Cloudera Streams Messaging - Kubernetes Operator 1.3.0

Kafka Deployment and Configuration

Date published: 2024-06-11
Date modified: 2025-02-28

https://docs.cloudera.com/

Legal Notice

Contents

Deploying Kafka

You deploy a Kafka cluster by creating a Kafka resource and one or more KafkaNodePool resources in the Kubernetes cluster. The Kafka cluster can use either KRaft (recommended) or ZooKeeper (deprecated) for metadata management. After cluster deployment you can validate your cluster with the console producer and consumer tools shipped with Kafka.

The `Kafka` resource describes a Kafka cluster instance. This resource specifies the following about Kafka:

- Kafka configuration that is common for the whole Kafka cluster (Kafka version, cluster name, and so on)
- Cruise Control configuration
- Entity Operator configuration
- ZooKeeper configuration (if ZooKeeper is used instead of KRaft)

A `KafkaNodePool` resource refers to a distinct group of Kafka nodes within a Kafka cluster. Using node pools enables you to specify different configurations for each node within the same Kafka cluster. Configuration options not specified in the node pool are inherited from the Kafka configuration.

You can deploy a Kafka cluster with one or more node pools. The number of node pools you create depends on how many groups of Kafka brokers you want to have that have differing configurations. The node pool configuration includes mandatory and optional settings. Configuration for replicas, roles, and storage is mandatory.

KRaft versus ZooKeeper

You can deploy Kafka in either KRaft or ZooKeeper mode. However, Cloudera recommends that you deploy clusters in KRaft mode. This is because ZooKeeper-based clusters are deprecated. Additionally, ZooKeeper will be removed in a future release.

KRaft offers enhanced reliability, scalability, and throughput over ZooKeeper. Metadata operations are more efficient as they are directly integrated. Additionally, when using KRaft, you are no longer required to maintain ZooKeeper, which reduces operational overhead.

Kafka node roles and deployment modes in KRaft

When you deploy a Kafka cluster in KRaft mode, you assign roles to each node in the Kafka cluster. Roles are assigned in the `KafkaNodePool` resource. There are two roles, broker and controller.

- Broker # These nodes manage Kafka records stored in topic partitions. Nodes with the broker role are your Kafka brokers.
- Controller # These nodes manage cluster metadata and the state of the cluster using a Raft-based consensus protocol. Controller nodes are the KRaft equivalent of ZooKeeper nodes.

A single Kafka node can have a single role or both roles. If you assign both roles to the node, it performs both broker and controller tasks. Depending on role assignments, your cluster will be running in one of the following modes.

- KRaft mode # In this mode, each Kafka node is either a broker or controller. Recommended for production clusters.
- KRaft combined mode # In this mode, some or all nodes in the cluster have both controller and broker roles assigned to them.

Combined mode is not recommended or supported for production environments. Use combined mode in development environments. Cloudera recommends that you always **fully** separate controller and broker nodes to avoid resource contention between roles.

Deploying a Kafka cluster in KRaft mode

You deploy a Kafka cluster in KRaft mode by deploying a Kafka resource and at least two KafkaNodePool resources. One KafkaNodepool describes your brokers, the other describes KRaft controllers. The Kafka resource must include the strimzi.io/kraft="enabled" annotation. Deploying Kafka in KRaft mode is the recommended mode for deployment.

Before you begin

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that a namespace is available where you can deploy your cluster. If not, create one.

```
kubectl create namespace[***NAMESPACE***]
```

- Ensure that the `Secret` containing credentials for the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted is available in the namespace where you plan on deploying your cluster. If the secret is not available, create it.

```
kubectl create secret docker-registry [***SECRET NAME***] \
  --docker-server [***REGISTRY***] \
  --docker-username [***USERNAME***] \
  --docker-password [***PASSWORD***] \
  --namespace [***NAMESPACE***]
```

  - *[***SECRET NAME***]* must be the same as the name of the `Secret` containing registry credentials that you created during Strimzi installation.
  - Replace *[***REGISTRY***]* with the server location of the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted. If your Kubernetes cluster has internet access, use cont ainer.repository.cloudera.com. Otherwise, enter the server location of your self-hosted registry.
  - Replace *[***USERNAME***]* and *[***PASSWORD***]* with credentials that provide access to the registry. If you are using container.repository.cloudera.com, use your Cloudera credentials. Otherwise, enter credentials providing access to your self-hosted registry.
- Scaling node pools that include KRaft controllers (controller roles) is not possible.
- The following steps contain `Kafka` and `KafkaNodePool` resource examples. You can find additional examples on the Cloudera Archive.

Procedure

1. Create a YAML configuration containing your `Kafka` resource manifest.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  annotations:
    strimzi.io/node-pools: enabled
    strimzi.io/kraft: enabled
spec:
  kafka:
    version: 3.9.0.1.3
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
```

```
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      default.replication.factor: 3
      min.insync.replicas: 2
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

- strimzi.io/node-pools: enabled - Enables Kafka node pools. KRaft mode is only supported with node pools.
- strimzi.io/kraft: enabled - Enables KRaft mode for the cluster.
- spec.kafka.version - Specifies the Kafka version to use. Must specify a Cloudera Kafka version supported by Cloudera Streams Messaging - Kubernetes Operator. For example, 3.9.0.1.3. Do not add Apache Kafka versions, they are not supported. You can find a list of supported Kafka versions in the Release Notes.

2. Create a YAML configuration containing your `KafkaNodePool` resource manifest for brokers.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: broker
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 10Gi
        kraftMetadata: shared
        deleteClaim: false
```

- spec.roles - Specifies the roles of the nodes in this pool. The value broker means that the replicas in this node pool are all brokers.
- spec.storage.volumes.kraftMetadata - Specifies whether a volume should be used for storing KRaft metadata. Used to specify which volume should be used to store metadata. In this example, volume 0 is specified for storage. This property is optional.

3. Create a YAML configuration containing your `KafkaNodePool` resource manifest for KRaft controllers.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: controller
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 10Gi
```

```
        kraftMetadata: shared
        deleteClaim: false
```

- spec.roles - Specifies the roles of the nodes in this pool. The value controller means that the replicas in this node pool are all KRaft controllers.
- spec.storage.volumes.kraftMetadata - Specifies whether a volume should be used for storing KRaft metadata. Used to specify which volume should be used to store metadata. In this example, volume 0 is specified for storage. This property is optional.

4. Deploy the cluster.

```
kubectl apply \
  --filename [**KAFKA YAML***],[***BROKER NODE POOL
 YAML***],[***CONTROLLER NODE POOL YAML***] \
  --namespace [***NAMESPACE***]
```

5. Verify that pods are created.

```
kubectl get pods --namespace [***NAMESPACE***]
```

If cluster deployment is successful, you should see an output similar to the following.

```
NAME                                          READY   STATUS    RESTARTS
my-cluster-broker-0                           1/1     Running   0
my-cluster-broker-1                           1/1     Running   0
my-cluster-broker-2                           1/1     Running   0
my-cluster-controller-3                       1/1     Running   0
my-cluster-controller-4                       1/1     Running   0
my-cluster-controller-5                       1/1     Running   0
my-cluster-entity-operator-858b7649df-v8jth   2/2     Running   0
strimzi-cluster-operator-589f9fd659-4bqnp     1/1     Running   0
```

The READY column shows the number of ready and total containers inside the pod, while the STATUS column shows if the pod is running or not.

In this example there are a total of six nodes (each node is a pod). Three are dedicated brokers, the other three are dedicated controllers.

What to do next

Validate your cluster. Complete Validating a Kafka cluster.

Related Information

Broker Configs | Apache Kafka

Deploying a Kafka cluster in KRaft mode | Strimzi

Kafka schema reference | Strimzi API Reference

KafkaNodePool schema reference | Strimzi API Reference

Deploying a Kafka cluster in KRaft combined mode

You deploy a Kafka cluster in KRaft combined mode by deploying a Kafka resource and one or more KafkaNodePool resources. Typically you create two node pools, one describing nodes with both roles, and one that

describes nodes that have the broker role only. Alternatively, you can create clusters where all nodes have both roles. In this case, a single node pool is sufficient. The Kafka resource must include the strimzi.io/kraft="enabled" annotation.

Important: Cloudera does not recommend that you use combined mode in production environments.

Before you begin

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that a namespace is available where you can deploy your cluster. If not, create one.

```
kubectl create namespace[***NAMESPACE***]
```

- Ensure that the `Secret` containing credentials for the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted is available in the namespace where you plan on deploying your cluster. If the secret is not available, create it.

```
kubectl create secret docker-registry [***SECRET NAME***] \
  --docker-server [***REGISTRY***] \
  --docker-username [***USERNAME***] \
  --docker-password [***PASSWORD***] \
  --namespace [***NAMESPACE***]
```

  - *[***SECRET NAME***]* must be the same as the name of the `Secret` containing registry credentials that you created during Strimzi installation.
  - Replace *[***REGISTRY***]* with the server location of the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted. If your Kubernetes cluster has internet access, use cont ainer.repository.cloudera.com. Otherwise, enter the server location of your self-hosted registry.
  - Replace *[***USERNAME***]* and *[***PASSWORD***]* with credentials that provide access to the registry. If you are using container.repository.cloudera.com, use your Cloudera credentials. Otherwise, enter credentials providing access to your self-hosted registry.
- Scaling node pools that include KRaft controllers (controller roles) is not possible.

  Because of this limitation, you can only scale clusters running in combined mode if the cluster includes a node pool that has broker nodes only. The examples in these steps set up a broker-only node pool.
- Ranger authorization does not work with combined mode.
- The following steps contain `Kafka` and `KafkaNodePool` resource examples. You can find additional examples on the Cloudera Archive.

Procedure

1. Create a YAML configuration containing your `Kafka` resource manifest.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  annotations:
    strimzi.io/node-pools: enabled
    strimzi.io/kraft: enabled
spec:
  kafka:
    version: 3.9.0.1.3
    listeners:
      - name: plain
        port: 9092
        type: internal
```

```
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      default.replication.factor: 3
      min.insync.replicas: 2
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

- strimzi.io/node-pools: enabled - Enables Kafka node pools. KRaft mode is only supported with node pools.
- strimzi.io/kraft: enabled - Enables KRaft mode for the cluster.
- spec.kafka.version - Specifies the Kafka version to use. Must specify a Cloudera Kafka version supported by Cloudera Streams Messaging - Kubernetes Operator. For example, 3.9.0.1.3. Do not add Apache Kafka versions, they are not supported. You can find a list of supported Kafka versions in the Release Notes.

2. Create a YAML configuration containing your `KafkaNodePool` resource manifests.

   The configuration and number of `KafkaNodePools` you create depends on the deployment architecture that you want.

   The following example creates two `KafkaNodePools`. One node pool specifies both the broker and controller roles. These nodes will run in combined mode. Additionally, a second node pool is created that includes broker nodes only.

   The second node pool is added because node pools that include controller nodes cannot be scaled. Creating a separate node pool for brokers when you first deploy the cluster makes it easier to scale the cluster in the future.

   Note: You can deploy a cluster with a single `KafkaNodePool` that has combined roles. However, in this case, if you want to scale the cluster, you must create a new `KafkaNodePool` that includes broker nodes only.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: combined
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 10Gi
        kraftMetadata: shared
        deleteClaim: false
---
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: broker-only
  labels:
    strimzi.io/cluster: my-cluster
spec:
```

```
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 10Gi
        kraftMetadata: shared
        deleteClaim: false
```

- spec.roles - Specifies the roles of the nodes in the pool. The combined node pool has both the controller and broker roles specified. Therefore, the three Kafka nodes described in the combined node pool operate in combined mode. On the other hand, the broker-only node pool has broker specified as the role. The three Kafka nodes described by the broker-only pool operate as brokers.
- spec.storage.volumes.kraftMetadata - Specifies whether a volume should be used for storing KRaft metadata. Used to specify which volume should be used to store metadata. In this example, volume 0 is specified for storage. This property is optional.

3. Deploy the cluster.

```
kubectl apply \
  --filename [**KAFKA YAML***],[***NODE POOL YAML***] \
  --namespace [***NAMESPACE***]
```

4. Verify that pods are created.

```
kubectl get pods --namespace [***NAMESPACE***]
```

If cluster deployment is successful, you should see an output similar to the following.

```
NAME                                          READY   STATUS    RESTARTS
my-cluster-broker-only-0                      1/1     Running   0
my-cluster-broker-only-1                      1/1     Running   0
my-cluster-broker-only-2                      1/1     Running   0
my-cluster-combined-3                         1/1     Running   0
my-cluster-combined-4                         1/1     Running   0
my-cluster-combined-5                         1/1     Running   0
my-cluster-entity-operator-74c95d6667-rstkf   2/2     Running   0
strimzi-cluster-operator-589f9fd659-4bqnp     1/1     Running   0
```

The READY column shows the number of ready and total containers inside the pod, while the STATUS column shows if the pod is running or not.

In this example, there are a total of six nodes (each node is a pod). Nodes 0, 1, and 2 are brokers, while nodes 3, 4, and 5 are both brokers and controllers. This means the cluster has a total of six brokers and three controllers.

What to do next

Validate your cluster. Complete Validating a Kafka cluster.

Related Information

Broker Configs | Apache Kafka

Deploying a Kafka cluster in KRaft mode | Strimzi
Kafka schema reference | Strimzi API Reference
KafkaNodePool schema reference | Strimzi API Reference

Deploying a Kafka cluster in ZooKeeper mode

You deploy a Kafka cluster in ZooKeeper mode by deploying a Kafka resource and at least a single KafkaNodePool resource. The Kafka resource must include ZooKeeper configuration.

Important: ZooKeeper-based clusters are deprecated. Additionally, ZooKeeper will be removed in a future release. Cloudera recommends that you deploy your Kafka cluster in KRaft mode instead. For more information, see Deploying a Kafka cluster in KRaft mode.

Warning: Strimzi allows creating Kafka brokers by creating only a single `Kafka` resource. However, Cloudera Streams Messaging - Kubernetes Operator only supports creating Kafka brokers by creating `KafkaNodePool` resources. Node pools allow for more flexible deployments with easier scaling options. Moreover, certain features like rack awareness and scaling are limited without node pools. Broker creation using the `Kafka` resource only is deprecated, and results in unnecessary effort of migrating the deployment to use node pools.

Before you begin

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that a namespace is available where you can deploy your cluster. If not, create one.

```
kubectl create namespace[***NAMESPACE***]
```

- Ensure that the `Secret` containing credentials for the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted is available in the namespace where you plan on deploying your cluster. If the secret is not available, create it.

```
kubectl create secret docker-registry [***SECRET NAME***] \
  --docker-server [***REGISTRY***] \
  --docker-username [***USERNAME***] \
  --docker-password [***PASSWORD***] \
  --namespace [***NAMESPACE***]
```

  - *[***SECRET NAME***]* must be the same as the name of the `Secret` containing registry credentials that you created during Strimzi installation.
  - Replace *[***REGISTRY***]* with the server location of the Docker registry where Cloudera Streams Messaging - Kubernetes Operator artifacts are hosted. If your Kubernetes cluster has internet access, use cont ainer.repository.cloudera.com. Otherwise, enter the server location of your self-hosted registry.
  - Replace *[***USERNAME***]* and *[***PASSWORD***]* with credentials that provide access to the registry. If you are using container.repository.cloudera.com, use your Cloudera credentials. Otherwise, enter credentials providing access to your self-hosted registry.
- The following steps contain `Kafka` and `KafkaNodePool` resource examples. You can find additional examples on the Cloudera Archive.

Procedure

1. Create a YAML configuration containing both your `Kafka` and `KafkaNodePool` resource manifests.

   The following examples deploy a simple Kafka cluster with three replicas in a single node pool.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
```

```
  name: first-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
---
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  annotations:
    strimzi.io/node-pools: enabled
spec:
  kafka:
    version: 3.9.0.1.3
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      default.replication.factor: 3
      min.insync.replicas: 2
  zookeeper:
    replicas: 3
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
  cruiseControl: {}
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

- The spec.kafka.version property in the `Kafka` resource must specify a Cloudera Kafka version supported by Cloudera Streams Messaging - Kubernetes Operator. For example, 3.9.0.1.3. Do not add Apache Kafka versions, they are not supported. You can find a list of supported Kafka versions in the *Release Notes*.
- You can find additional information about the properties configured in this example in the Strimzi and Apache Kafka documentation.

2. Deploy the cluster.

```
kubectl apply --filename [***YAML CONFIG***] --namespace [***NAMESPACE***]
```

3. Verify that pods are created.

```
kubectl get pods --namespace [***NAMESPACE***]
```

If cluster deployment is successful, you should see an output similar to the following.

```
NAME                                          READY   STATUS    RESTARTS
my-cluster-entity-operator-79846c5cbd-jqn9k   2/2     Running   0
my-cluster-cruise-control-8475c5gdw0-juqi7h   1/1     Running   0
my-cluster-first-pool-0                       1/1     Running   0
my-cluster-first-pool-1                       1/1     Running   0
my-cluster-first-pool-2                       1/1     Running   0
my-cluster-zookeeper-0                        1/1     Running   0
my-cluster-zookeeper-1                        1/1     Running   0
my-cluster-zookeeper-2                        1/1     Running   0
strimzi-cluster-operator-5b465446b8-jfpmr     1/1     Running   0
```

The READY column shows the number of ready and total containers inside the pod, while the STATUS column shows if the pod is running or not.

In this example there are a total of six nodes (each node is a pod). Three are Kafka broker nodes, the other three are ZooKeeper nodes.

What to do next

What to do next

Validate your cluster. Complete Validating a Kafka cluster.

Related Information

Broker Configs | Apache Kafka

Deploying a ZooKeeper-based Kafka cluster | Strimzi

Kafka schema reference | Strimzi API Reference

KafkaNodePool schema reference | Strimzi API Reference

Validating a Kafka cluster

After the Kafka broker pods are successfully started, you can use the Kafka console producer and consumer to validate the cluster. The following steps use the exact same docker images that were used to deploy the Kafka cluster by the Strimzi Cluster Operator. The images contain all the Kafka built-in tools and you can start a custom Kubernetes pod, starting the Kafka tools in the containers.

Before you begin

The following example commands assume that the cluster is configured with PLAINTEXT authentication and credentials do not need to be provided. If your cluster is secured, you will need to pass the corresponding security parameters in the command line as well.

Procedure

1. Create a topic.

```
IMAGE=$(kubectl get pod [***BROKER POD***] --namespace [***NAMESPACE***]
 --output jsonpath='{.spec.containers[0].image}')
```

```
kubectl run kafka-admin -it \
  --namespace [***NAMESPACE***] \
  --image=$IMAGE \
  --rm=true \
```

```
  --restart=Never \
  --command -- /opt/kafka/bin/kafka-topics.sh \
    --bootstrap-server [***CLUSTER NAME***]-kafka-bootstrap:9092 \
    --create \
    --topic my-topic
```

2. Produce message to the topic using the Kafka console producer.

```
kubectl run kafka-producer -it \
  --namespace [***NAMESPACE***] \
  --image=$IMAGE \
  --rm=true \
  --restart=Never \
  --command -- /opt/kafka/bin/kafka-console-producer.sh \
    --bootstrap-server [***CLUSTER NAME***]-kafka-bootstrap:9092 \
    --topic my-topic
```

Start typing to produce messages.

```
>hello
>csm
>operator
>^C
```

3. Consume the messages using the Kafka Console consumer.

```
kubectl run kafka-consumer -it \
  --namespace [***NAMESPACE***] \
  --image=$IMAGE \
  --rm=true \
  --restart=Never \
  --command -- /opt/kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server [***CLUSTER NAME***]-kafka-bootstrap:9092 \
    --topic my-topic \
    --from-beginning
```

If successful, the messages you produced are printed on the output.

```
>hello
>csm
>operator
```

Deploying Cruise Control

Learn how to deploy Cruise Control alongside your Kafka Cluster using cruiseControl properties in the Kafka resource. Deploying Cruise Control is optional but strongly recommended as it automates the partition rebalancing in the cluster.

About this task

You can deploy Cruise Control alongside a Kafka cluster by adding cruiseControl properties to your `Kafka` resource. Deploying Cruise Control creates a Cruise Control deployment that contains a Cruise Control pod.

If you specify an empty object (cruiseControl: {}), Cruise Control is deployed with the upstream recommended default configuration. You can customize the configuration of Cruise Control by specifying the required options in the cruiseControl property.

Before you begin

Cruise Control requires at least two Kafka brokers. If you try to add Cruise Control while there is only a single Kafka broker in the cluster, the deployment fails. Increase your broker replica count if necessary.

Procedure

1. Add a cruiseControl property to your `Kafka` resource.

```
#...
kind: Kafka
spec:
  cruiseControl: {}
```

2. Create or update your resource.

```
kubectl apply --filename [***YAML CONFIG***] --namespace [***NAMESPACE***]
```

3. Verify the status of the deployment.

```
kubectl get deployments --namespace [***NAMESPACE***]
```

   If deployment is successful, you should see a Cruise Control deployment in the output.

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
my-cluster-cruise-control     1/1     1            1           5m1s
```

   The READY column shows the number of replicas that are ready/expected. The deployment is successful when the AVAILABLE output shows 1.

What to do next

After Cruise Control is deployed, you can use `KafkaRebalance` resources to rebalance your cluster. Typically you initiate a rebalance process when scaling your cluster, but rebalances can be carried out at any time.

Related Information

Scaling brokers

CruiseControlSpec schema reference | Strimzi API Reference

KafkaRebalance schema reference | Strimzi API Reference

Rebalancing clusters using Cruise Control | Strimzi

Deploying and configuring the Strimzi Entity Operator

Learn how to deploy and configure the Strimzi Entity Operator in your cluster by configuring your Kafka resource. Deploying the Entity Operator is required if you want to use custom resources to manage Kafka topics and users in your cluster.

The Entity Operator is responsible for managing Kafka users (clients) and Kafka topics in your Kafka cluster. The Entity Operator comprises the following two operators.

- Strimzi Topic Operator – An operator application that creates and manages Kafka topics in your Kafka cluster with `KafkaTopic` resources.
- Strimzi User Operator – An operator application that creates and manages Kafka users in your Kafka cluster with `KafkaUser` resources.

To deploy and configure the Entity Operator you configure your `Kafka` resource to include the entityOperator property. The entityOperator property can include topicOperator and userOperator properties.

These properties specify which of the two operators are deployed with the Entity Operator. You can choose to deploy either the Topic or User Operator, or deploy both at once.

The following example deploys both the Topic and User Operator with default configurations.

```
#...
kind: Kafka
spec:
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

Note: Your configuration must include either the topicOperator or userOperator property. If neither are included, the Entity Operator is not deployed.

You can further configure all three operators by including additional supported properties in the configuration. The entityOperator property can include the template property that specifies configuration related to pod and deployment templates. The topicOperator and userOperator support various sub-properties that allow you to configure watched namespaces, reconciliation intervals, and others.

The Entity, Topic, and User Operator are deployed by the Strimzi Cluster Operator. On successful deployment, the Cluster Operator creates an Entity Operator deployment and pod. The Topic and User Operator run within the pod in their own containers.

Deploying the Topic or User Operator as standalone components is not supported in Cloudera Streams Messaging - Kubernetes Operator.

Related Information

EntityOperatorSpec schema reference | Strimzi API Reference

Deploying and configuring the Strimzi Topic Operator

You deploy and configure the Strimzi Topic Operator by configuring the entityOperator property in your Kafka resource to include topicOperator properties. Deploying the Topic Operator is required if you want to manage Kafka topics with KafkaTopic resources instead of the KafkaAdmin API.

About this task

The Topic Operator enables you to manage Kafka topics using `KafkaTopic` resources. In Cloudera Streams Messaging - Kubernetes Operator, you deploy the Topic Operator through the Strimzi Entity Operator. The Entity and Topic Operator are both deployed by the Strimzi Cluster Operator.

To deploy the Topic Operator, you configure the entityOperator property in your Kafka resource to include topi cOperator properties. You configure the Topic Operator by specifying additional sub-properties in the topicOperator property.

By default, the Topic Operator watches `KafkaTopic` resources in the namespace of the Kafka cluster deployed by the Cluster Operator. You can also specify a namespace to watch using the watchedNamespace property.

A single Topic Operator can watch a single namespace. One namespace should be watched by only one Topic Operator. If you are deploying multiple Kafka clusters into the same namespace, enable the Topic Operator for only one Kafka cluster or use the watchedNamespace property to configure the Topic Operators to watch other namespaces

Before you begin

- Strimzi must be installed in your cluster. The Strimzi Cluster Operator must be running. See Installation.
- For a full list of supported properties, see the EntityTopicOperatorSpec schema reference in the Strimzi API Reference.

Procedure

1. Edit the entityOperator property in your Kafka resource to include topicOperator properties.

   The following example configures the Topic Operator to watch a specified namespace. Additionally, it configures the reconciliation interval as well as various resource properties.

   ```
   #...
   kind: Kafka
   spec:
     entityOperator:
       topicOperator:
         watchedNamespace: [***TOPIC NAMESPACE ***]
         reconciliationIntervalMs: 60000
         resources:
           requests:
             cpu: "1"
             memory: 500Mi
           limits:
             cpu: "1"
             memory: 500Mi
   ```

   If you want to deploy the Topic Operator with default configuration, add an empty object ({}).

   ```
   #...
   kind: Kafka
   spec:
     entityOperator:
       topicOperator: {}
   ```

2. Create or update your Kafka resource.

   ```
   kubectl apply --filename [***YAML CONFIG***] --namespace [***NAMESPACE***]
   ```

3. Verify the status of the deployment.

   ```
   kubectl get pods --namespace [***NAMESPACE***]
   ```

   If deployment is successful, you should see an Entity Operator pod in the output.

   ```
   NAME                                          READY   STATUS    RESTARTS
   #...
   my-cluster-entity-operator-67947ff779-k5sbv   2/2     Running      0
   ```

   The READY column shows the number of replicas that are ready/expected. Deployment is successful when the STATUS displays as Running.

   Note: The Topic Operator is running in a container within the Entity Operator pod.

What to do next

Create and manage Kafka topics with `KafkaTopic` resources. See Managing topics.

Deploying and configuring the Strimzi User Operator

You deploy and configure the Strimzi User Operator by configuring the entityOperator property in your Kafka resource to include userOperator properties. Deploying the User Operator is required if you want to manage Kafka users with KafkaUser resources instead of the KafkaAdmin API.

About this task

The User Operator enables you to manage Kafka users (clients) with `KafkaUser` resources. In Cloudera Streams Messaging - Kubernetes Operator you deploy the User Operator through the Strimzi Entity Operator. The Entity and User Operator are both deployed by the Strimzi Cluster Operator.

To deploy the User Operator, you configure the entityOperator property in your `Kafka` resource to include userOperator properties. You configure the User Operator by specifying additional sub-properties in the userOperator property.

By default, the User Operator watches `KafkaUser` resources in the namespace of the Kafka cluster deployed by the Cluster Operator. You can also specify a namespace to watch using the watchedNamespace property. A single User Operator can watch a single namespace. One namespace should be watched by only one User Operator.

Before you begin

- Strimzi must be installed in your cluster. The Strimzi Cluster Operator must be running. See Installation.
- For a full list of supported properties, see the EntityUserOperatorSpec schema reference in the Strimzi API Reference.

Procedure

1. Edit the entityOperator property in your Kafka resource to include userOperator properties.

   The following example configures the User Operator to watch a specified namespace. Additionally, it configures the reconciliation interval as well as various resource properties.

   ```
   #...
   kind: Kafka
   spec:
     entityOperator:
       userOperator:
         watchedNamespace: [***USER NAMESPACE***]
         reconciliationIntervalMs: 60000
         resources:
           requests:
             cpu: "1"
             memory: 500Mi
           limits:
             cpu: "1"
             memory: 500Mi
   ```

   If you want to deploy the User Operator with default configuration, add an empty object ({}).

   ```
   #...
   kind: Kafka
   spec:
     entityOperator:
       userOperator: {}
   ```

2. Create or update your Kafka resource.

   ```
   kubectl apply --filename [***YAML CONFIG***] --namespace [***NAMESPACE***]
   ```

3. Verify the status of the deployment.

   ```
   kubectl get pods --namespace [***NAMESPACE***]
   ```

   If deployment is successful, you should see an Entity Operator pod in the output.

   ```
   NAME                                          READY   STATUS    RESTARTS
   #...
   ```

```
my-cluster-entity-operator-67947ff779-k5sbv   2/2     Running      0
```

The READY column shows the number of replicas that are ready/expected. Deployment is successful when the STATUS displays as Running.

Note: The User Operator is running in a container within the Entity Operator pod.

What to do next

Create and manage Kafka users with `KafkaUser` resources. See User management.

Configuring Kafka brokers

Learn how you can update Kafka broker properties in your Kafka resource. Additionally, learn which broker properties are configurable and which are managed by Strimzi.

Related Information

Broker Configs | Apache Kafka

Updating broker configuration

You update broker configuration by editing your Kafka and KafkaNodePool resources.

About this task

You can update your `Kafka` and `KafkaNodePool` resource with kubectl edit. Which resource you update depends on what exact broker configurations you want to change.

Most broker configuration properties are specified in your `Kafka` resource. For example, properties like the default replication factor (default.replication.factor), minimum in sync replicas (min.insync.replicas), as well as many others. The `KafkaNodePool` resource contains configuration related to replicas, roles, and storage. Additionally, it can contain configuration related to CPU and memory resources, JVM options, as well as templates.

Procedure

1. Edit your resource.

   ```
   kubectl edit [***RESOURCE***] --namespace [***NAMESPACE***]
   ```

   Running kubectl edit opens the resource manifest in an editor.
2. Make your changes.
3. Save the file.

Results

Once the changes are saved, a rolling update is triggered and the brokers restart one after the other with the applied changes.

Note: The Strimzi Cluster Operator supports dynamic updates for broker configuration properties. Properties that support dynamic updates are updated without restarting the brokers

Configurable broker properties and exceptions

Learn which Kafka broker properties you can configure in the Kafka resource and which are managed by Strimzi.

Kafka broker properties are configured by adding them to spec.kafka.config in your `Kafka` resource. The values can be on of the following JSON types:

- String
- Number
- Boolean

You can find full reference of the available broker properties in the Apache Kafka documentation. While all properties can be specified, some properties are managed by Strimzi. Broker properties managed by Strimzi generally cannot be configured, however, there are a few exceptions.

If spec.kafka.config contains a broker property that cannot be changed, it is disregarded, and a warning message is logged to the Strimzi Cluster Operator log. All other supported properties are forwarded to Kafka.

Properties managed by Strimzi

Strimzi takes care of configuring and managing options related to the following.

- Security (encryption, authentication, and authorization)
- Listener configuration
- Broker ID configuration
- Configuration of log data directories
- Inter-broker communication
- ZooKeeper connectivity

This means that the properties with the following prefixes cannot be set.

- controller
- cruise.control.metrics.reporter.bootstrap.
- cruise.control.metrics.topic
- host.name
- inter.broker.listener.name
- listener.
- listeners.
- log.dir
- password.
- port
- process.roles
- sasl.
- security.
- servers,node.id
- ssl.
- super.user
- zookeeper.clientCnxnSocket
- zookeeper.connect
- zookeeper.set.acl
- zookeeper.ssl

Exceptions

There are a few exceptions within the list of broker properties managed by Strimzi. These properties are forwarded to Kafka rather than being disregarded. The properties are as follows:

- Any ssl configuration for supported TLS versions and cipher suites
- Configuration for the zookeeper.connection.timeout.ms property to set the maximum time allowed for establishing a ZooKeeper connection.

- The following Cruise Control metrics properties:
  - cruise.control.metrics.topic.num.partitions
  - cruise.control.metrics.topic.replication.factor
  - cruise.control.metrics.topic.retention.ms
  - cruise.control.metrics.topic.auto.create.retries
  - cruise.control.metrics.topic.auto.create.timeout.ms
  - cruise.control.metrics.topic.min.insync.replicas
- The following controller properties:
  - controller.quorum.election.backoff.max.ms
  - controller.quorum.election.timeout.ms
  - controller.quorum.fetch.timeout.ms

Related Information

KafkaClusterSpec schema reference | Strimzi API Reference
KafkaNodePool schmea reference | Strimzi API Reference
Supported TLS versions and cipher suites | Strimzi

Storage configuration

Learn about storage configuration, available storage types, and storage configuration recommendations for Kafka and ZooKeeper in Cloudera Streams Messaging - Kubernetes Operator.

Warning: You cannot change the storage type following cluster deployment.

Kafka and Zookeeper storage is configured in separate resources. Kafka storage is configured in the `KafkaNodePool` resource using the spec.storage property. ZooKeeper Storage is configured in the `Kafka` resource using the spec.zookeeper.storage property.

For Kafka storage

```
#...
kind: KafkaNodePool
spec:
  storage:
    type: persistent-claim
    size: 100Gi
    deleteClaim: true
```

This configuration snippet defines a 100 GB persistent storage with the default storage class for Kafka in a `KafkaNodePool` resource. The deleteClaim property specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.

For ZooKeeper storage

```
#...
kind: Kafka
spec:
  zookeeper:
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
```

This configuration snippet defines a 100 GB persistent storage with the default storage class for ZooKeeper in a `Kafka` resource. The deleteClaim property specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.

Cloudera Streams Messaging - Kubernetes Operator supports multiple types of storage depending on the platform. The supported storage types are as follows:

- Ephemeral
- Persistent
- JBOD (Just a Bunch of Disks) – Kafka brokers only

The storage type is configured with storage.type. The property accepts three values, ephemeral, persistent-claim, and jbod. Each value corresponds to its respective storage type. JBOD (jbod) is only supported for Kafka. JBOD is not supported for ZooKeeper clusters.

The following sections provide a more in-depth look at each storage type, and collect Cloudera recommendations on storage.

Ephemeral storage

Learn about ephemeral storage.

When using ephemeral storage, data is only retained as long as the pod that uses it is running and it is lost when the pod is deleted. Ephemeral storage can be used for both Kafka brokers and ZooKeeper servers. Since this storage type does not preserve your data on the long run, this is not recommended and should only be used for development and test clusters.

To use ephemeral storage, set storage.type to ephemeral.

For Kafka storage

```
#...
kind: KafkaNodePool
spec:
  storage:
    type: ephemeral
```

For ZooKeeper storage

```
#...
kind: Kafka
spec:
  zookeeper:
    storage:
      type: ephemeral
```

The available configuration options are listed in the Strimzi documentation.

Related Information

EphemeralStorage schema reference | Strimzi API reference

Persistent storage

Learn about persistent storage, which is the storage type recommended by Cloudera for Kafka and ZooKeeper clusters.

When using persistent storage, data is retained even in case of a system disruption. Because of this, persistent storage is the storage type recommended by Cloudera for production environments. When using this configuration, a single persistent storage volume is defined. Persistent storage can be used for both Kafka brokers and ZooKeeper servers.

To use persistent storage, set storage.type to persistent-claim.

Note: Persistent volumes used by the Kafka and ZooKeeper servers may have an effect on the scheduling of their pods if their node affinity is set.

For Kafka storage

```
#...
kind: KafkaNodePool
spec:
  storage:
    type: persistent-claim
```

For ZooKeeper storage

```
#...
kind: Kafka
spec:
  zookeeper:
    storage:
      type: persistent-claim
```

Custom storage classes

Storage classes define storage profiles and dynamically provision persistent volumes based on that profile. If there is no default storage class, or you would not like to use the default, you can specify your storage class by setting storage.class.

Tip: For Kafka brokers, Cloudera recommends a `StorageClass` that has volume expansion enabled (allo wvolumeexpansion set to true).

For Kafka storage

```
#...
kind: KafkaNodePool
spec:
  storage:
    type: persistent-claim
    class: custom-storage-class
```

For ZooKeeper storage

```
#...
kind: Kafka
spec:
  zookeeper:
    storage:
      type: persistent-claim
      class: custom-storage-class
```

These examples configure a custom storage class for the pods in the cluster which it is configured for. Custom storage classes can be configured on a more granular level as well with storage overrides.

Storage overrides

Persistent volumes can be configured on a per-broker and ZooKeeper server basis by specifying the Kubernetes storage class for each volume with storage overrides. Specifying storage overrides can be used to influence the storage parameters and pod scheduling constraints of each broker and ZooKeeper server.

Note: The overrides.broker property is used in both Kafka and ZooKeeper configurations. In the case of ZooKeeper, the broker property represents the ZooKeeper server instance.

For Kafka storage

```
#...
kind: KafkaNodePool
spec:
  storage:
    type: persistent-claim
    overrides:
      - broker: 0
        class: storageclass1
      - broker: 1
        class: storageclass2
```

For ZooKeeper storage

```
#...
kind: Kafka
spec:
  zookeeper:
    storage:
      type: persistent-claim
      overrides:
        - broker: 0
          class: storageclass1
        - broker: 1
          class: storageclass2
```

The available configuration options for persistent storage are listed in the Strimzi documentation.

Related Information

Pod scheduling

PersistentStorage schema reference | Strimzi API reference

Storage Classes | Kubernetes

Node Affinity | Kubernetes

JBOD storage

Just a bunch of disks (JBOD) refers to a system configuration where disks are used independently rather than organizing them into redundant arrays. Learn how you can configure JBOD storage for Kafka.

JBOD storage allows you to configure your Kafka cluster to use multiple volumes. This approach provides increased data storage capacity for Kafka nodes, and can lead to performance improvements. A JBOD configuration is defined by one or more volumes, each of which can be either ephemeral or persistent. JBOD is only applicable to the Kafka storage in the `KafkaNodePool` resource.

To use JBOD storage, set the storage.type to jbod and specify the volumes.

```
#...
kind: KafkaNodePool
```

```
spec:
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
```

This example uses a jbod storage type with two attached persistent volumes. The volumes must all be identified by a unique ID.

You can always increase or decrease the number of disks or increase the volume size by modifying the `KafkaNodePool` resource and reapplying the changes. However, you cannot change the IDs once volumes are created.

The available configuration options are listed in the Strimzi documentation.

Related Information

JbodStorage schema reference | Strimzi API reference

Storage recommendations

Cloudera recommends using persistent storage to store Kafka and ZooKeeper data. Ephemeral storage is only suitable for short-lived test clusters. Use a dynamic provisioner storage class with block storage (ReadWriteOnce access) and prefer SSD or NVMe disks.

Consider the following when using persistent storage.

Local storage

Using local storage makes the deployment similar to a bare-metal deployment in terms of scheduling and availability. It provides good throughput as both Kafka and ZooKeeper storage operations have less overhead when replication and network hops are not necessary.

However, the Kafka and ZooKeeper pods become bound to the node where the backing volume is located. This means that the pods cannot be scheduled to a different node, which impacts availability

Distributed storage

Using distributed storage with synchronous replication allows leveraging the flexibility of Kubernetes pod scheduling. Both Kafka and ZooKeeper pods can be migrated across nodes due to the availability of the same storage on different nodes. This improves the availability of the Kafka cluster. Node failures do not bring down Kafka brokers and ZooKeeper servers permanently.

However, distributed storage reduces throughput in the Kafka cluster. The synchronous replication of storage adds extra overhead to disk writes. Additionally, if the backing storage class does not support data locality, reads and writes require extra network hops.

Pod scheduling

Learn about the default affinity rules and tolerations that Strimzi sets for pod scheduling. Additionally, learn what affinity rules Cloudera recommends for making pod scheduling stricter.

The scheduling of Kafka broker, KRaft controller, and ZooKeeper pods can be customized in the `Kafka` and `KafkaNodePool` resources through various configurations such as storage configurations, affinity rules, and tolerations. Strimzi by default only sets a few of the pod scheduling configurations. It is your responsibility to ensure that pod scheduling configurations are customized correctly for your environment and use case.

Both storage and rack awareness configuration might have an impact on pod scheduling. For storage, depending on the configuration, it is possible that a pod is bound to a node or a group of nodes and cannot be scheduled elsewhere.

If rack awareness is configured, your pods by default get preferred and required affinity rules, which influence pod scheduling.

Related Information

Storage recommendations

Rack awareness

Default tolerations

The Strimzi Cluster Operator does not set any tolerations on the Kafka broker, KRaft controller, and ZooKeeper pods by default. The pods get a default toleration from the Kubernetes platform.

The default tolerations are as follows.

```
#...
kind: Kafka
spec:
  kafka:
    template:
      pod:
        tolerations:
          - effect: NoExecute
            key: node.kubernetes.io/not-ready
            operator: Exists
            tolerationSeconds: 300
          - effect: NoExecute
            key: node.kubernetes.io/unreachable
            operator: Exists
            tolerationSeconds: 300
```

This means that whenever the Kubernetes node running the pod is tainted as unreachable or not-ready, the pod should be terminated after five minutes. This means that even if you lose an entire Kubernetes node, the pod will be terminated and rescheduled only after five minutes.

Depending on your platform and the type of failure of a Kubernetes worker node, it is possible that the pods will not be rescheduled from a dead worker node and the pod will stay in terminating state forever. In this case manual intervention is needed to move forward.

Related Information

Taints and Tolerations | Kubernetes

Node Shutdowns | Kubernetes

Pod scheduling recommendations

Learn about the pod scheduling configurations recommended by Cloudera.

Tolerations

Instead of using the default tolerations with 300 seconds, you can consider setting tolerations with smaller timeouts if a five minute downtime of Kafka brokers, KRaft controllers or ZooKeeper nodes is not acceptable for you.

For Kafka brokers it is possible to set tolerations globally using spec.kafka.template.pod.tolerations in the `Kafka` resource. Alternatively, you can set tolerations for a group of broker nodes only using spec.template.pod.tolerations in the `KafkaNodePool` resource.

For KRaft controllers, configuration of the tolerations is the same as for Kafka brokers. You can set tolerations globally using spec.kafka.template.pod.tolerations in the `Kafka` resource. Alternatively, you can set tolerations for a group of controller nodes only using spec.template.pod.tolerations in the `KafkaNodePool` resource.

For ZooKeeper it is only possible to set tolerations globally in spec.zookeeper.template.pod.tolerations in the `Kafka` resource.

Other affinity rules

You can use required and preferred rules to fine tune scheduling according to your needs.

If you use required rules, it is your platform's responsibility to always have enough resources (for example, enough nodes) to satisfy the rules. Otherwise, the scheduler will not be able to schedule pods and they will be in a pending state.

If you use preferred rules with any weight, ensure that the rule weight is correctly set. The scheduler will consider the rules with higher weight more important than others with lower weight.

Note: Kubernetes will still run the pod even if it has to break a preferred rule.

For Kafka brokers it is possible to set affinity rules globally using spec.kafka.template.pod.affinity in the `Kafka` resource. Alternatively, you can set affinity rules for a group of broker nodes only using spec.template.pod.affinity in the `KafkaNodePool` resource.

For KRaft controllers, configuration of affinity rules is the same as for Kafka brokers. You can set affinity rules globally using spec.kafka.template.pod.affinity in the `Kafka` resource. Alternatively, you can set affinity rules for a group of controller nodes only using spec.template.pod.affinity in the `KafkaNodePool` resource.

For ZooKeeper it is only possible to set affinity rules globally in spec.zookeeper.template.pod.affinity in the `Kafka` resource.

The following collects a number of example required rules for typical use cases.

Run each Kafka broker pod on different nodes

```
#...
kind: KafkaNodePool
spec:
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/cluster
                    operator: In
                    values:
                      - [***CLUSTER NAME***]
                  - key: strimzi.io/broker-role
                    operator: In
                    values:
                      - "true"
              topologyKey: kubernetes.io/hostname
```

Run each KRaft controller pod on different nodes

```
#...
```

```
kind: KafkaNodePool
spec:
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/cluster
                    operator: In
                    values:
                      - [***CLUSTER NAME***]
                  - key: strimzi.io/controller-role
                    operator: In
                    values:
                      - "true"
              topologyKey: kubernetes.io/hostname
```

Run each Zookeeper pod on different nodes

```
#...
kind: Kafka
spec:
  zookeeper:
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/component-type
                      operator: In
                      values:
                        - zookeeper
                    - key: strimzi.io/cluster
                      operator: In
                      values:
                        - [***CLUSTER NAME***]
                topologyKey: kubernetes.io/hostname
```

Run ZooKeeper and Kafka broker pods on different nodes

```
#...
kind: Kafka
spec:
  kafka:
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/cluster
                      operator: In
                      values:
                        - [***CLUSTER NAME***]
                topologyKey: kubernetes.io/hostname
  zookeeper:
    template:
      pod:
```

```
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/cluster
                      operator: In
                      values:
                        - [***CLUSTER NAME***]
                topologyKey: kubernetes.io/hostname
```

Run KRaft controller and Kafka broker pods on different nodes

```
#...
kind: Kafka
spec:
  kafka:
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: strimzi.io/cluster
                  operator: In
                  values:
                  - [***CLUSTER NAME***]
              topologyKey: kubernetes.io/hostname
```

Rack awareness

Racks provide information about the physical location of a broker or a client. A Kafka cluster can be made rack aware by configuring rack awareness for the Kafka brokers, consumers, and ZooKeeper servers. Enabling rack awareness can help in hardening your cluster, it provides durability guarantees, and significantly decreases the chances of data loss.

To enable rack awareness for a Kafka cluster running in Kubernetes with Cloudera Streams Messaging - Kubernetes Operator you complete the following tasks.

1. Configure rack information for your Kubernetes nodes using labels.
2. Configure rack awareness for both Kafka and ZooKeeper clusters.
3. Configure follower fetching for both Kafka brokers and consumers.

Note: Although the feature is called rack awareness, the term rack does not necessarily mean an actual physical server rack. Instead, a rack from Kafka's perspective represents any physical location or independent physical infrastructure like data centers, regions, zones, and so on.

Configuring rack information on Kubernetes nodes

Before you can enable rack awareness for Kafka or ZooKeeper, you must ensure that a label is configured in your Kubernetes cluster that holds rack information. You configure labels with kubectl label.

About this task

Kubernetes nodes can hold their respective rack information in labels. You can set any labels to store your rack information, however, Cloudera recommends using the topology.kubernetes.io/zone label. This is because it is a

well-known Kubernetes label and cloud providers typically set this label for you automatically. If your (cloud) environment provider does not automatically set this label in your environment, you have to set it manually. This is done with kubectl label.

Procedure

1. Set your chosen label with kubectl label.

```
kubectl label node [***NODE NAME***] topology.kubernetes.io/zone=[***ZONE/
RACK***]
```

   Repeat this step for each of your nodes. For example, assuming you have six nodes, three different racks, and two nodes per rack, you would run commands similar to the following.

```
kubectl label node kubernetes-m02 topology.kubernetes.io/zone=eu-zone-1
kubectl label node kubernetes-m03 topology.kubernetes.io/zone=eu-zone-1
kubectl label node kubernetes-m04 topology.kubernetes.io/zone=eu-zone-2
kubectl label node kubernetes-m05 topology.kubernetes.io/zone=eu-zone-2
kubectl label node kubernetes-m06 topology.kubernetes.io/zone=eu-zone-3
kubectl label node kubernetes-m07 topology.kubernetes.io/zone=eu-zone-3
```

2. Verify your configuration.

```
kubectl get node -o=custom-columns=NODE:.metadata.name,ZONE:.metadata.la
bels."topology\.kubernetes\.io/zone" | sort -k2
```

   The output lists your nodes and their rack information (zone). Output will be similar to the following example.

```
NODE                ZONE
kubernetes-m01       <none>
kubernetes-m02   eu-zone-1
kubernetes-m03   eu-zone-1
kubernetes-m04   eu-zone-2
kubernetes-m05   eu-zone-2
kubernetes-m06   eu-zone-3
kubernetes-m07   eu-zone-3
```

Note: Rack information for the control-plane node (kubernetes-m01) is not set in this example, because it should not function as a workload node.

Related Information

toplogy.kubernetes.io/zone | Kubernetes

Configuring rack awareness for ZooKeeper

ZooKeeper rack awareness is configured in the Kafka resource by specifying affinity rules.

Zookeeper rack awareness can only be configured through the `Kafka` resource. As a result, you can only set configuration that applies for all ZooKeeper instances.

To configure rack awareness for ZooKeeper, Cloudera recommends setting the following two affinity rules for Zookeeper in the `Kafka` resource.

```
#...
kind: Kafka
spec:
 zookeeper:
   template:
     pod:
```

```
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: Exists
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - podAffinityTerm:
                labelSelector:
                  matchLabels:
                    strimzi.io/cluster: [***CLUSTER NAME***]
                    strimzi.io/name: [***CLUSTER NAME***]-ZOOKEEPER
                topologyKey: topology.kubernetes.io/zone
              weight: 100
```

These rules are proper for most cases, but it is still possible that ZooKeeper pods are scheduled to another node in a different rack after a node failure. This is because these rules do not force keeping the ZooKeeper pods in a specific rack.

Currently, the only way to enforce ZooKeeper instances to stick to specific racks is to use storage overrides with your own storage classes and volume node affinities. If a pod has a persistent volume claim on a volume with node affinity set, the scheduler considers the restrictions on the volume in use when scheduling the pod. This way, you can configure a rack-aware cluster without the limitations mentioned above.

Related Information

Storage overrides

Configuring rack awareness for Kafka brokers

Rack awareness for Kafka is configured in your Kafka resource by specifying the Kubernetes node label that holds rack information. Optionally, you can configure nodeAffinity rules in the KafkaNodePool resource for stricter broker placement.

About this task

Kafka brokers are made rack-aware by configuring the broker.rack property. When broker racks are configured, Kafka intentionally places replicas of the same partition (whenever a topic is created, modified, and so on) into different racks to protect the data from rack failures.

In Cloudera Streams Messaging - Kubernetes Operator, you do not set broker.rack directly in your `Kafka` resource to configure rack awareness. Instead, you specify which node label to use as rack information by configuring the kafk a.rack.topologyKey property in the `Kafka` resource.

If kafka.rack.topologyKey is set, the broker.rack property of each broker is automatically set based on the node label value that the broker pod is scheduled to. Additionally, the broker pods automatically get an affinity and anti-affinity rule. These rules guarantee best effort spreading of brokers between racks, but do not force having the same broker always in the same rack.

Because the default rules only guarantee best effort spreading, Cloudera recommends that you override these rules with stricter rules explicitly configuring which group of nodes should be placed in which racks.

The following steps demonstrate how to configure kafka.rack.topologyKey and demonstrate what rules you have to set in the `KafkaNodePool` resource if you want to ensure that a group of nodes are always placed in the same rack.

Note: By default, KRaft controllers also get the same affinity rules as Kafka brokers. Rules are applied if kafka.rack.topologyKey is set in the `Kafka` resource. The broker.rack property is not used by KRaft controllers, because all controllers hold the same data.

Before you begin

- Ensure that you chose and configured a label that holds rack information. See Configuring rack information on Kubernetes nodes on page 30.
- The default affinity rules are documented in Default affinity rules for rack awareness on page 35.

Procedure

1. Configure kafka.rack.topologyKey in your `Kafka` resource.

```
#...
kind: Kafka
spec:
  kafka:
    rack:
      topologyKey: topology.kubernetes.io/zone
```

2. Explicitly configure which group of nodes are placed in which rack.

   This can be done by adding a required nodeAffinity rule in your `KafkaNodePool` resources. This step is marked as optional but is recommended by Cloudera. The following examples demonstrate a configuration where there are two node pools. The nodes in each pool are assigned to separate racks (zones).

   For First pool

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: first-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: In
                    values:
                      - eu-zone-1
```

   For Second pool

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: second-pool
  labels:
    strimzi.io/cluster: my-cluster
spec:
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: In
                    values:
```

```
                        - eu-zone-2
```

Results
After the changes are applied, a rolling restart is initiated.

What to do next
After the cluster is restarted, check the broker.rack values of each broker. You can get the broker.rack values of multiple brokers that are in the same pool with the following command.

```
for broker in [***CLUSTER NAME***]-[***POOL NAME***]-[***ID RANGE***]; do
  kubectl exec -namespace [***NAMESPACE***] -it \
  $broker --container kafka \
  -- /bin/bash -c "cat /tmp/strimzi.properties" \
  | grep -E "broker.id|broker.rack" && echo "\n"
done
```

- *[***CLUSTER NAME***]* is the name of your cluster.
- *[***POOL NAME***]* is the name of the node pool.
- *[***ID RANGE***]* is a range of broker IDs enclosed in curly braces ({}). For example, {1..3}.

This command will output the broker IDs and the rack information set for each broker. For example:

```
broker.id=0
broker.rack=eu-zone-1


broker.id=1
broker.rack=eu-zone-1


broker.id=2
broker.rack=eu-zone-1
```

Configuring follower fetching

You enable follower fetching by configuring your Kafka resource and specifying rack information in your Kafka clients.

About this task

If rack awareness is enabled for Kafka brokers, consumers by default continue to consume messages from partition leaders. This behavior remains the same even if the consumer and the partition leader are located in different racks.

It is possible (especially in cloud environments) that a consumer application is in a different region than the partition leader, but there is a partition follower in the same region as the consumer application. In this case it is better to consume from the partition follower instead. This way you can avoid unnecessary traffic across data centers, reducing costs and application latency. This is called follower fetching.

Follower fetching is enabled by configuring the replica selector implementation in your Kafka resource to be rack-aware. Additionally, you need to configure the client.rack property of your clients.

Procedure

1. Update your `Kafka` resource.

   To enable follower fetching, set the replica.selector.class broker property to the RackAwareReplicaSelector.

   ```
   #...
   ```

```
kind: Kafka
spec:
  kafka:
    rack:
      topologyKey: topology.kubernetes.io/zone
    config:
      replica.selector.class: org.apache.kafka.common.replica.RackAware
ReplicaSelector
```

2. Wait until the rolling restart finishes.

   Use the following command to monitor cluster state.

   ```
   kubectl get pods --namespace [***NAMESPACE***] --output wide --watch
   ```

3. Configure your consumers.

   ```
   client.rack=[***RACK ID***]
   ```

   The *[***RACK ID***]* is one of the rack IDs (zones) that you configured in the topology.kubernetes.io/zone label. The client reads from a follower replica if a follower replica host broker has a broker.rack value that is identical with the value of client.rack on the client side. If there isn't one, the client fetches data from the leader.

Default affinity rules for rack awareness

Kafka broker pods automatically get the following affinity and anti-affinity rules when rack awareness is enabled.

Affinity rule

This is a required rule, the scheduler will only schedule a broker pod to a node, if the node has the configured label set.

```
template:
  pod:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: topology.kubernetes.io/zone
                  operator: Exists
```

Anti-affinity rule

This is a preferred rule, it spreads Kafka brokers evenly across racks in a best-effort manner.

```
template:
  pod:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchLabels:
                  strimzi.io/cluster: [***CLUSTER NAME***]
                  strimzi.io/name: [***CLUSTER NAME***]-kafka
              topologyKey: topology.kubernetes.io/zone
            weight: 100
```

Configuring Kafka broker node IDs

Learn how you can configure Kafka brokers to get IDs from a specified range.

It might be important to specify the ID range of the Kafka brokers to avoid confusion before creating the cluster. This can be configured on the level of the Kafka node pools. Your chosen range is configured using an annotation in the `KafkaNodePool` resource.

```
#...
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/next-node-ids: "[0-99]"
```

In this example, a range from 0 to 99 is configured. The desired range can be provided by ranges, individual numbers, and so on. The range can also be provided in a reversed order, in that case the IDs are assigned in reversed order if possible.

Configuring Kafka for Prometheus monitoring

To monitor your Kafka cluster with Prometheus, you must configure your Kafka cluster to expose the necessary metric endpoints that integrate with your Prometheus deployment. This is done by configuring metricsConfig properties for components in your Kafka resource.

About this task

By default cluster components deployed with your `Kafka` resource do not expose metrics that Prometheus can scrape. In order to use Prometheus to monitor your Kafka cluster, you must enable and expose these metrics. This is done by adding a metricsConfig property to the spec of each cluster component in your Kafka resource.

Specifying metricsConfig in the Kafka resource enables the Prometheus JMX Exporter which exposes metrics through a HTTP endpoint. The metrics are exposed on port 9094. The metricsConfig property can reference a ConfigMap that holds your JMX metrics configuration or will include the metrics configurations in-line. The following steps demonstrate the configuration by referencing a ConfigMap.

Before you begin

A Prometheus deployment that can connect to the metric endpoints of the Kafka cluster running in the Kubernetes environment is required. Any properly configured Prometheus deployment can be used to monitor Kafka. You can find additional information and examples on Prometheus setup in the Strimzi documentation.

Procedure

1. Create a ConfigMap with JMX metrics configuration for both Kafka and ZooKeeper.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  kafka-metrics-config.yml: |
```

```
    [***KAFKA METRICS CONFIGURATION***]
  zookeeper-metrics-config.yml: |
    [***ZOOKEEPER METRICS CONFIGURATION***]
```

Replace *[***KAFKA METRICS CONFIGURATION***]* and *[***ZOOKEEPER METRICS CONFIGURATION***]* with your JMX Prometheus metrics configurations.

2. Update your `Kafka` resource with metricsConfig property.

   Add metricsConfig to the spec of both Kafka and ZooKeeper. The property needs to reference the ConfigMap you created in Step 1.

```
#...
kind: Kafka
spec:
  kafka:
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: kafka-metrics
          key: kafka-metrics-config.yml
  zookeeper:
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: kafka-metrics
          key: zookeeper-metrics-config.yml
```

What to do next

- Configure Prometheus and specify alert rules to start scraping metrics from the ZooKeeper and Kafka pods. You can find an example rules file (prometheus-rules.yaml) as well as various other configuration examples on the Cloudera Archive. Examples related to Prometheus are located in the /csm-operator/1.3/examples/metrics directory.
- Review Cloudera recommendations on what alerts and metrics to configure. See Monitoring with Prometheus.

Related Information

Cloudera Archive

Prometheus JMX Exporter | GitHub

Configuring logging for Kafka cluster components

Learn how to configure logging for Kafka cluster components. You can configure logging for these components directly in the Kafka resource, or by referencing a ConfigMap.

The logging properties of Kafka cluster components like Kafka brokers, ZooKeeper, Cruise Control, and all other components deployed and managed through the `Kafka` resource are configured in the `Kafka` resource.

Logging properties are specified in spec.*[***COMPONENT***]*.logging. Logging properties can be added directly to this property, or can be defined in an external ConfigMap that is referenced in the `Kafka` using configMapKeyRef property.

You choose the configuration method by setting the logging.type property to either inline or external.

Inline

Inline configuration means that you directly specify the logging properties in the Kafka resource at the spec of each component.

```
#...
kind: Kafka
spec:
  #...
    logging:
      type: inline
      loggers:
        kafka.root.logger.level: INFO
```

External

External configuration means that you reference your own ConfigMap that holds the logging properties.

```
#...
kind: Kafka
spec:
  #...
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-config-map-key
```

A ConfigMap is generated for each Kafka cluster component after pod creation. These ConfigMaps contain the actual logging configuration. Do not edit the generated ConfigMaps directly, as direct changes are ignored.

Depending on the changes made, they are either applied dynamically, or a rolling restart is triggered.

The following Kafka cluster components use log4j configuration:

- Kafka
- ZooKeeper

The following Kafka cluster components use log4j2 configuration:

- CruiseControl
- UserOperator
- EntityOperator

Related Information

Logging options for Kafka components and operators | Strimzi

Listener configuration

Client access to your cluster is set up in Cloudera Streams Messaging - Kubernetes Operator by configuring listeners in your Kafka resource. Listeners can be used to expose your brokers, allowing clients to access them.

Each listener is configured as an array in your `Kafka` resource. For example:

```
#...
kind: Kafka
spec:
  kafka:
    version: 3.9.0.1.3
    replicas: 3
    listeners:
```

```
      - name: plain
        port: 9092
        type: internal
        tls: false
```

You can configure any number of listeners as long as their names and ports are unique. Their configuration is also highly customizable. For an exhaustive list of accepted properties, see the `GenericKafkaListener` as well as other listener schema references in the Strimzi API reference.

Listener categories and types

There are two categories of listeners, internal and external. Internal listeners are used to expose Kafka to clients that are internal to the Kubernetes cluster. External listeners provide a way to expose Kafka to the outside world.

Listeners are further categorized by their type. The different listener types expose Kafka with different connection mechanisms. The types of listeners available are as follows.

Internal listener types

- internal

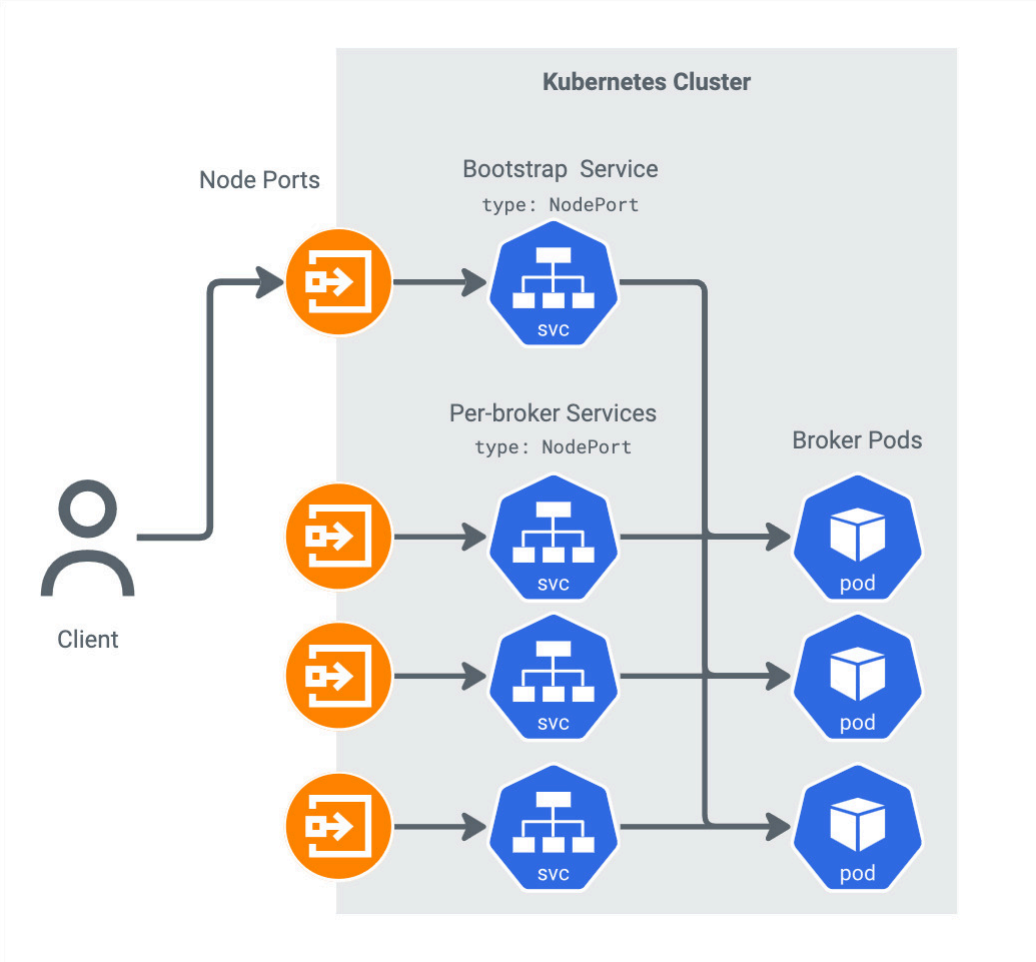
  An internal type listener uses a Kubernetes headless Service that gives each broker pod a stable hostname. These hostnames are set as advertised listeners for Kafka. In addition, a ClusterIp Kubernetes Service is set up that acts as the Kafka bootstrap. The initial connection is done using the bootstrap, subsequent connections are opened using the hostnames given to the pods by the headless Kubernetes Service.
- cluster-ip

  With a cluster-ip type listener, individual ClusterIP type Kubernetes services are set up for each broker. The hostnames of the ClusterIP services are configured as the advertised listeners for Kafka. In addition, another ClusterIP is provisioned that acts as the Kafka bootstrap. The initial connection is done using the bootstrap, subsequent connections are opened using the ClusterIP Services corresponding to each broker.

All Kafka resources that you create in Cloudera Streams Messaging - Kubernetes Operator most likely contain an internal listener by default. This means that you can test your cluster and connect your client as soon as the cluster is up and running. To connect a client, direct it to the address of the bootstrap service that was set up by the listener. From there Kubernetes and the Strimzi Cluster Operator handle everything else ensuring that connection requests are sent to the appropriate brokers.

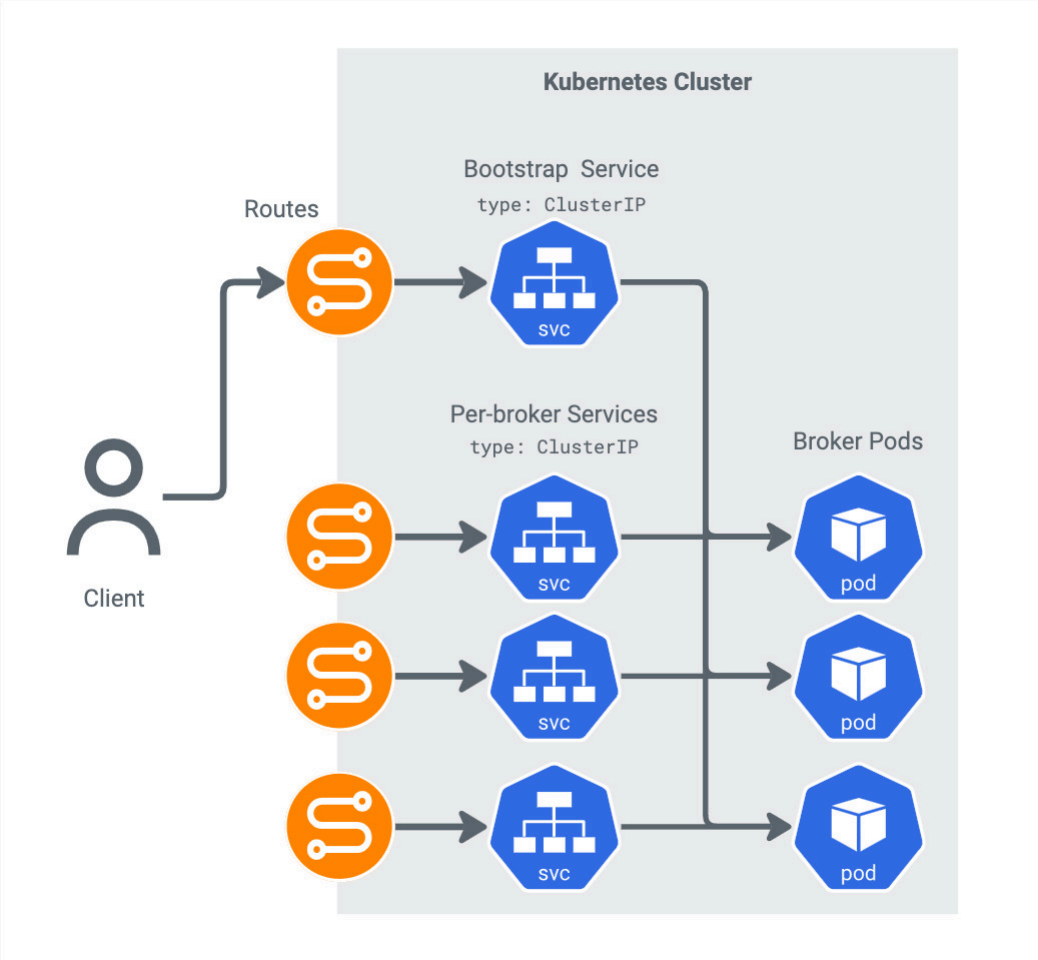
External listener types

- nodeport



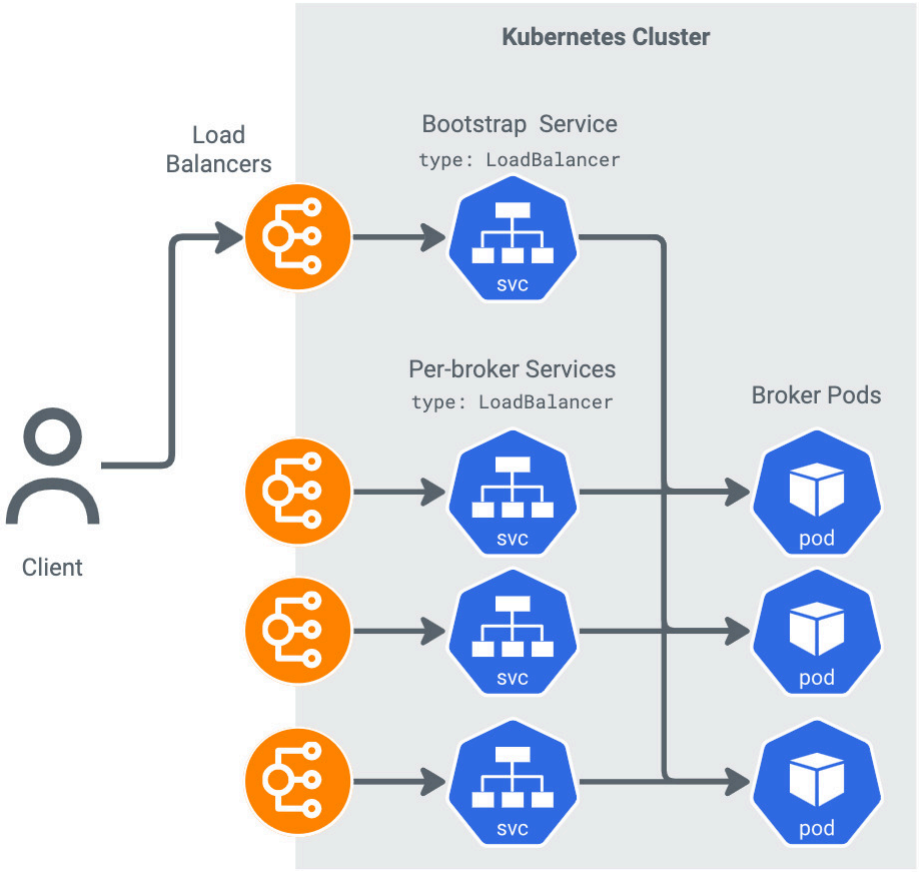
A nodeport type listener sets up NodePort type Kubernetes Services to provide external access to Kafka.

- route



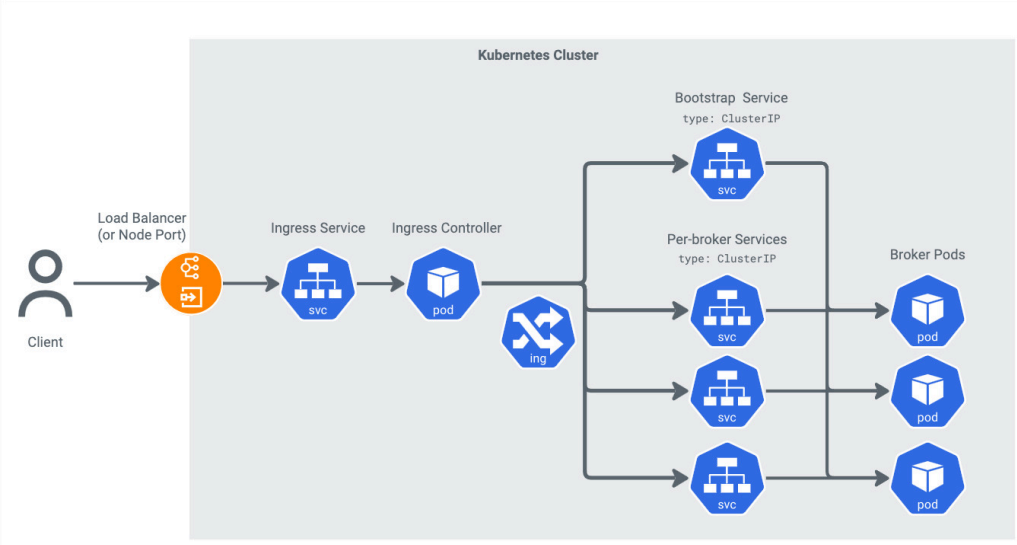
A route type listener uses OpenShift routes and the default HAProxy router to provide external access to Kafka.

- loadbalancer



A loadbalancer type listener sets up LoadBalancer type Kubernetes Services and cloud provider or infrastructure managed load balancers to provide external access to Kafka.

- ingress



An ingress type listener uses Kubernetes Ingress and the Ingress-NGINX controller to provide external access to Kafka.

Which of the available external listener types you choose will depend on your requirements and infrastructure. Each external listener type is further documented in their dedicated section. See these sections for more information on how they work as well as instructions on how to set them up.

Default listener ports

When configuring listeners for Kafka clients, you can use ports from 9092 and above. There are two default listeners configured in each cluster that can not be configured and accessed by external clients. These are as follows.

- Control plane listener (9090) - This port is used for controller communication
- Replication listener (9091) - This port is used by replica fetchers for replicating topic partitions

Related Information

Strimzi API Reference

NodePort

Learn about Kubernetes NodePorts and how NodePorts are used to provide Kafka clients access to your cluster.

NodePort is a Kubernetes Service type that allocates a port, referred to as a node port, on every node of the Kubernetes cluster. NodePort ensures that all traffic routed to the node port gets to a specific pod.

To set up external cluster access with NodePorts, you add nodeport type listeners to your Kafka resource (listener.type:nodeport).

Note: By default the node port numbers are assigned by Kubernetes from a configurable default range. Unless you choose to configure specific port numbers, new ports might be assigned when you redeploy the `Kafka` resource.

Once configuration is done, the Strimzi Cluster Operator deploys multiple NodePort Services. Specifically, you will have the following:

- One NodePort that serves as an external bootstrap. This is used by clients for the initial connection and to receive metadata (advertised listeners) from the Kafka cluster.
- A NodePort for each Kafka broker. These are used by clients to directly access the individual brokers.

The addresses of the nodes and the node ports are collected by the Strimzi Cluster Operator and configured as the advertised listeners of the brokers. So brokers are automatically configured to advertise the right address and ports. As a result, once listener setup is complete, you can connect your clients running outside of the Kubernetes network by directing them to the NodePort Service that acts as the external bootstrap. Kubernetes handles everything else and ensures that client requests are routed to the correct brokers.

Configuring nodeport listeners

Complete the following steps to set up and configure a nodeport type listener in your Kafka resource. The following steps also include an example on how to connect a Kafka console client to the cluster.

About this task

These steps demonstrate basic listener configuration with typical customizations. In addition to the configuration shown here, you can further customize your listener and enable or disable TLS encryption using the tls property, specify a client authentication mechanism with the authentication property, as well as add various additional configurations using the configuration property. For a comprehensive list of available properties, see the *GenericKafkaListener schema reference* in the Strimzi API reference.

Procedure

1. Configure your Kafka resource.

   Add an external listener that has its type property set to nodeport. In addition, Cloudera recommends that you customize your listeners and specify exact port numbers with the nodePort property. This way, you do not need to reconfigure your clients every time you redeploy Kafka.

   However, note that no validation is done, so you must ensure that the configured ports are not used by any other service and are within the range assigned for node ports. If port numbers are not specified, the Strimzi Cluster Operator chooses available ports from the range assigned to node ports.

   The following snippet shows a configuration where listener.type is set to nodeport and exact port numbers are also specified.

   ```
   #...
   kind: Kafka
   spec:
     kafka:
       listeners:
         - name: external
           port: 9094
           type: nodeport
           tls: true
           authentication:
             type: tls
           configuration:
             bootstrap:
               nodePort: 32000
             brokers:
               - broker: 0
                 nodePort: 32001
               - broker: 1
                 nodePort: 32002
               - broker: 2
                 nodePort: 32003
   ```

2. Verify that NodePort Services are created and running.

   ```
   kubectl get services --namespace [***NAMESPACE***]
   ```

   The output will be similar to the following example.

   ```
   NAME                                    TYPE        CLUSTER-IP      EXTERN
   AL-IP
   #...
   my-cluster-kafka-external-bootstrap   NodePort    10.43.137.124   <none>

   my-cluster-kafka-0                    NodePort    10.43.78.187    <none>

   my-cluster-kafka-1                    NodePort    10.43.5.207     <none>

   my-cluster-kafka-2                    NodePort    10.43.75.51     <none>
   ```

   Notice that there is a NodePort Service deployed for each Kafka broker. Additionally you have a separate external bootstrap NodePort called *[***CLUSTER NAME***]*-kafka-external bootstrap. Clients connecting to the Kafka cluster should be directed to the external bootstrap.

3. Get the node port of the external bootstrap service.

   ```
   kubectl get service [***CLUSTER NAME***]-kafka-external-bootstrap \
     --namespace [***NAMESPACE***] \
   ```

```
  --output=jsonpath='{.spec.ports[0].nodePort}{"\n"}'
```

4. Get the address (hostname or IP) of any node.

```
kubectl get node [***NODE NAME***] \
  --output=jsonpath='{range.status.addresses[*]}{.type}{"\t"}{.address}{"
\n"}'
```

5. Configure and run your client.

   The following example shows a Kafka console producer.

```
kafka-console-producer.sh \
  --bootstrap-server [***NODE ADDRESS***]:[***NODE PORT***] \
  --topic [***TOPIC***]
```

Results

A nodeport type listener is configured. External Kafka clients can now access your Kafka cluster through the NodePort Services.

Related Information

Service | Kubernetes

Accessing Kafka: Part 2 – Node ports | Strimzi blog

GenericKafkaListener schema reference | Strimzi API reference

Route

Routes is an OpenShift concept and solution that allows you to expose Kubernetes Services at a public URL so that external clients can reach your applications running in the Kubernetes cluster.

To set up external cluster access using Openshift routes, you add a route type listener to your `Kafka` resource (list ener.type:route).

Once configuration is done, the Strimzi Cluster Operator deploys multiple routes as well as multiple ClusterIP type Kubernetes Services. This means that you will have the following:

- A route and a corresponding ClusterIP that serves as an external bootstrap. This is used by clients for the initial connection and to receive metadata (advertised listeners) from the Kafka cluster.
- A unique route and a CluserIP for each Kafka Broker. The routes and the corresponding ClusterIPs are used to access the brokers directly and to distinguish the traffic for different brokers.

Kafka clients connect to the bootstrap route, which routes the request through the bootstrap ClusterIP to one of the brokers. From this broker, the client receives metadata that contains the hostnames of the per-broker routes. The client uses these addresses to connect to the routes dedicated to the specific broker. Afterward, the route directs traffic through its corresponding ClusterIP to its corresponding broker.

The Strimzi Cluster Operator uses the HAProxy router and sets up routes with passthrough termination. This results in the following:

- Traffic going through a route is always secured and uses TLS encryption.
- Encrypted traffic is sent to the ClusterIP Service without data being decrypted in the process.
- The port that the routes listen on is fixed and is always 443. This is because HAProxy uses port 443 by default for HTTPS requests.

The Strimzi Cluster Operator collects the hostnames assigned to the routes and uses the addresses to configure the advertised listeners in the Kafka brokers. So brokers are automatically configured to advertise the right address and ports. As a result, once setup is complete, you can connect your clients running outside of the Kubernetes network by directing them to the bootstrap route. Kubernetes and OpenShift handle everything else and ensure that client requests are routed to the correct brokers.

Configuring route listeners

Complete the following steps to set up and configure a route type listener in your Kafka resource. The following steps also include an example on how to connect a Kafka console client to the cluster.

About this task

These steps demonstrate basic listener configuration with typical customizations. In addition to the configuration shown here, you can further customize your listener and specify a client authentication mechanism with the authenti cation property and add various additional configurations using the configuration property. For a comprehensive list of available properties, see *GenericKafkaListener* schema reference in the Strimzi API reference.

Procedure

1. Configure your Kafka resource.

   Add an external listener that has its type property set to route. Additionally, you must ensure that tls is set to true as TLS/SSL encryption is mandatory when using routes.

   Optionally, you can further customize the listener. For example, the following configuration snippet shows an example where the hostnames of routes are specified with the host property.

   ```
   #...
   kind: Kafka
   spec:
     kafka:
       listeners:
         - name: external
           port: 9094
           type: route
           tls: true
           authentication:
             type: tls
           configuration:
             bootstrap:
               host: kafka-bootstrap.router.com
             brokers:
               - broker: 0
                 host: kafka-0.router.com
               - broker: 1
                 host: kafka-1.router.com
               - broker: 2
                 host: kafka-2.router.com
   ```

   Note: Hosts are automatically assigned by OpenShift if you do not assign them. If you choose to override hostnames, ensure that they are available for use and match the configuration of the router as the Strimzi Cluster Operator does not perform any validation.

2. Verify that the configured services are created and ready.

   ```
   oc get svc
   ```

3. Get the host of the bootstrap route.

   ```
   oc get routes [***CLUSTER NAME***]-kafka-bootstrap \
     --output=jsonpath='{.status.ingress[0].host}{"\n"}'
   ```

4. Extract the TLS certificate from your broker and import it into a Java truststore file.

   Extracting the TLS certificate is required because TLS encryption is mandatory when using routes. Because of this, you must run your clients with a valid certificate. You can use the OpenShift CLI (oc) to extract the certificate and the keytool utility to import the certificate into a Java truststore file. For example:

   ```
   oc extract secret/[***CLUSTER NAME***]-cluster-ca-cert \
     --keys=ca.crt --to=- > ca.crt
   ```

   ```
   keytool -import -trustcacerts -alias [***ALIAS***] \
     -file ca.crt \
     -keystore truststore.jks \
     -storepass [***PASSWORD***] \
     -noprompt
   ```

5. Ensure that the resulting truststore is available on the machine where you run your client and that the client has access to the file.
6. Configure and run your client.

   The following example shows a Kafka console producer.

   ```
   kafka-console-producer.sh \
     --bootstrap-server [***BOOTSTRAP ROUTE HOST***]:443 \
     --producer-property security.protocol=SSL \
     --producer-property ssl.truststore.password=[***PASSWORD***] \
     --producer-property ssl.truststore.location=[***TRUSTSTORE LOCATION***]
    \
     --topic [***TOPIC***]
   ```

   Tip: Instead of passing TLS/SSL properties directly using the --producer-property option, you can also create a configuration file containing the properties and pass the file with --producer.config option.

Related Information

Service | Kubernetes

Accessing Kafka: Part 3 – OpenShift Routes | Strimzi blog

GenericKafkaListener schema reference | Strimzi API reference

Load balancer

Load balancers automatically and efficiently distribute network traffic between multiple backend servers. A load balancer setup can be used to expose your Kafka brokers to the outside world.

There are many load balancer implementations available and all cloud providers provide their own solutions. Different implementations handle load balancing on different levels of the network, most commonly you have layer 4 (transport) and layer 7 (application) load balancing. Strimzi in Cloudera Streams Messaging - Kubernetes Operator uses layer 4 load balancing. This is because common load balancer implementations do not support the Kafka protocol.

To set up external cluster access using load balancers, you add a loadbalancer type listener to your Kafka resource (listeners.type:loadbalancer).

Once configuration is done, the Strimzi Cluster Operator sets up multiple load balancers as well as multiple LoadBalancer type Kubernetes Services. This means that you will have the following:

- A load balancer and a corresponding LoadBalancer Service that serves as an external bootstrap. This is used by clients for the initial connection and to receive metadata (advertised listeners) from the Kafka cluster.
- A unique load balancer and a LoadBalancer Service for each Kafka Broker.

Note: Do not confuse the LoadBalancer type Service with the actual load balancers. The LoadBalancer Services are managed by Kubernetes. The load balancers are separate entities and are managed by the infrastructure or cloud provider.

The Strimzi Cluster Operator creates the LoadBalancer type Services first. Following the creation of the Services, the load balancers are automatically created. Typically your infrastructure provider assigns the load balancer a hostname and IP address. These are automatically added to the status section of the `Kafka` resource. The Strimzi Cluster Operator collects both hostname and IP address and uses them to configure the advertised listeners of your Kafka brokers.

The Strimzi Cluster Operator uses hostnames instead of IP addresses by default. This is because load balancer IP addresses might change, the hostnames, however, are fixed and remain the same as long as the load balancer is running. By default, the Strimzi Cluster Operator uses the IP address if there is no hostname assigned to the load balancer. In case you want to use IP addresses, you can do so by manually configuring them during setup.

Once the listener is configured, you can connect your clients running outside of the Kubernetes network by directing them to the bootstrap load balancer. The load balancers, Kubernetes, and Kafka handle everything else and ensure that client requests are routed to the correct brokers.

Configuring load balancer listeners

Complete the following steps to set up and configure a loadbalancer listener in your Kafka resource. The following steps also include an example on how to connect a Kafka console client to the cluster.

About this task

These steps demonstrate basic listener configuration with typical customizations. In addition to the configuration shown here, you can further customize your listener and enable and disable TLS encryption using the tls property, specify a client authentication mechanism with the authentication property, as well as add various additional configurations using the configuration property. For a comprehensive list of available properties, see the *GenericKafkaListener schema reference* in the Strimzi API reference.

Procedure

1. Configure your Kafka resource.

   Add a new external listener that has its type set to loadbalancer.

   Optionally, you can further customize the listener. For example, the following configuration snippet shows an example where the advertised hostnames and ports are specified using advertisedHost and advertisedPort properties.

```
#...
kind: Kafka
spec:
  kafka:
    listeners:
      - name: external
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: tls
        configuration:
          brokers:
            - broker: 0
              advertisedHost: my-broker-0.cloudera.com
              advertisedPort: 12340
            - broker: 1
              advertisedHost: my-broker-1.cloudera.com
              advertisedPort: 12341
            - broker: 2
```

```
              advertisedHost: my-broker-2.cloudera.com
              advertisedPort: 12342
```

Tip: The advertisedHost property also accepts IP addresses. Specify IP addresses instead if DNS resolution does not work for the Kafka clients. Configuring exact hostnames or ports does not change the hostname or port of the load balancer, instead it changes the address advertised by Kafka.

2. Verify that LoadBalancer type services as well as load balancers are running

```
kubectl get services --namespace [***NAMESPACE***]
```

The output will be similar to the following example.

```
NAME                                    TYPE        CLUSTER-IP       EXTER
NAL-IP
#...
my-cluster-kafka-external-bootstrap   LoadBalancer   10.43.18.136    10.65
.0.5
my-cluster-kafka-external-0           LoadBalancer   10.43.1.63      10.
65.0.6
my-cluster-kafka-external-1           LoadBalancer   10.43.46.74     10.
65.0.7
my-cluster-kafka-external-2           LoadBalancer   10.43.113.194   10.
65.0.8
```

Notice that there is a LoadBalancer Service deployed for each Kafka broker. Additionally you have a separate external bootstrap LoadBalancer called *[***CLUSTER NAME***]*-kafka-external-bootstrap.

Clients connecting to the Kafka cluster should be directed to the external bootstrap. The addresses in the EXTE RNAL-IP column are the hostnames or IPs of the load balancers. Having this column populated with values indicates that the load balancers are created.

3. Extract the TLS certificate form your broker and import it into a Java truststore file.

Doing the following is only required if you have TLS/SSL encryption enabled for the load balancer listener.

```
kubectl get secret [***CLUSTER NAME***]-cluster-ca-cert \
  --namespace [***NAMESPACE***] --output jsonpath='{.data.ca\.crt}' \
  | base64 -d > ca.crt
```

```
keytool -import -trustcacerts -alias [***ALIAS***] \
  -file ca.crt \
  -keystore truststore.jks \
  -storepass [***PASSWORD***] \
  -noprompt
```

4. Ensure that the resulting truststore is available on the machine where you run your client and that the client has access to the file.
5. Get the address of the bootstrap load balancer.

```
kubectl get kafka [***CLUSTER NAME***] \
  --namespace [***NAMESPACE***] \
  --output=jsonpath='{.status.listeners[?(@.name=="[***LISTENER
 NAME***]")].bootstrapServers}{"\n"}'
```

Clients that you want to connect to the cluster should be directed to this address.

6. Configure and run your client.

The following example shows a Kafka console producer. Configuring TLS/SSL related properties is only required if TLS/SSL is enabled for the load balancer listener.

```
kafka-console-producer.sh \
```

```
  --bootstrap-server [***BOOTSTRAP LOAD BALANCER HOST***]:9094 \
  -- producer-property security.protocol=SSL \
  --producer-property ssl.truststore.password=[***PASSWORD***] \
  --producer-property ssl.truststore.location=[***TRUSTSTORE LOCATION***]
\
  --topic [***TOPIC***]
```

Tip: Instead of passing TLS/SSL properties directly using the --producer-property option, you can also create a configuration file containing the properties and pass the file with --producer.config option.

Related Information

Service | Kubernetes

Accessing Kafka: Part 4 – Load Balancers | Strimzi blog

GenericKafkaListener schema reference | Strimzi API reference

Ingress

You can use Ingress to route HTTP/HTTPS traffic from outside the cluster to services within the cluster.

Important: If you are on OpenShift, use OpenShift routes (route type listeners) to configure external access to the cluster instead of Ingress.

Ingress has two main components. You have Ingress resources, which define the traffic routing rules to your services and pods. In addition, you have Ingress controllers, which route incoming requests based on the rules defined by Ingress resources.

The Ingress API is a native part of Kubernetes, Ingress controllers are not. This means that while creating Ingress resources is possible by default on any Kubernetes cluster, the Ingress controller must be installed separately, otherwise, Ingress cannot function.

While there are numerous controller implementations available for Kubernetes, Strimzi only supports Ingress-Nginx controllers running in TLS passthrough mode.

To set up external cluster access with Ingress, you add an ingress type listener to your Kafka resource (listener.type:ingress) and specify the hostnames for each broker and a bootstrap using the configuration property. In addition, TLS must be enabled for the listener, and, depending on your environment, specifying the Ingress class might be required.

Once configuration is done, the Strimzi Cluster Operator deploys multiple Ingress resources as well as multiple ClusterIP Services. This means that you will have the following:

- An Ingress and a corresponding ClusterIP that serves as an external bootstrap. This is used by clients for the initial connection and to receive metadata (advertised listeners) from the Kafka cluster.
- A unique Ingress and a ClusterIP for each Kafka Broker. These are used to access the brokers directly and to distinguish the traffic for different brokers.

Kafka clients connect to the bootstrap Ingress, which routes the request through the corresponding bootstrap service to one of the brokers. Connections to the individual brokers are then established using advertised listeners received from the broker. Traffic is then routed from the client to the broker through the broker-specific Ingresses and services.

Once the listener is configured, you can connect your clients running outside of the Kubernetes network by directing them to the bootstrap Ingress. Kubernetes, Ingress, and Kafka handle everything else and ensure that client requests are routed to the correct brokers.

Related Information

Ingress-Nginx Controller | Kubernetes Github.io

Configuring ingress listeners

Complete the following steps to set up and configure an ingress listener in your Kafka resource. The following steps also include an example on how to connect a Kafka console client to the cluster.

About this task

These steps demonstrate basic listener configuration. In addition to the configuration shown here, you can further customize your listener and specify a client authentication mechanism with the authentication property and add various additional configurations using the configuration property. For a comprehensive list of available properties, see *GenericKafkaListener schema reference* in the Strimzi API reference.

Before you begin

- Ensure that an Ingress-Nginx controller is deployed in your Kubernetes cluster.
- Ensure that the Ingress-Nginx controller has TLS Passthrough enabled.

Procedure

1. Configure your `Kafka` resource.

   To set up an ingress type listener, you need to configure multiple properties in your Kafka resource.

   a) Add an external listener that has its type property set to ingress.
   b) Specify Ingress hosts used for the different brokers as well as the bootstrap.

      This is done with the configuration property. Add the hostnames to the bootstrap and broker-*[***INDEX***]* prefixes that identify the bootstrap and brokers.
   c) Ensure that tls is set to true.
   d) Specify the Ingress class with the class property.

   Once configuration is done, your `Kafka` resource should look similar to the following example.

```
#...
kind: Kafka
spec:
  kafka:
    listeners:
      - name: external
        port: 9094
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: my-bootstrap.cloudera.com
          brokers:
            - broker: 0
              host: my-broker-0.cloudera.com
            - broker: 1
              host: my-broker-1.cloudera.com
            - broker: 2
              host: my-broker-2.cloudera.com
          class: nginx
```

2. Verify that both Ingress resources and ClusterIP Services are created and running.

   Use kubectl get ingress to list ingresses.

```
kubectl get ingress --namespace [***NAMESPACE***]
```

   The output will be similar to the following example.

```
NAME                          CLASS    HOSTS                          ADDRESS
  PORTS
#...
```

```
my-cluster-kafka-bootstrap   nginx   my-bootstrap.cloudera.com  10.14.9
1.1  80, 443
my-cluster-kafka-0           nginx   my-broker-0.cloudera.com   10.14.9
1.1  80, 443
my-cluster-kafka-1           nginx   my-broker-1.cloudera.com   10.14.9
1.1  80, 443
my-cluster-kafka-2           nginx   my-broker-2.cloudera.com   10.14.9
1.1  80, 443
```

Use kubectl get servicesto list Kubernetes Services.

```
kubectl get services --namespace [***NAMESPACE***]
```

The output will be similar to the following example.

```
NAME                                    TYPE        CLUSTER-IP      EXTERN
AL-IP
#...
my-cluster-kafka-external-bootstrap   ClusterIP   10.43.16.137    <none>
my-cluster-kafka-0                    ClusterIP   10.43.67.184    <none>
my-cluster-kafka-1                    ClusterIP   10.43.189.61    <none>
my-cluster-kafka-2                    ClusterIP   10.43.177.221   <none>
```

3. Extract the TLS certificate from your broker and import it into a Java truststore file.

   Extracting the TLS certificate is required because TLS encryption is mandatory when using Ingress. Because of this, you must run your clients with a valid certificate. You can use the kubectl get to extract the certificate and the keytool utility to import the certificate into a Java truststore file. For example:

   ```
   kubectl get secret [***CLUSTER NAME***]-cluster-ca-cert \
     --namespace [***NAMESPACE***] \
     --output jsonpath='{.data.ca\.crt}' \
     | base64 -d > ca.crt
   ```

   ```
   keytool -import -trustcacerts -alias [***ALIAS***] \
     -file ca.crt \
     -keystore truststore.jks \
     -storepass [***PASSWORD***] \
     -noprompt
   ```

4. Ensure that the resulting truststore is available on the machine where you will run your client and that the client has access to the file.
5. Configure your client.

   The following example shows a Kafka console producer. The port used by Ingress is typically 443.

   ```
   kafka-console-producer.sh \
     --bootstrap-server [***BOOTSTRAP INGRESS HOST***]:443 \
     --producer-property security.protocol=SSL \
     --producer-property ssl.truststore.password=[***PASSWORD***] \
     --producer-property ssl.truststore.location=[***TRUSTSTORE LOCATION***]
    \
     --topic [***TOPIC***]
   ```

   Tip: Instead of passing TLS/SSL properties directly using the --producer-property option, you can also create a configuration file containing the properties and pass the file with --producer.config option.

Related Information

Service | Kubernetes

Accessing Kafka: Part 5 – Ingress | Strimzi blog

GenericKafkaListener schema reference | Strimzi API reference

Accessing the Cruise Control REST API

Learn how you set up access to the Cruise Control REST API.

The Cruise Control REST API supports a number of GET requests, which can be used for read-only operations. These operations do not perform any Kafka changes and do not change the state or configuration of Cruise Control. Having access to these endpoints enables you to carry out operations such as the following.

- Query detailed Cruise Control specific statistics and data in a secure way. For example you can get access to information surrounding cluster and partition load as well as user tasks.
- Monitor Kafka cluster with the Cruise Control user interface.
- Debug Cruise Control securely.

Important: Using REST API endpoints that have a method different from GET (for example, POST, PUT, DELETE, and so on) interfere with the Strimzi Cluster Operator's management of Cruise Control leading to unexpected behavior. The use of these endpoints is not recommended or supported by Cloudera.

You configure access control to the Cruise Control REST API endpoints using a single Kubernetes `Secret`. The `Secret` contains the list of all users who are granted access to the endpoints and their role.

Strimzi uses roles to grant users or third-party applications different levels of access to the Cruise Control REST API. Each user is configured with a static password for basic HTTP authentication.

By default Cruise Control defines the following three roles.

- VIEWER – has access to the most lightweight `kafka_cluster_state`, `user_tasks` and `review_board` endpoints.
- USER – has access to all GET endpoints except `bootstrap` and `train`.
- ADMIN – has access to all endpoints.

Strimzi supports the USER and VIEWER roles only. This restriction is in place so that REST API calls made by users and third-party applications do not interfere with the calls, for example, the write operations, made by the Strimzi Cluster Operator and potentially cause damage to the Kafka cluster managed by Strimzi.

Configuring Cruise Control users

Learn how to configure REST API users for Cruise Control. Users you configure are granted access to the Cruise Control REST API.

About this task

You specify the users you want to grant access to the Cruise Control REST API in a `Secret`. The `Secret` must be referenced in spec.cruiseControl.apiusers of the `Kafka` resource.

Procedure

1. Create API users in Jetty's HashLoginService file format (cruise-control-auth.txt).

   Add your users, their passwords, as well as the roles.

   ```
   [***USER 1***]: [***PASSWORD 1***], VIEWER
   [***USER 2***]: [***PASSWORD 2***], USER
   ```

   Important: Ensure that there are no ADMIN role users defined in the file that you create. ADMIN role users are not supported. If you specify ADMIN role users, Cruise Control will fail to start.

2. Create a `Secret` using the file you created in the previous step.

```
kubectl create secret generic cruise-control-api-users-secret \
  --from-file=cruise-control-auth.txt=cruise-control-auth.txt
```

3. Reference the `Secret` in spec.cruiseControl.apiUsers of the `Kafka` resource.

```
#...
kind: Kafka
spec:
  cruiseControl:
    config:
      webserver.security.enable: true
      webserver.ssl.enable: true
    apiUsers:
      type: hashLoginService
      valueFrom:
        secretKeyRef:
          name: cruise-control-api-users-secret
          key: cruise-control-auth.txt
```

- webserver.security.enable – Enables HTTP Basic authentication for the Cruise Control REST API and enforces the policies defined in spec.cruiseControl.apiUsers.
- webserver.ssl.enable – Enables TLS encryption for the Cruise Control REST API.
- apiUsers – Configures the Cruise Control REST API users by referencing a `Secret`.

Note: Both webserver.security.enable and webserver.ssl.enable are set to true by default. Explicitly configuring them is not required.

Related Information

Cruise Control REST API reference

API users | Strimzi API reference

Security | Cruise Control

Configuring external access

Learn how to enable external access for the Cruise Control REST API. Configuring external access makes it possible for Cruise Control users to access the REST API from outside the Kubernetes cluster.

About this task

The Cruise Control REST API can be secured with authentication, authorization and encryption. As a result, it is considered safe to allow access from outside the Kubernetes cluster as well. Cloudera recommends that access control is always used when enabling external access to Cruise Control.

By default, the Strimzi Cluster Operator generates a strict network policy that blocks external connections to Cruise Control. Additionally, the TLS certificates for Cruise Control are automatically generated and cannot be modified. As a result, to set up external access to Cruise Control you require the following.

- Have or create a resource, like an NGINX-based Ingress, to route and manage traffic coming from outside the cluster. Any type of resource can be used that can route outside traffic.
- Create a new network policy that enables access to Cruise Control.
- Use the Cruise Control certificates internally.

  When TLS is enabled for Cruise Control the service certificates are generated by Strimzi and cannot be modified. This is because most of the ssl. configurations are restricted and managed by the Strimzi Cluster Operator. Because of this, the resource (for example, an Ingress) providing access to the Kubernetes cluster must use the

generated TLS credentials to communicate with Cruise Control. External connections can be configured with user generated and managed certificates.

The following steps demonstrate how you can configure NGINX-based Ingress to access Cruise Control. This is just a specific example and any other type of Ingress can be used.

Procedure

1. Create a `NetworkPolicy` that allows the connection from the Ingress pod.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: [***NEW CRUISE CONTROL NETWORK POLICY NAME***]
  namespace: [***CRUISE CONTROL NAMESPACE***]
spec:
  podSelector:
    matchLabels:
      strimzi.io/cluster: [***KAFKA CLUSTER NAME***]
      strimzi.io/kind: Kafka
      strimzi.io/name: [***KAFKA CLUSTER NAME***]-cruise-control
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector: {}
      podSelector:
        matchLabels:
          app.kubernetes.io/instance: ingress-nginx
    ports:
    - protocol: TCP
      port: 9090
```

2. Get the generated Cruise Control certificate and key.

```
kubectl get secret [***KAFKA CLUSTER NAME***]-cruise-control-certs \
  --namespace [***KAFKA NAMESPACE***] \
  --output "jsonpath={.data.cruise-control\.crt}" \
| base64 -d > cert.crt
```

```
kubectl get secret [***KAFKA CLUSTER NAME***]-cruise-control-certs \
  --namespace [***KAFKA NAMESPACE***] \
  --output "jsonpath={.data.cruise-control\.key}" \
| base64 -d > cert.key
```

3. Create a `Secret` with the specific format of your Ingress using the files created in the previous step.

   These needed to be updated manually if the Cruise Control `Secret` was regenerated.

```
kubectl create secret tls [***CRUISE CONTROL INGRESS SECRET NAME***] \
  --key ./cert.key \
  --cert ./cert.crt \
  --namespace [***KAFKA NAMESPACE***]
```

4. Create the Ingress rule.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cruise-control-ingress-service
  namespace: [***KAFKA NAMESPACE***]
  annotations:
    nginx.ingress.kubernetes.io/use-regex: 'true'
```

```
    nginx.ingress.kubernetes.io/backend-protocol: 'HTTPS'
    nginx.ingress.kubernetes.io/proxy-ssl-secret: '[***KAFKA NAMESPACE***]
/[***CRUISE CONTROL INGRESS SECRET NAME***]'
spec:
  ingressClassName: nginx
  tls:
  - hosts:
      - [***HOST NAME***]
    secretName: [***INGRESS SECRET NAME***]
  rules:
  - host: [***HOST NAME***]
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: [***KAFKA CLUSTER NAME***]-cruise-control
            port:
              number: 9090
```

- nginx.ingress.kubernetes.io/backend-protocol– Instructs the Ingress to use encrypted communication between the Ingress and the Cruise Control pod.
- nginx.ingress.kubernetes.io/proxy-ssl-secret – Specifies the `Secret` which contains the Cruise Control certificate in the required format of the Ingress solution.
- spec.tls – Enables secure connection between the clients and the Ingress itself. This property must define the same host as the rule. the `Secret` should point to the `Secret` where the credentials for the secure client communication are stored.

Related Information

Ingress | Kubernetes

Network Policies | Kubernetes

TLS/HTTPS | Ingress-Nginx Controller