el paso de estos a Docker y docker-compose. Para Kafka y PostgreSQL nos basamos en ejemplos de la web sobre cómo lanzar contenedores con estos componentes, simplemente cambiando los parámetros a los que queríamos. Para nuestro código (frontend, worker y observer) tuvimos algunos problemas porque no entendíamos cómo incluir cada dockerfile en el docker-composer. Otro problema que nos surgió fue con el mapeo de puertos, ya que de primeras no podíamos conectarnos localmente, pero vimos algunos ejemplos y solucionamos nuestro fallo.

Para crear la tabla en nuestra base de datos decidimos crear un script, que en caso de no existir esta añade los atributos necesarios.

Seguidamente, empezamos con la configuración de Keycloak. Para ello, creamos una carpeta aparte con otro docker-composer en el que se lanzan los dos contenedores. Accediendo a la configuración desde la interfaz web, creamos el Realm que va a utilizar el frontend y los roles. Posteriormente, cuando Keycloak estaba configurado, añadimos al frontend el código necesario y en cada método le asignamos el rol necesario. Todo esto se ha hecho siguiendo los pasos en <https://medium.com/devops-dudes/securing-node-js-express-rest-apis-with-keycloak-a4946083be51>.

Para el despliegue hemos utilizado los siguientes comandos:

docker-compose build

docker-compose start

docker-compose up //build y start

docker-compose down //stop y eliminar

docker-compose down -v //eliminar tambien volumenes

docker-compose up -d - --build //lanzar en segundo plano, reconstruir las imagenes

docker-compose logs <componente> //logs de cada componente

docker-compose -d - --scale worker=<n> //escalar a n workers

6. Resultados

A continuación, mostramos ejemplos de ejecución del programa y logs generados:

- Añadir trabajo

```
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$ curl localhost:80/add?url=http://github.com/IsmaelMira/trabajoi
Trabajo añadido: c0a63b31-76a0-4888-a58a-dbe803fb2e9c
```

- Consultar trabajos

```
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$ curl localhost:80/all
[{"usuario":"user","key":"fcd5e92-1b80-4fe1-acfb-01572ee3fcc","status":"Finished"}, {"usuario":"user","key":"d92ee499-81c8-4789-8813-97054c481c27","status":"Finished"}, {"usuario":"user","key":"8422d187-2f3a-4aaa-ab2a-6d3bdfcbdfa9","status":"Finished"}, {"usuario":"user","key":"58007544-a07a-477a-8f5a-a2aa426ddcfe","status":"Finished"}, {"usuario":"user","key":"3825a808-aedb-4dbc-b778-a60b32c59c75","status":"Finished"}, {"usuario":"user","key":"0b5fd362-f538-4be9-b265-d924deb3aa50","status":"Finished"}, {"usuario":"user","key":"6e4bd9f3-9b9f-4985-82a4-f72ecd7ed0aa","status":"Finished"}, {"usuario":"user","key":"2017002a-4cf2-405d-9360-6898babaf242","status":"Finished"}, {"usuario":"user","key":"1d9f5c21-4745-4b26-bcf9-874784b62fac","status":"Finished"}, {"usuario":"user","key":"681895cc-c59a-446b-bd1a-d46c02bb2bf3","status":"Finished"}, {"usuario":"user","key":"8438a35d-e755-4c12-ba7c-b931bb5a625f","status":"Finished"}, {"usuario":"user","key":"080122e0-5c99-4469-a4b3-89e754f4f5a4","status":"Finished"}, {"usuario":"user","key":"4b13717f-72d4-47e3-830c-8d8d43d55b8d","status":"Finished"}, {"usuario":"user","key":"fe283c54-2dea-478a-b025-32aefcfc51","status":"Finished"}, {"usuario":"user","key":"c0a63b31-76a0-4888-a58a-dbe803fb2e9c","status":"Finished"}]admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$
```

- Consultar estado de trabajo

```
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$ curl localhost:80/status?key=c0a63b31-76a0-4888-a58a-dbe803fb2e9c
[{"usuario":"user","key":"c0a63b31-76a0-4888-a58a-dbe803fb2e9c","status":"Finished"}]admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$
```

- Logs frontend

```
Attaching to docker-compose-frontend_1
frontend_1 | {"level":"WARN","timestamp":"2023-02-05T22:15:22.201Z","logger":"kafkajs","message":"KafkaJS v2.0.0 switched default partitioner. To retain the same partitioning behavior as in previous versions, create the producer with the option \createPartitioner: Partitioners.LegacyPartitioner\". See the migration guide at https://kafka.js.org/docs/migration-guide-v2.0.0#producer-new-default-partitioner for details. Silence this warning by setting the environment variable \"KAFKAJS_NO_PARTITIONER_WARNING=1\""}
frontend_1 | Frontend listening on port 3000
frontend_1 | {"level":"ERROR","timestamp":"2023-02-05T22:15:24.421Z","logger":"kafkajs","message":"[Connection] Response CreateTopics(key: 19, version: 3)","broker": "6db27839ecd3:9092","clientId":"frontend","error":"Topic creation errors","correlationId":0,"size":58}
frontend_1 | {
frontend_1 |   {
frontend_1 |     usuario: 'user',
frontend_1 |     key: 'fcd5e92-1b80-4fe1-acfb-01572ee3fcc',
frontend_1 |     status: 'Finished'
frontend_1 |   },
frontend_1 |   {
frontend_1 |     usuario: 'user',
frontend_1 |     key: 'd92ee499-81c8-4789-8813-97054c481c27',
frontend_1 |     status: 'Finished'
frontend_1 |   },
frontend_1 |   {
frontend_1 |     usuario: 'user',
frontend_1 |     key: '8422d187-2f3a-4aaa-ab2a-6d3bdfcbdfa9',
frontend_1 |     status: 'Finished'
frontend_1 |   }
frontend_1 | },
```

- Logs worker

```
worker_1 | {"level":"INFO","timestamp":"2023-02-05T22:15:30.872Z","logger":"kafkajs","message":"[ConsumerGroup] Consumer has joined the group","groupId":"worker","memberId":"workers-d7a19c20-ecd0-46ab-88d8-0d822e0d01e4","leaderId":"workers-d7a19c20-ecd0-46ab-88d8-0d822e0d01e4","isLeader":true,"memberAssignment":{"trabajos":[0]}","groupProtocol":"RoundRobinAssigner","duration":5544}
worker_1 | received message: c0a63b31-76a0-4888-a58a-dbe803fb2e9c http://github.com/IsmaelMira/trabajoi
worker_1 | timeStart: Sun Feb 05 2023 22:16:04 GMT+0000 (Coordinated Universal Time)
worker_1 | Trabajo c0a63b31-76a0-4888-a58a-dbe803fb2e9c terminado
worker_1 | Escrito mensaje de estado
worker_1 | []
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$
```

- Logs observer

```
observer_1 | received message: {"code":"Finished","timeJobPosted":167563563993,"timeExecution":10} timeResultArrived: 167563564131 wait was: 128
observer_1 | received message: {"code":"Finished","timeJobPosted":1675635631978,"timeExecution":0} timeResultArrived: 1675635631990 wait was: 12
observer_1 | received message: {"code":"Finished","timeJobPosted":1675635633158,"timeExecution":0} timeResultArrived: 1675635633170 wait was: 12
observer_1 | received message: {"code":"Finished","timeJobPosted":1675635634038,"timeExecution":1} timeResultArrived: 1675635634055 wait was: 16
observer_1 | received message: {"code":"Finished","timeJobPosted":1675635635107,"timeExecution":0} timeResultArrived: 1675635635121 wait was: 14
observer_1 | received message: {"code":"Finished","timeJobPosted":1675635635899,"timeExecution":0} timeResultArrived: 1675635635911 wait was: 12
```

- Logs PostgreSQL(worker y frontend)

```
Attaching to docker-compose_database_1
database_1 | PostgreSQL Database directory appears to contain a database; Skipping initialization
database_1 |
database_1 | 2023-02-05 22:14:50.467 UTC [1] LOG: starting PostgreSQL 15.1 (Debian 15.1-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1
database_1 | 2023-02-05 22:14:50.481 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
database_1 | 2023-02-05 22:14:50.481 UTC [1] LOG: listening on IPv6 address ":::", port 5432
database_1 | 2023-02-05 22:14:50.485 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
database_1 | 2023-02-05 22:14:50.522 UTC [16] LOG: database system was shut down at 2023-02-05 21:15:30 UTC
database_1 | 2023-02-05 22:14:50.591 UTC [1] LOG: database system is ready to accept connections
database_1 | 2023-02-05 22:10:50.610 UTC [14] LOG: checkpoint starting: time
database_1 | 2023-02-05 22:10:50.834 UTC [14] LOG: checkpoint complete: wrote 5 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.204 s, sync=0.004 s, total=0.216 s; sync files=4, longest=0.002 s, average=0.001 s; distance=2 kB, estimate=2 kB
```

- Docker-compose ps

```
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose$ docker-compose ps

```

Name	Command	State	Ports
docker-compose_database_1	docker-entrypoint.sh postgres	Up (healthy)	5432/tcp
docker-compose_frontend_1	docker-entrypoint.sh node ...	Up (healthy)	0.0.0.0:80->3000/tcp, :::80->3000/tcp
docker-compose_kafka_1	entrypoint.sh /etc/kafka/s ...	Up (healthy)	9092/tcp
docker-compose_observer_1	docker-entrypoint.sh node ...	Up	
docker-compose_worker_1	docker-entrypoint.sh node ...	Up	
docker-compose_zookeeper_1	/opt/kafka/bin/zookeeper-s ...	Up	2181/tcp

```
admin@cc-bmelpin-k:~/ProyectoCC/CC/docker-compose-keycloak$ docker-compose ps

```

Name	Command	State	Ports
docker-compose-keycloak_database_1	docker-entrypoint.sh postgres	Up (healthy)	5432/tcp
docker-compose-keycloak_keycloak_1	/opt/bitnami/scripts/keycl ...	Up	0.0.0.0:8082->8080/tcp, :::8082->8080/tcp

7. Conclusiones

Como conclusión, nos gustaría señalar que este trabajo ha supuesto un gran reto para nosotros, ya que la mayoría de las tecnologías utilizadas eran nuevas para nosotros.

Decidimos apostar por el uso del lenguaje Node ya que lo consideramos más apropiado en este contexto, teniendo en cuenta nuestra falta de experiencia. Usar otros lenguajes como Golang quizá hubieran hecho el proceso más complicado. Consideramos que la decisión fue acertada, pues a pesar del aprendizaje necesario, es un lenguaje que se adapta muy bien al caso de uso, y a la programación en la nube utilizando microservicios.

En el caso del resto de tecnologías, consideramos que hemos adquirido experiencia en el uso de docker y docker-compose, y hemos podido observar la utilidad del uso de contenedores en el proceso de desarrollo, despliegue y mantenimiento de un servicio web. Nos gustaría en el futuro investigar más el uso de distintas redes para cada comunicación, ya que uno de nuestros objetivos fue exponer al exterior sólo los puertos estrictamente necesarios, y creemos que el uso de redes internas separadas también sería un mecanismo de seguridad adicional.

También hemos investigado en cierta profundidad la herramienta Kafka, pudiendo observar las ventajas y desventajas del uso de las colas como paso de mensajes. En el caso de las desventajas, observamos que filtrar una cola para encontrar un mensaje no era

sencillo y, debido a que Kafka trabaja mediante logs sobre los que mantiene el offset, menos apropiado que el uso de una base de datos.

Por ello decidimos utilizar Postgres para almacenar y sondear los resultados de las operaciones en base a una clave, lo que consideramos más eficiente y nos permitía que los resultados fuesen persistentes. Sería interesante explorar el eliminar estos mensajes cada cierto tiempo, para que la base de datos no crezca demasiado.

La herramienta que nos supuso más problemas Keycloak, ya que la configuración es más complicada que en el resto de herramientas, y la integración del servicio con código de Node.js no funcionaba de manera adecuada debido a la configuración precisa de roles que hay que realizar en el Realm.

Por último, nos gustaría haber podido desarrollar más el apartado de despliegue sobre la PaaS Kumori, ya que tampoco contamos con conocimientos sobre herramientas como Kubernetes.