

INTRODUCCIÓN

Este proyecto es en parte la continuación o la extensión de la POC anterior, donde creábamos un Config-Server al que se conectaban todos los microservicios para obtener sus respectivas configuraciones. Además, también configuramos un Spring Cloud Bus, que nos permitía actualizar atributos concretos de los microservicios sin la necesidad de reiniciarlos. En este caso, la diferencia será el Message Broker que usaremos, en la anterior usábamos Rabbit MQ y en esta veremos como hacerlo con Kafka.

Por lo tanto para la configuración previa del proyecto (repositorio, microservicios, Config-Server y Spring Cloud Bus) podemos utilizar el documento anterior para evitar repetirlo en este. Ya que la base será la misma, simplemente cambiaremos algunas dependencias que mas adelante explicare.

El documento se llama “ConfigServer – SpringCloudBus.pdf” y deberemos leerlo hasta el apartado ***Spring Cloud Bus & Rabbit-MQ*** (pág. 11).

Lo que si que veremos en este documento es como hacer la configuración previa de Kafka para poder usarlo como Message Broker. Y seguidamente, la configuración necesaria en los microservicios para que se conecten a este.

Dicho esto vamos a entrar mas en detalle.

CONFIGURACIÓN KAFKA

Para este paso, hay mas de una manera en que se puede instalar Kafka para usarlo con nuestros microservicios. En un inicio trate de descargarlo directamente de la página oficial de Kafka: <https://kafka.apache.org/downloads>

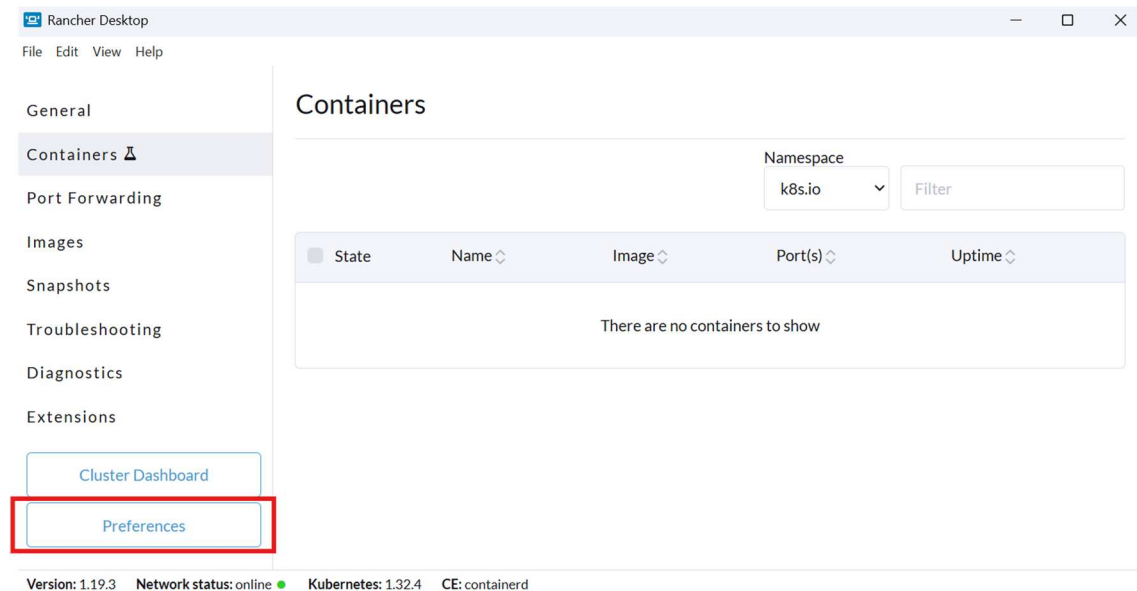
Pero lo cierto es que tuve muchas complicaciones para instalarlo y al final no lo conseguí. Ya que si no recuerdo mal, algunos de los comandos que requería utilizar estaban bloqueados por el ordenador de la empresa.

De todas manera tenemos una alternativa, que es utilizar una imagen de Kafka que es lo que termine utilizando para esta POC. La imagen me lo proporciono mi equipo y estará al final del documento por si se necesita.

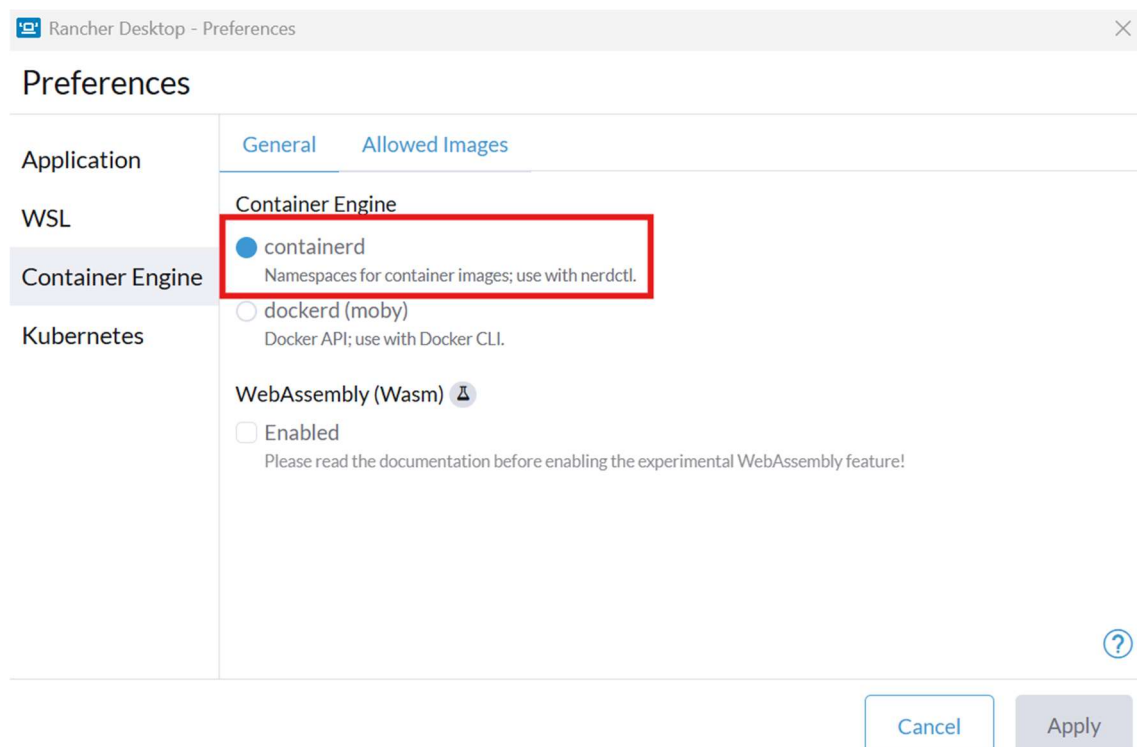
Primero de todo, es importante destacar que no deberíamos usar Docker para levantar Kafka, ya que este puede suponernos un coste monetario no deseado. Para ello existen otras alternativas totalmente gratuitas, al menos para el tamaño de este proyecto.

La alternativa que usaremos se llama **Rancher Desktop**, una plataforma de Kubernetes que nos permitirá desplegar con facilidad diferentes imágenes. Deberemos instalarlo en nuestro ordenador.

Una vez lo tengamos instalado y abierto, deberemos dirigirnos al botón de “Preferences”



Y una vez aquí, deberemos dirigirnos al apartado de Container Engine y asegurarnos de que tenemos marcada la opción de “Containerd”



De esta manera no usaremos Docker, si no que usaremos la alternativa gratuita **nerdctl**.

Una vez tengamos esto, con el Rancher Desktop iniciado y nuestro archivo docker-compose localizado (en mi caso esta nombrado como rancher-compose.yml). Desde una terminal de comandos ejecutaremos lo siguiente:

```
C:\Users\rmaillob\Documents\Kafka>nerdctl compose -f rancher-compose.yml up
```

Esto iniciara las diferentes imágenes. Le podemos dar a Ctrl + C para parar el comando, y seguidamente escribir el siguiente comando para comprobar que los contenedores están activos: **nerdctl ps**

```
C:\Users\rmaillob\Documents\Kafka>nerdctl ps
CONTAINER ID   IMAGE                                     COMMAND                  NAMES                  CREATED
STATUS        PORTS
9fbb95d81d6f   docker.io/confluentinc/cp-kafka-rest:7.3.1  "/etc/confluent/dock...  rest-proxy-local      56 seconds ago
Up            0.0.0.0:8082->8082/tcp
ede1c4ce213e   docker.io/confluentinc/cp-enterprise-control-center:7.3.1  "/etc/confluent/dock...  control-center        56 seconds ago
Up            0.0.0.0:9021->9021/tcp
a46b2a69f042   docker.io/confluentinc/cp-ksqldb-server:7.3.1  "/etc/confluent/dock...  ksqldb-server         56 seconds ago
Up            0.0.0.0:8088->8088/tcp
200db84bf762   docker.io/cnfldeos/cp-server-connect-datagen:0.5.3-7.1.0  "/etc/confluent/dock...  connect               57 seconds ago
Up            0.0.0.0:8083->8083/tcp
0685374a0a99   docker.io/confluentinc/cp-server:7.3.1        "/etc/confluent/dock...  broker                57 seconds ago
Up            0.0.0.0:29092->29092/tcp, 0.0.0.0:9092->9092/tcp, 0.0.0.0:9101->9101/tcp
9ebd451484df   docker.io/confluentinc/cp-zookeeper:7.3.1      "/etc/confluent/dock...  zookeeper              58 seconds ago
Up            0.0.0.0:2181->2181/tcp
```

Debemos fijarnos que todos los contenedores estén UP. Para comprobar que todo funciona correctamente, podemos entrar a la siguiente URL:

<http://localhost:9021/>

Deberíamos ver el Confluent con nuestro cluster activo:

CONFLUENT

Home

1 Healthy clusters 0 Unhealthy clusters

Search cluster name or id

controlcenter.cluster

Running

Overview

Brokers	1
Partitions	194
Topics	55
Production	14.72KB/s
Consumption	11.18KB/s

Connected services

ksqldb clusters	1
Connect clusters	1

Podemos entrar a este para comprobar la información que necesitamos. Con esto ya tenemos Kafka activo y funcionando.

CONFIGURACIÓN MICROSERVICIOS

Para este apartado, como ya he comentado en la introducción, la base que usaremos ya está hecha de la anterior POC, por lo que hasta aquí ya deberíamos tener las dependencias necesarias para que funcione correctamente el config-server, el repositorio con los distintos perfiles, etc. Y sobre todo tener hecha la parte del **Client Actuator**, que empieza al final de la pág. 9 del anterior documento.

Por eso, vamos a empezar directamente con el Spring Cloud Bus. Para esto, como ya tenemos configurado Kafka, lo primero será añadir la siguiente dependencia a todos los microservicios (no al config-server):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-kafka</artifactId>
  <version>4.3.0</version>
  <exclusions>
    <!-- Para evitar conflicto entre commons logging y spring-jcl -->
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Y seguidamente en sus respectivos application.yml añadiremos la siguiente configuración:

```
#Kafka
---
spring:
  cloud:
    bus:
      enabled: true
      refresh:
        enabled: true
    stream:
      bindings:
        springCloudBusInput:
          destination: springCloudBus
          group: product-service-group
        springCloudBusOutput:
          destination: springCloudBus
    kafka:
      binder:
        brokers: localhost:9092
        auto-create-topics: true
```

Cada microservicio deberá tener en la parte de **group**: el siguiente formato:

{nombre-del-servicio}-group

Una vez tengamos esto, tendremos un nuevo endpoint.

/actuator/busrefresh

Esto nos permitirá llamar a un microservicio para que se actualice tanto ese como todos los demás a la vez. De esta manera no tenemos que ir uno por uno llamándolos para que se actualicen con el *actuator/refresh*.

Por ultimo solo debemos añadir dos cosas al config-server. Primero las siguientes dependencias:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-monitor</artifactId>
  <version>4.2.2</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

Y por último su configuración para su application.yml:

```
#Kafka
---
spring.cloud:
  bus:
    enabled: true
    destination: springCloudBus
  stream:
    bindings:
      springCloudBusInput:
        destination: springCloudBus
        group: config-server-group
      springCloudBusOutput:
        destination: springCloudBus

    kafka:
      binder:
        brokers: localhost:9092
        auto-create-topics: true
```

Y con esto habremos terminado, con la dependencia de monitor nos permite crear un WEBHOOK para ponerlo en nuestro repositorio y que se llame automáticamente a <http://localhost:8888/monitor>

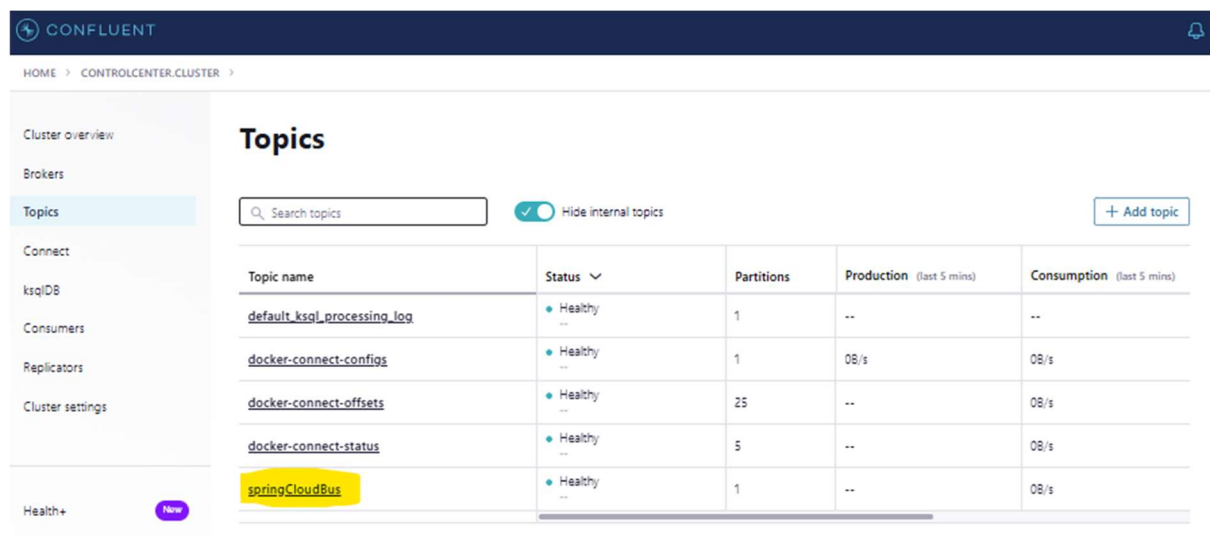
De todas manera, esta todo bien explicado en el anterior documento, en la pág. 14 se explica los del **bushrefresh**, y en la siguiente, la pág.15 se explica la dependencia de **monitor**.

EXTRA

Cabe destacar que para esta POC no he podido comprobar que el endpoint de monitor funcione con un webhook, ya que está en un entorno local y el ordenador de NTT no permite hacer tunneling o exponer los puertos locales. El endpoint de busrefresh si que he podido comprobarlo y todo funciona correctamente junto a Kafka, se actualizan todos los microservicios que han tenido algún cambio.

De todas maneras, la configuración es la misma que en la anterior POC y en esa si que pude comprobar que todo funcionara con un webhook, por lo tanto creo que el comportamiento debería ser el mismo.

Adjunto imagen del tópicos que se crea al iniciar los microservicios de esta POC, y que se utiliza para actualizarlos:



The screenshot shows the Confluent Topics page. On the left is a sidebar with navigation links: Cluster overview, Brokers, Topics (selected), Connect, ksqlDB, Consumers, Replicators, and Cluster settings. At the bottom of the sidebar is a 'Health+' button with a 'Now' indicator. The main area is titled 'Topics' and contains a search bar, a 'Hide internal topics' toggle (which is checked), and an '+ Add topic' button. Below this is a table with the following columns: Topic name, Status, Partitions, Production (last 5 mins), and Consumption (last 5 mins). The table lists five topics, all with a status of 'Healthy'.

Topic name	Status	Partitions	Production (last 5 mins)	Consumption (last 5 mins)
default_ksql_processing_log	Healthy	1	--	--
docker-connect-configs	Healthy	1	0B/s	0B/s
docker-connect-offsets	Healthy	25	--	0B/s
docker-connect-status	Healthy	5	--	0B/s
springCloudBus	Healthy	1	--	0B/s

IMAGEN DE KAFKA

Esta imagen fue compartida por mi equipo para que pudiera tener un entorno de Kafka desplegado y poder hacer así las pruebas de la POC.

¡PARA MI PROYECTO HE COMENTADO LA ULTIMA LINEA DEL DOCUMENTO, YA QUE ME GENERABA ERRORES. PERO SI SE UTILIZA ESTA IMAGEN QUIZAS ES NECESARIO DESCOMENTARLA DE NUEVO!

```
152     ksqldb-cli:  
153         image: confluentinc/cp-ksqldb  
154         container_name: ksqldb-cli  
155         depends_on:  
156             - broker  
157             - connect  
158             - ksqldb-server  
159         entrypoint: /bin/sh  
160         #tty: true
```

Si no me equivoco esta línea sirve para crear una terminal virtual, que en mi caso no era necesario.

CONTENIDO:

version: "3.4"

services:

broker:

container_name: broker

depends_on:

- zookeeper

environment:

KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092

KAFKA_BROKER_ID: 1

KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0

KAFKA_JMX_HOSTNAME: localhost

KAFKA_JMX_PORT: 9101

KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT

KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1

KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1

KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181

KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.ConfluentMetricsReporter

CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092

CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1

CONFLUENT_METRICS_ENABLE: 'true'

CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'

hostname: broker

image: confluentinc/cp-server:7.3.1

ports:

- 29092:29092
- 9092:9092
- 9101:9101

schema-registry:

container_name: schema-registry

depends_on:

- broker

environment:

SCHEMA_REGISTRY_HOST_NAME: schema-registry

SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: broker:29092

SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081

hostname: schema-registry

image: confluentinc/cp-schema-registry:7.3.1

platform: linux/arm64/v8

ports:

- 8081:8081

zookeeper:

container_name: zookeeper

environment:

ZOOKEEPER_CLIENT_PORT: 2181

ZOOKEEPER_TICK_TIME: 2000

hostname: zookeeper

image: confluentinc/cp-zookeeper:7.3.1

ports:

- 2181:2181

control-center:

container_name: control-center

depends_on:

- broker
- connect

environment:

CONTROL_CENTER_BOOTSTRAP_SERVERS: 'broker:9092'

CONTROL_CENTER_CONNECT_DC1_CLUSTER: http://connect:8083

CONTROL_CENTER_CONNECT_CONNECT-DC1_CLUSTER: 'connect:8083'

CONTROL_CENTER_KAFKA_DC1_BOOTSTRAP_SERVERS: 'broker:9092'

CONTROL_CENTER_REPLICATION_FACTOR: 1

CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1

CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1

CONTROL_CENTER_DEPRECATED_VIEWS_ENABLE: "true"

CONFLUENT_METRICS_TOPIC_REPLICATION: 1

PORT: 9021

CONTROL_CENTER_BOOTSTRAP_SERVERS: broker:29092

CONTROL_CENTER_SCHEMA_REGISTRY_URL: http://schema-registry:8081

CONTROL_CENTER_REPLICATION_FACTOR: 1

CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1

CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1

CONFLUENT_METRICS_TOPIC_REPLICATION: 1

PORT: 9021

CONTROL_CENTER_CONNECT_CONNECT-DEFAULT_CLUSTER: 'connect:8083'

CONTROL_CENTER_KSQL_KSQLDB1_URL: "http://ksqldb-server:8088"

CONTROL_CENTER_KSQL_KSQLDB1_ADVERTISED_URL: "http://localhost:8088"

hostname: control-center

image: confluentinc/cp-enterprise-control-center:7.3.1

ports:

- 9021:9021

rest-proxy-local:

image: confluentinc/cp-kafka-rest:7.3.1

depends_on:

- broker

- schema-registry

ports:

- 8082:8082

hostname: rest-proxy-local

container_name: rest-proxy-local

environment:

KAFKA_REST_HOST_NAME: rest-proxy-local

KAFKA_REST_BOOTSTRAP_SERVERS: 'broker:29092'

KAFKA_REST_LISTENERS: "http://0.0.0.0:8082"

KAFKA_REST_SCHEMA_REGISTRY_URL: 'http://schema-registry:8081'

connect:

image: cnfldemos/cp-server-connect-datagen:0.5.3-7.1.0

hostname: connect

container_name: connect

depends_on:

- broker
- schema-registry

ports:

- "8083:8083"

environment:

CONNECT_BOOTSTRAP_SERVERS: 'broker:29092'

CONNECT_REST_ADVERTISED_HOST_NAME: connect

CONNECT_GROUP_ID: compose-connect-group

CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs

CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1

CONNECT_OFFSET_FLUSH_INTERVAL_MS: 10000

CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets

CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1

CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status

CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1

CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter

CONNECT_VALUE_CONVERTER: io.confluent.connect.avro.AvroConverter

CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: http://schema-registry:8081

CLASSPATH required due to CC-2422

CLASSPATH: /usr/share/java/monitoring-interceptors/monitoring-interceptors-7.3.1.jar

CONNECT_PRODUCER_INTERCEPTOR_CLASSES:

"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"

CONNECT_CONSUMER_INTERCEPTOR_CLASSES:

"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"

CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-components"

CONNECT_LOG4J_LOGGERS:
org.apache.zookeeper=ERROR,org.I0ltec.zkclient=ERROR,org.reflections=ERROR

ksqldb-server:

image: confluentinc/cp-ksqldb-server:7.3.1

hostname: ksqldb-server

container_name: ksqldb-server

depends_on:

- broker

- connect

ports:

- "8088:8088"

environment:

KSQL_CONFIG_DIR: "/etc/ksql"

KSQL_BOOTSTRAP_SERVERS: "broker:29092"

KSQL_HOST_NAME: ksqldb-server

KSQL_LISTENERS: "http://0.0.0.0:8088"

KSQL_CACHE_MAX_BYTES_BUFFERING: 0

KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"

KSQL_PRODUCER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"

KSQL_CONSUMER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"

KSQL_KSQL_CONNECT_URL: "http://connect:8083"

KSQL_KSQL_LOGGING_PROCESSING_TOPIC_REPLICATION_FACTOR: 1

KSQL_KSQL_LOGGING_PROCESSING_TOPIC_AUTO_CREATE: 'true'

KSQL_KSQL_LOGGING_PROCESSING_STREAM_AUTO_CREATE: 'true'

ksqldb-cli:

image: confluentinc/cp-ksqldb-cli:7.3.1

container_name: ksqldb-cli

depends_on:

- broker

- connect

- ksqldb-server

entrypoint: /bin/sh

#tty: true