

How to Install and Secure Apache Kafka on Vultr Kubernetes Engine (VKE)

Learn how to install Apache Kafka on Vultr Kubernetes Engine (VKE) with step-by-step instructions and best practices for securing your deployment.

Contents

01	Introduction	3
02	Introduction to Apache Kafka	3
03	Introduction to Strimzi	3
04	Prerequisites	5
05	Install Strimzi on Vultr Kubernetes Engine	6
06	Setup Kafka cluster	8
07	Setup a secure Kafka cluster	13
08	Delete Vultr Kubernetes Engine cluster	20
09	Conclusion	20

Introduction

In this article, you will learn how to run Apache Kafka on Kubernetes using the open-source Strimzi project. You will setup Strimzi on Vultr Kubernetes Engine, setup a Kafka cluster, sent messages to a topic and received messages from that topic. You will also secure the Kafka cluster using encryption and user authentication.

Introduction to Apache Kafka

Apache Kafka is a messaging system that allows clients to publish and read streams of data (also called events). It has an ecosystem of open-source solutions that you can combine to store, process, and integrate these data streams with other parts of your system in a secure, reliable, and scalable manner.

Key components of Apache Kafka:

- **Broker (Node):** A Kafka broker runs the Kafka JVM process. A best practice is to run three or more brokers for scalability and high availability. These groups of Kafka brokers form a cluster.
- **Producers:** These are client applications that send messages to Kafka. Each message is nothing but a key-value pair.
- **Topics:** Events (messages) are stored in topics, and each topic has one or more partitions. Data in each of these partitions are distributed across the Kafka cluster for high availability and redundancy.
- **Consumers:** Just like producers, consumers are also client applications. They receive and process data/events from Kafka topics.

Introduction to Strimzi

Strimzi can be used to run an Apache Kafka cluster on Kubernetes. In addition to the cluster itself, Strimzi can also help you manage topics, users, Mirror Maker and Kafka Connect deployments. With Strimzi, you can configure the

cluster as per your needs. This includes advanced features such as rack awareness configuration to distribute Kafka nodes across availability zones, as well as Kubernetes taints and tolerations to pin Kafka to dedicated worker nodes in your Kubernetes cluster. You can also expose Kafka to external clients outside the Kubernetes cluster using `Service` types such as `NodePort`, `LoadBalancer` etc. and these can be secured using `SSL`.

All this is made possible with a combination of Custom resources, Operators and respective Docker container images.

Strimzi Custom Resources and Operators

You can customize Strimzi Kafka components in a Kubernetes cluster using custom resources. These are created as instances of APIs added by Custom resource definitions (CRDs) that extend Kubernetes resources. Each Strimzi component has an associated CRD which is used to describe that component. Thanks to CRDs, Strimzi resources benefit from Kubernetes features such as CLI accessibility and configuration validation.

Once a Strimzi custom resource is created, it's managed using Operators. Operators are a method of packaging, deploying, and managing a Kubernetes-native application. Because Strimzi Operators automate common and complex tasks related to a Kafka deployment, Kafka administration tasks are simplified and require less manual intervention.

Let's look at the Strimzi operators and the custom resources they manage.

Cluster Operator

Strimzi Cluster Operator is used to deploy and manage Kafka components. Although a single Cluster Operator instance is deployed by default, you can add replicas with leader election to ensure operator high availability.

The Cluster Operator manages the following Kafka components:

- **Kafka** - The `Kafka` resource is used to configure a Kafka deployment. Configuration options for the ZooKeeper cluster also included within the `Kafka` resource.
- **KafkaConnector** - This resources allow you to create and manage connector instances for Kafka Connect.

- **KafkaMirrorMaker2** - It can be used to run and manage a Kafka MirrorMaker 2.0 deployment. MirrorMaker 2.0 replicates data between two or more Kafka clusters, within or across data centers.
- **KafkaBridge** - This recourse managed Kafka Bridge, which is component that provides an API for integrating HTTP-based clients with a Kafka cluster.

Entity Operator

Entity Operator comprises the `Topic` and `User` Operator.

- **Topic Operator** - It provides a way of managing topics in a Kafka cluster through Kubernetes resources. You can declare a `KafkaTopic` resource as part of your application's deployment and the Topic Operator will take care of creating the topic for you and keeping them in-sync with corresponding Kafka topics. Information about each topic in a topic store, which is continually synchronized with updates from Kafka topics or Kubernetes `KafkaTopic` custom resources. If a topic is reconfigured or reassigned to other brokers, the `KafkaTopic` will always be up to date.
- **User Operator** - It allows you to declare a `KafkaUser` resource as part of your application's deployment along with authentication and authorization mechanisms for the user. You can also configure user quotas that control usage of Kafka resources. In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's access rights in the `KafkaUser` declaration.

Prerequisites

1. [Install kubectl](#) on your local workstation. It is a Kubernetes command-line tool that allows you to run commands against Kubernetes clusters.
2. Deploy a Vultr Kubernetes Engine (VKE) cluster using the [Reference Guide](#). Once it's deployed, from the **Overview** tab, click the **Download Configuration** button in the upper-right corner to download your `kubeconfig` file and save it to a local directory.

Point `kubectl` to Vultr Kubernetes Engine cluster by setting the `KUBECONFIG` environment variable to the path where you downloaded the cluster `kubeconfig` file in the previous step.

```
export KUBECONFIG=<path to VKE kubeconfig file>
```

Verify the same using the following command:

```
kubectl config current-context
```

Install Strimzi on Vultr Kubernetes Engine

1. Create a namespace called `kafka`.

```
kubectl create namespace kafka
```

You should see this output:

```
namespace/kafka created
```

2. Apply the Strimzi installation files, including `ClusterRoles`, `ClusterRoleBindings` and Custom Resource Definitions (CRDs).

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

You should see this output:

```
clusterrole.rbac.authorization.k8s.io/strimzi-kafka-broker created
clusterrole.rbac.authorization.k8s.io/strimzi-cluster-operator-namespaced
created
customresourcedefinition.apiextensions.k8s.io/
kafkamirrormaker2s.kafka.strimzi.io created
rolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator-leader-election
created
customresourcedefinition.apiextensions.k8s.io/kafkaconnectors.kafka.strimzi.io
created
customresourcedefinition.apiextensions.k8s.io/kafkabridges.kafka.strimzi.io
```

```

created
  customresourcedefinition.apiextensions.k8s.io/kafkamirrormakers.kafka.strimzi.io
created
  clusterrolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator-kafka-
broker-delegation created
  rolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator-watched created
  clusterrolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator created
  customresourcedefinition.apiextensions.k8s.io/kafkatopics.kafka.strimzi.io
created
  customresourcedefinition.apiextensions.k8s.io/kafkaconnects.kafka.strimzi.io
created
  deployment.apps/strimzi-cluster-operator created
  clusterrole.rbac.authorization.k8s.io/strimzi-cluster-operator-global created
  customresourcedefinition.apiextensions.k8s.io/kafkarebalances.kafka.strimzi.io
created
  clusterrole.rbac.authorization.k8s.io/strimzi-cluster-operator-leader-election
created
  clusterrolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator-kafka-
client-delegation created
  customresourcedefinition.apiextensions.k8s.io/kafkausers.kafka.strimzi.io
created
  clusterrole.rbac.authorization.k8s.io/strimzi-cluster-operator-watched created
  clusterrole.rbac.authorization.k8s.io/strimzi-kafka-client created
  configmap/strimzi-cluster-operator created
  clusterrole.rbac.authorization.k8s.io/strimzi-entity-operator created
  rolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator-entity-operator-
delegation created
  rolebinding.rbac.authorization.k8s.io/strimzi-cluster-operator created
  serviceaccount/strimzi-cluster-operator created
  customresourcedefinition.apiextensions.k8s.io/strimzipodsets.core.strimzi.io
created
  customresourcedefinition.apiextensions.k8s.io/kafkas.kafka.strimzi.io created

```

- Follow the deployment of the Strimzi cluster operator and wait for the `Pod` to transition to `Running` status.

```
kubectl get pod -n kafka --watch
```

You should see this output (the `Pod` name might differ in your case):

NAME	READY	STATUS	RESTARTS	AGE
strimzi-cluster-operator-56d64c8584-7k6sr	1/1	Running	0	43s

To check the operator's log:

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

Setup Kafka cluster

1. Create a directory and switch to it:

```
mkdir vultr-vke-kafka  
cd vultr-vke-kafka
```

2. Create a new file `kafka-cluster-1.yml`:

```
touch kafka-cluster-1.yml
```

3. Add the below contents to `kafka-cluster-1.yml` file and save it.

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: Kafka  
metadata:  
  name: my-cluster-1  
spec:  
  kafka:  
    version: 3.3.1  
    replicas: 1  
    listeners:  
      - name: plain  
        port: 9092  
        type: internal  
        tls: false  
      - name: tls  
        port: 9093  
        type: internal  
        tls: true  
    config:  
      offsets.topic.replication.factor: 1  
      transaction.state.log.replication.factor: 1  
      transaction.state.log.min.isr: 1  
      default.replication.factor: 1
```

```
min.insync.replicas: 1
inter.broker.protocol.version: "3.3"
storage:
  type: ephemeral
zookeeper:
  replicas: 1
  storage:
    type: ephemeral
entityOperator:
  topicOperator: {}
  userOperator: {}
```

4. Install the Kafka cluster:

```
kubectl apply -f kafka-cluster-1.yml -n kafka
```

You should see this output:

```
kafka.kafka.strimzi.io/my-cluster-1 created
```

Wait for cluster to be created.

```
kubectl wait kafka/my-cluster-1 --for=condition=Ready --timeout=300s -n kafka
```

Once completed, you will see this output:

```
kafka.kafka.strimzi.io/my-cluster-1 condition met
```

5. Verify Kafka cluster

```
kubectl get kafka -n kafka
```

You should see this output:

NAME	DESIRED KAFKA REPLICAS	DESIRED ZK REPLICAS	READY	WARNINGS
my-cluster-1	1	1	True	True

1. Verify Kafka `Pod`

```
kubectl get pod/my-cluster-1-kafka-0 -n kafka
```

You should see this output:

NAME	READY	STATUS	RESTARTS	AGE
my-cluster-1-kafka-0	1/1	Running	0	9m23s

2. Verify Zookeeper `Pod`

```
kubectl get pod/my-cluster-1-zookeeper-0 -n kafka
```

You should see this output:

NAME	READY	STATUS	RESTARTS	AGE
my-cluster-1-zookeeper-0	1/1	Running	0	10m

3. Check the `ConfigMap`s associated with the cluster:

```
kubectl get configmap -n kafka
```

You should see this output:

NAME	DATA	AGE
kube-root-ca.crt	1	57m
my-cluster-1-entity-topic-operator-config	1	9m20s
my-cluster-1-entity-user-operator-config	1	9m20s
my-cluster-1-kafka-0	3	9m45s
my-cluster-1-zookeeper-config	2	10m
strimzi-cluster-operator	1	57m

4. Check `Service`s associated with the cluster:

```
kubectl get svc -n kafka
```

You should see this output (the `ClusterIP`s might differ in your case):

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
my-cluster-1-kafka-bootstrap	ClusterIP	10.96.23.73	<none>
my-cluster-1-kafka-brokers	ClusterIP	None	<none>
my-cluster-1-zookeeper-client	ClusterIP	10.108.246.28	<none>
my-cluster-1-zookeeper-nodes	ClusterIP	None	<none>

5. Check `Secret`s associated with the cluster:

```
kubectl get secret -n kafka
```

You should see this output:

NAME	TYPE	DATA	AGE
my-cluster-1-clients-ca	Opaque	1	10m
my-cluster-1-clients-ca-cert	Opaque	3	10m
my-cluster-1-cluster-ca	Opaque	1	10m
my-cluster-1-cluster-ca-cert	Opaque	3	10m
my-cluster-1-cluster-operator-certs	Opaque	4	10m
my-cluster-1-entity-topic-operator-certs	Opaque	4	9m44s
my-cluster-1-entity-user-operator-certs	Opaque	4	9m44s
my-cluster-1-kafka-brokers	Opaque	4	10m
my-cluster-1-zookeeper-nodes	Opaque	4	10m

You can test the Kafka cluster using the Kafka CLI based consumer and producer.

Verify cluster operation

You will verify cluster functionality by producing data using Kafka CLI producer and consuming data using Kafka CLI consumer.

1. Run a `Pod` to execute Kafka CLI producer and send data to a topic

```
kubectl -n kafka run kafka-producer -ti --image=quay.io/strimzi/kafka:0.32.0-kafka-3.3.1 --rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server my-cluster-1-kafka-bootstrap:9092 --topic my-topic
```

You should see the following output with prompt

```
If you don't see a command prompt, try pressing enter.  
>
```

Enter messages in the prompt. These will be send to the specified Kafka topic.

2. Open a new terminal. Point `kubectl` to Vultr Kubernetes Engine cluster by setting the `KUBECONFIG` environment variable to the path where you downloaded the cluster `kubeconfig` file.

```
export KUBECONFIG=<path to VKE kubeconfig file>
```

3. Run a `Pod` to execute Kafka CLI consumer to consume data from a topic

```
kubectl -n kafka run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.32.0-kafka-3.3.1 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-cluster-1-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

You should receive messages you sent from the producer terminal.

4. Press `ctrl+c` on each terminal to close them. This will delete both the `Pod` s.
5. Delete the Kafka cluster

```
kubectl delete -f kafka-cluster-1.yml -n kafka
```

Verify that the associated `Pod` s were deleted. Wait for `my-cluster-1-kafka-0` and `my-cluster-1-zookeeper-0` `Pod` s to terminate.

```
kubectl get pods -n kafka
```

Setup a secure Kafka cluster

So far, you have setup a simple Kafka cluster. In the next section, you will learn how to secure the setup by using the following:

- Encryption via TLS.
- Authentication via `SASL_SCRAM`.

1. Create a new file `kafka-cluster-2.yml`:

```
touch kafka-cluster-2.yml
```

2. Add the below contents to `kafka-cluster-2.yml` file and save it.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster-2
spec:
  kafka:
    version: 3.3.1
    replicas: 1
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: true
        authentication:
          type: scram-sha-512
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.3"
```

```
storage:
  type: ephemeral
zookeeper:
  replicas: 1
  storage:
    type: ephemeral
entityOperator:
  topicOperator: {}
  userOperator: {}
```

3. Install the Kafka cluster:

```
kubectl apply -f kafka-cluster-2.yml -n kafka
```

You should see this output:

```
kafka.kafka.strimzi.io/my-cluster-2 created
```

Wait for cluster to be created.

```
kubectl wait kafka/my-cluster-2 --for=condition=Ready --timeout=300s -n kafka
```

Once completed, you will see this output:

```
kafka.kafka.strimzi.io/my-cluster-2 condition met
```

4. Verify Kafka cluster

```
kubectl get kafka -n kafka
```

You should see this output:

NAME	DESIRED KAFKA REPLICAS	DESIRED ZK REPLICAS	READY	WARNINGS
my-cluster-2	1	1	True	True

5. Create a new file `kafka-user.yml`:

```
touch kafka-user.yml
```

6. Add the below contents to `kafka-user.yml` file and save it.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: test-kafka-user
  labels:
    strimzi.io/cluster: my-cluster-2
spec:
  authentication:
    type: scram-sha-512
```

7. Create the `KafkaUser` resource

```
kubectl apply -f kafka-user.yml -n kafka
```

You should see this output:

```
kafkauser.kafka.strimzi.io/test-kafka-user created
```

8. Verify user creation

```
kubectl get kafkauser -n kafka
```

You should see this output:

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
test-kafka-user	my-cluster-2	scram-sha-512		True

When the user is created, the User Operator creates a Kubernetes `Secret` and seeds it with the user credentials required to authenticate to the Kafka cluster.

9. Verify the `Secret`

```
kubectl get secret/test-kafka-user -n kafka -o yaml
```

Verify cluster operation

You will verify cluster functionality by producing data using Kafka CLI producer and consuming data using Kafka CLI consumer.

- The CLI clients will connect to the Kafka broker using SSL.
- The CLI clients will need to authenticate to the Kafka broker using username and password.

Send data to Kafka topic

1. Fetch the password for the Kafka user that you had created and save it to your local workstation.

```
kubectl get secret test-kafka-user -n kafka -o jsonpath='{.data.password}' | base64 --decode > user.password
```

2. Fetch the Kafka server certificate and save it to your local workstation.

```
kubectl get secret my-cluster-2-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' -n kafka | base64 --decode > ca.p12
```

3. Fetch the Kafka server certificate password and save it to your local workstation.

```
kubectl get secret my-cluster-2-cluster-ca-cert -o jsonpath='{.data.ca\.password}' -n kafka | base64 --decode > ca.password
```

4. Open a new terminal. Point `kubectl` to Vultr Kubernetes Engine cluster by setting the `KUBECONFIG` environment variable to the path where you downloaded the cluster `kubeconfig` file.

```
export KUBECONFIG=<path to VKE kubeconfig file>
```

5. Start a new `Pod` name `kafka-producer`

```
kubectl -n kafka run kafka-producer -ti --image=quay.io/strimzi/kafka:0.32.0-kafka-3.3.1 --rm=true --restart=Never
```

You should see a shell prompt after the `Pod` starts

```
[kafka@kafka-producer kafka]$
```

6. From the previous terminal, copy the local certificate into the `kafka-producer Pod` that you just started:

```
kubectl cp ca.p12 kafka-producer:/tmp -n kafka
```

7. Go back to the terminal where the `kafka-producer Pod` is running and execute the below commands

```
cp $JAVA_HOME/lib/security/cacerts /tmp/cacerts
chmod 777 /tmp/cacerts
```

8. Import the server CA certificate in to the keystore. For `keypass`, use the password you had saved to your local `ca.password` file

```
keytool -importcert -alias strimzi-kafka-cert -file /tmp/ca.p12 -keystore /tmp/cacerts -keypass <password from ca.password file> -storepass changeit -noprompt
```

You should see this output

```
Certificate was added to keystore
```

9. Create the configuration file which will be used by the Kafka CLI producer. For `password`, use the password you had saved to your local `user.password` file

```
cat > /tmp/producer.properties << EOF
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required username="test-kafka-user" password="<password from user.password
```

```
file>;  
ssl.truststore.location=/tmp/cacerts  
ssl.truststore.password=changeit  
EOF
```

10. Send data to a topic

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-2-kafka-bootstrap:  
9092 --topic my-topic --producer.config /tmp/producer.properties
```

You should see the following output with prompt

```
If you don't see a command prompt, try pressing enter.  
>
```

Enter messages in the prompt. These will be send to the specified Kafka topic.

Receive data from Kafka topic

1. Open a new terminal. Point `kubectl` to Vultr Kubernetes Engine cluster by setting the `KUBECONFIG` environment variable to the path where you downloaded the cluster `kubeconfig`.

```
export KUBECONFIG=<path to VKE kubeconfig file>
```

2. Start a new `Pod` name `kafka-consumer`

```
kubectl -n kafka run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.32.0-  
kafka-3.3.1 --rm=true --restart=Never
```

You should see a shell prompt after the `Pod` starts

```
[kafka@kafka-consumer kafka]$
```

3. From the previous terminal, copy the local certificate into the `kafka-consumer` `Pod` that you just started:

```
kubectl cp ca.p12 kafka-consumer:/tmp -n kafka
```

- Go back to the terminal where the `kafka-consumer` Pod is running and execute the below commands to

```
cp $JAVA_HOME/lib/security/cacerts /tmp/cacerts
chmod 777 /tmp/cacerts
```

- Import the server CA certificate in to the keystore. For `keypass`, use the password you had saved to your local `ca.password` file

```
keytool -importcert -alias strimzi-kafka-cert -file /tmp/ca.p12 -keystore /tmp/cacerts -keypass <password from ca.password file> -storepass changeit -noprompt
```

You should see this output

```
Certificate was added to keystore
```

- Create the configuration file which will be used by the Kafka CLI consumer. For `password`, use the password you had saved to your local `user.password` file

```
cat > /tmp/consumer.properties << EOF
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required username="test-kafka-user" password="jCtF8vhh23Lu";
ssl.truststore.location=/tmp/cacerts
ssl.truststore.password=changeit
EOF
```

- Use Kafka CLI consumer to consume data from the topic

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-2-
kafka-bootstrap:9092 --topic my-topic --consumer.config /tmp/consumer.properties
--from-beginning
```

You should receive messages you sent from the producer terminal.

8. From a new terminal, delete the Kafka cluster

```
kubectl delete -f kafka-cluster-2.yml -n kafka
```

Verify that the associated `Pod`s were deleted. Wait for `my-cluster-2-kafka-0` and `my-cluster-2-zookeeper-0` `Pod`s to terminate.

```
kubectl get pods -n kafka
```

Delete Vultr Kubernetes Engine cluster

After you have completed the tutorial in this article, you can delete the Vultr Kubernetes Engine cluster.

Conclusion

In this article, you learnt how to use Strimzi to run Kafka and its related components on Kubernetes. You installed Strimzi on Vultr Kubernetes Engine, setup a Kafka cluster, sent messages to a topic and received messages from that topic. Next, you secured the Kafka cluster by enforcing TLS encryption as well as SASL authentication. TLS encryption ensured that the clients could only connect via SSL and with SASL authentication, clients had to specify the username and password to interact with the cluster (send or receive data).

You can also learn more in the following documentation:

- [Vultr Kubernetes Engine \(VKE\) Reference Guide](#)
- [Vultr Kubernetes Engine \(VKE\) Changelog](#)
- [How to Install MongoDB on Vultr Kubernetes Engine \(VKE\)](#)
- [How to Deploy MS SQL Server 2022 on Vultr Kubernetes Engine](#)
- [How to Install GitLab Runner on Vultr Kubernetes Engine](#)



VULTR

