How to start with Apache Kafka

Description

This is step-by-step guide, which will help you to start with Apache Kafka in OTC Cloud Container Engine. There are 3 different options described in this document. You can choose by your own which exact you need, depending on your use-case.

All tools and their versions described in this article you can find in the Requirements section below. Files mentioned here can be found here https://github.com/iits-consulting/otc-kafka-example

> **Please keep in mind that we are not pretending to have production-ready guide that you should follow up without worries. Production systems setup and configuration must be done by persons who have enough experience in Cloud Technologies and Kafka platform.**

Requirements

- OTC CCE cluster
- Kubectl configured for your Kubernetes cluster context properly
- Helm package manager (for options B and C)
- Kafkacat (optional)

Option A. Plain Kubernetes manifests

Benefits and Cautions

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. Configuration as transparent as possible.

Negative side – 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. If you need to apply these manifests in different environments with different configuration – you should duplicate your code below.

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
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
  ```

Option B. Official Kafka Helm chart

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

Benefits and Cautions

Most of the things that you usually need to configure Apache Kafka properly already exists in this Helm chart. Default configuration/behavior can be easily changed by overriding values. By Helm you can simplify transition to GitOps for your company now or in the future.

By the other hand entry level for maintaining this solution a bit bigger, because of templating mechanism complexity. Usually, it does not take much time to sort out with Helm templating mechanism.

Install Helm chart with your variables

- Add Helm chart repository

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

- Override default variables as (if) you need

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

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

Check that everything works (Optional)

- 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

Option C. 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:

```yaml
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
        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
  ```

Check that it works (Optional)

- Start with forwarding broker port locally

  ```
  kubectl port-forward service/my-cluster-kafka-brokers -n kafka 9092:9092
  ```

- 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

Conclusion

We described here 3 different options which can be used to start with Apache Kafka in OTC CCE solution. A lot of other options also available (using different helm-chart, write your own, use any other Kafka-Operator, using kustomization or ksonnet files).

You need to keep in mind that all recommendations and best practices (as low-latency network and storage, high-availability, data protection and security) for Kafka on Kubernetes cluster should be followed as much as possible