How to setup and configure Apache Kafka for Open-TelekomCloud

Introduction

This is step-by-step guide, which will help you to setup Apache Kafka in Open Telekom Cloud with Kubernetes. There are 3 different options described in this document. You can choose by your own which one suits you the most, depending on your use-case.

The first approach with Bitnami kafka helm chart is multiple times tested and used in our production environments.

> **Please keep in mind that the configuration of kafka highly depends on your projects and setup. There is no bullet-proof production ready single setup. The setup and configuration should be done by experts like iits-consulting who have enough experience in cloud native and kafka.**

Requirements

- OTC CCE cluster
- Kubectl properly configured for your Kubernetes cluster
- Helm package manager
- Kafkacat (optional)

Setup with helm

In this example bitnami Apache Kafka helm-chart was used https://github.com/bitnami/charts/tree/master/bi

- Add Helm chart repository

  ```
  helm repo add bitnami https://charts.bitnami.com/bitnami
  ```

- Install the chart

  ```
  helm upgrade --install kafka bitnami/kafka \
  --create-namespace \
  --set global.storageClass='csi-disk'
  ```

Configuration

This Helm Charts provides a default configuration for kafka. In production you most probably want to override the default configuration which can be easily done by setting helm values.

Create a file called *values.yaml* and adjust it to your needs like this:

```
global:
  storageClass: "csi-disk"
replicaCount: 3
maxMessageBytes: _52428800
autoCreateTopicsEnable: false
deleteTopicEnable: false
metrics:
  kafka:
    enabled: true
  jmx:
    enabled: true
persistence:
  size: 30G
```

Now install the chart like this: `shell  helm upgrade --install kafka bitnami/kafka \  --create-namespace \  -f values.yaml`

> More information about variables, that can be overridden you can find here

Write a test message to a topic (Optional)

- Run Kafka Client

  ```
  kubectl run kafka-client --restart='Never' --image docker.io/bitnami/kafka:2.8.0-debiar
  ```

- Start consumer

  ```
  kubectl exec --tty -i kafka-client --namespace kafka -- kafka-console-consumer.sh --boo
  ```

- Open another terminal instance (window or tab)

- Start producer

  ```
  kubectl exec --tty -i kafka-client --namespace kafka -- kafka-console-producer.sh --bro
  ```

- Start produce messages line by line and check results in consumer

Setup a admin UI for Kafka (Optional)

If you want a pretty UI to interact with kafka then we would recommend installing akhq

Another option is Confluent Control Center but to use this service you need to buy a license from Confluent.

- Add the AKHQ helm charts repository:

```
helm repo add akhq https://akhq.io/
```

Create a file called *values.yaml* and adjust it to your needs like this:

```yaml
image:
  tag: 0.17.0
resources:
  requests:
    memory: "400Mi"
    cpu: "1m"
replicaCount: 1
configuration: |
  akhq:
    connections:
      kafka:
        properties:
          bootstrap.servers: "kafka:9092"
    server:
      access-log:
        enabled: true
        name: org.akhq.log.access
  micronaut:
    server:
      netty:
        max-initial-line-length: 16384
        max-header-size: 32768
      context-path: "/akhq"
readinessProbe:
  prefix: "/akhq"
```

Now install the chart like this: `shell   helm upgrade --install akhq akhq/akhq \   --create-namespace \   -f values.yaml`

After that akhq you can expose your service over ingress or just make a kubectl port-forward to access the ui like this: `shell kubectl -n kafka port-forward svc/kafka-akhq 8080:80`

Open the browser and access http://localhost:8080/akhq/ui/kafka/topic

Setup with Strimzi Kafka Operator

In this example bitnami Apache Kafka helm-chart was used https://github.com/bitnami/charts/tree/master/bi

Benefits and Cautions

Operators are quite smart in how they manage applications in Kubernetes. Usually, you need to define only high-level parameters like CPU, Memory, Storage, Authentication, Encryption etc. Operator will take care about Kubernetes resources by your requirements. It can automate certificate management.

You have additional abstraction level - complexity of the system potentially can bring problems. Engineers need to have additional knowledge. Besides Cloud Technologies, Kubernetes, Helm they need to know how this exact operator works.

Apply Strimzi installation files

- Create namespace

  ```
  kubectl create ns kafka
  ```

- This command will create all needed CRD's inside your cluster

  ```
  kubectl create -f 'https://strimzi.io/install/latest?namespace=kafka' -n kafka
  ```

- You can check that strimzi-cluster-operator successfully started by

  ```
  kubectl logs deployment/strimzi-cluster-operator -n kafka -f
  ```

Provision Apache Kafka cluster

- Save snippet below to `kafka-cluster.yml` file:

  ```
  apiVersion: kafka.strimzi.io/v1beta2
  kind: Kafka
  metadata:
    name: my-cluster
  spec:
    kafka:
      version: 2.8.0
      replicas: 1
      listeners:
        - name: plain
          port: 9092
          type: internal
  ```

```yaml
          tls: false
        - name: tls
          port: 9093
          type: internal
          tls: true
      config:
        offsets.topic.replication.factor: 1
        transaction.state.log.replication.factor: 1
        transaction.state.log.min.isr: 1
        log.message.format.version: "2.8"
        inter.broker.protocol.version: "2.8"
      storage:
        type: ephemeral
        volumes:
        - id: 0
          type: ephemeral
          size: 100Gi
          deleteClaim: false
    zookeeper:
      replicas: 1
      storage:
        type: ephemeral
        size: 100Gi
        deleteClaim: false
    entityOperator:
      topicOperator: {}
      userOperator: {}
```

- Apply changes by `kubectl apply -f kafka-cluster.yml`
- Wait for pods starts

  ```
  kubectl wait kafka/my-cluster --for=condition=Ready --timeout=300s -n my-kafka-project
  ```

Try to send and receive messages

- Run Kafka Client

  ```
  kubectl run kafka-client --restart='Never' --image docker.io/bitnami/kafka:2.8.0-debian
  ```

- Start consumer

  ```
  kubectl exec --tty -i kafka-client --namespace kafka -- kafka-console-consumer.sh --boo
  ```

- Open another terminal instance (window or tab)
- Start producer

  ```
  kubectl exec --tty -i kafka-client --namespace kafka -- kafka-console-producer.sh --bro
  ```

- Start produce messages line by line and check results in consumer

Setup with Plain Kubernetes manifests

Benefits and Cautions

We would not recommend this approach. You don't have elasticity in terms of configuration. Since there is no any packaging (like helm) you cannot use benefits of versioning and templating.

But if you are not familiar with helm and kubernetes operator you can also use kubernetes manifests. If you need to apply these manifests in different environments with different configuration – you should duplicate your code below

This option should be used for **testing** purposes. No additional tools and pre-configuration steps needed. You are using plain Kubernetes manifests with standard API objects.

Create Zookeeper

- Create namespace

  ```
  kubectl create ns kafka
  ```

- Save snippet below to `zookeeper-statefullset.yml` file:

  ```yaml
  apiVersion: apps/v1
  kind: StatefulSet
  metadata:
    name: zookeeper
  spec:
    selector:
      matchLabels:
        app: zookeeper
    serviceName: zookeeper
    replicas: 1
    template:
      metadata:
        labels:
          app: zookeeper
      spec:
        containers:
        - name: zoo1
          image: zookeeper
          imagePullPolicy: IfNotPresent
          resources:
            requests:
              cpu: 128m
              memory: 500Mi
            limits:
              cpu: 128m
  ```

```
            memory: 500Mi
        ports:
        - containerPort: 2181
        env:
        - name: ZK_SERVER_HEAP
          value: "256"
        - name: ZOOKEEPER_ID
          value: "1"
        - name: ZOOKEEPER_SERVER_1
          value: zoo1
```

- Apply changes

  ```
  kubectl apply -f zookeeper-statefullset.yml
  ```

Expose Zookeeper service

- Save snippet below to `zookeeper-service.yml` file:

  ```
  apiVersion: v1
  kind: Service
  metadata:
    name: zookeeper
    labels:
      app: zookeeper
  spec:
    ports:
    - name: client
      port: 2181
      protocol: TCP
    - name: follower
      port: 2888
      protocol: TCP
    - name: leader
      port: 3888
      protocol: TCP
    selector:
      app: zookeeper
  ```

- Apply changes

  ```
  kubectl apply -f zookeeper-service.yml
  ```

Create Broker

- Save snippet below to `broker-statefullset.yml` file:

  ```
  apiVersion: apps/v1
  kind: StatefulSet
  ```

```yaml
metadata:
  name: broker
spec:
  selector:
    matchLabels:
      app: broker
  serviceName: broker
  replicas: 1
  template:
    metadata:
      labels:
        app: broker
    spec:
      containers:
      - name: kafka
        image: wurstmeister/kafka
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9092
        - containerPort: 9094
        resources:
          requests:
            cpu: 128m
            memory: 1Gi
          limits:
            cpu: 128m
            memory: 1Gi
        env:
        - name: "KAFKA_HEAP_OPTS"
          value: "-Xmx512M -Xms512M"
        - name: KAFKA_LISTENERS
          value: "INSIDE://:9094,OUTSIDE://localhost:9092"
        - name: KAFKA_ADVERTISED_LISTENERS
          value: "INSIDE://:9094,OUTSIDE://localhost:9092"
        - name: KAFKA_LISTENER_SECURITY_PROTOCOL_MAP
          value: "INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT"
        - name: KAFKA_INTER_BROKER_LISTENER_NAME
          value: INSIDE
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: zookeeper:2181
        - name: KAFKA_BROKER_ID
          value: "0"
```

- Apply changes

```
kubectl apply -f broker-statefullset.yml
```

Expose Broker service

- Save snippet below to `kafka-service.yml` file:

  ```yaml
  apiVersion: v1
  kind: Service
  metadata:
    name: broker
    labels:
      app: broker
  spec:
    ports:
    - port: 9092
      name: broker-port
      protocol: TCP
    selector:
      app: broker
    type: ClusterIP
  ```

  Apply changes

  ```
  kubectl apply -f kafka-service.yml
  ```

Try to send and receive messages

- Forward Broker service to your local machine

  ```
  kubectl port-forward service/broker -n kafka 9092:9092
  ```

- Produce something like

  ```
  kcat -b localhost:9092 -t test-topic -P <<EOF
  hello
  world
  EOF
  ```

- Consume it by

  ```
  kcat -b localhost:9092 -t test-topic -C
  ```

- When tests will finish, just remove namespace

  ```
  kubectl delete ns kafka
  ```

Things to mention

- We would recommend if you use multiple stages to have one kafka cluster for each stage. Then you don't need to thing how to expose your kafka cluster and also the topic creation and consuming messages is way much easier. From the security side it is much more secure if the kafka system is not reachable from outside

- Setting up kafka is easy but to configure it correctly is the hard part. We would recommend get some support from professionals like iits-consulting