Hortonworks Data Platform

Apache Kafka Component Guide

(Nov 30, 2016)

docs.hortonworks.com

Hortonworks Data Platform: Apache Kafka Component Guide

Copyright © 2012-2016 Hortonworks, Inc. Some rights reserved.

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 3.0 License.
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Table of Contents

List of Tables

1. Building a High-Throughput Messaging System with Apache Kafka

Apache Kafka is a fast, scalable, durable, fault-tolerant publish-subscribe messaging system. Common use cases include:

- Stream processing
- Messaging
- Website activity tracking
- Metrics collection and monitoring
- Log aggregation
- Event sourcing
- Distributed commit logging

Kafka works with Apache Storm and Apache Spark for real-time analysis and rendering of streaming data. The combination of messaging and processing technologies enables stream processing at linear scale.

For example, Apache Storm ships with support for Kafka as a data source using Storm's core API or the higher-level, micro-batching Trident API. Storm's Kafka integration also includes support for writing data to Kafka, which enables complex data flows between components in a Hadoop-based architecture. For more information about Apache Storm, see the Storm User Guide.

2. What's New

New features and changes for Apache Kafka have been introduced in Hortonworks Data Platform, version 2.5, along with documentation updates. New features and documentation updates are described in the following sections.

2.1. Apache Kafka

HDP 2.5 supports Apache Kafka version 0.10.0. Important new features include the following:

Fault tolerance

Balancing replicas across racks	This rack awareness feature limits the risk of data loss if all brokers on a rack fail at once. The feature distributes replicas of a partition across different racks, extending guarantees that Kafka provides for broker failures so that they now cover rack failure. For more information, see Balancing Replicas Across Racks at apache.org.

Security

Security/SASL improvements	Kafka supports authentication using SASL/PLAIN. For more information, see Apache JIRA KAFKA-2658.

Application Development

New client-side event interceptors	Two new plugin interfaces, `ProducerInterceptor` on producer and `ConsumerInterceptor` on consumer, allow developers to implement and configure custom interceptors. • The *ProducerInterceptor* interface allows processes to intercept events happening to a producer record, such as sending the producer record or receiving an acknowledgment when a record is published. For more information, see the ProducerInterceptor javadoc. • The *ConsumerInterceptor* interface allows processes to intercept consumer events, such as record being received or a record being consumed by a client. For more information, see the ConsumerInterceptor javadoc. For more information, see Add Producer and Consumer Interceptors at apache.org.
New Kafka Streams client library	The Kafka Streams API allows developers to implement distributed stream processing applications that consume from and produce data to Kafka topics.

	Note that the Kafka Streams API is a technical preview; the code is considered to be at alpha quality level. Public APIs are likely to change in future releases. For more information, see Streams API at apache.org.
New timestamp field for messages	Messages are now tagged with timestamps when they are produced. For more information, see Apache JIRA KAFKA-3025.
New configuration parameter `max.poll.records`	`max.poll.records` is a Kafka Consumer parameter that allows developers to limit the number of messages returned in a single call to `poll()`. For more information, see Apache JIRA KAFKA-3007

For detailed information about new features in Kafka version 0.10.0, see the Apache Release Notes for Kafka 0.10.0.

2.2. Content Updates

- Added detailed instructions for installing Kafka on an Ambari-managed cluster; see Installing Kafka using Ambari.
- Added Configuring Kafka for a Production Environment.

3. Apache Kafka Concepts

This chapter describes several basic concepts that support fault-tolerant, scalable messaging provided by Apache Kafka:

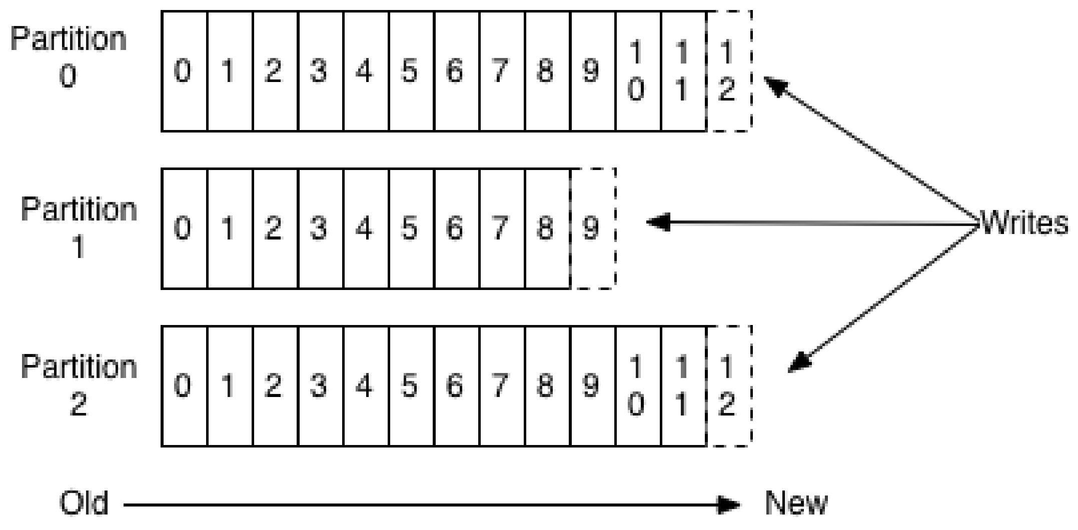
- Topics
- Producers
- Consumers
- Brokers

For additional introductory information about Kafka, see the Apache introduction to Kafka. For an example that simulates the use of streaming geo-location information (based on a previous version of Kafka), see Simulating and Transporting the Real-Time Event Stream with Apache Kafka.

Topics

Kafka maintains feeds of messages in categories called *topics*. Each topic has a user-defined category (or feed name), to which messages are published.

Architecturally, topics consist of one or more partitions:



Kafka appends new messages to a partition in an ordered, immutable sequence. Each message in a topic is assigned a sequential number that uniquely identifies the message within a partition. This number is called an *offset*, and is represented in the diagram by numbers within each cell (such as 0 through 12 in partition 0).

Partition support within topics provides parallelism within a topic. In addition, because writes to a partition are sequential, the number of hard disk seeks is minimized. This reduces latency and increases performance.

Producers

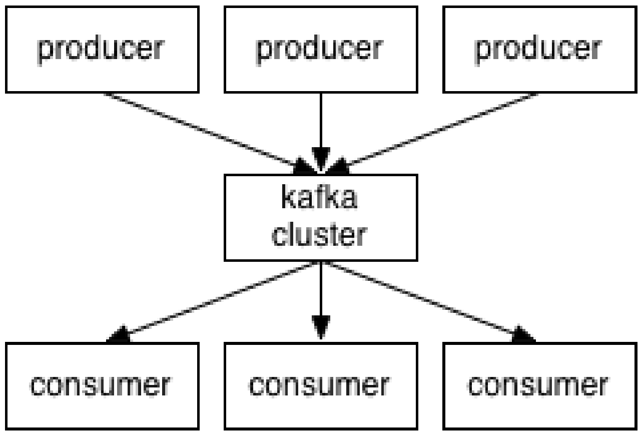
Producers are processes that publish messages to one or more Kafka topics. The producer is responsible for choosing which message to assign to which partition within a topic. Assignment can be done in a round-robin fashion to balance load, or it can be based on a semantic partition function.

Consumers

Consumers are processes that subscribe to one or more topics and process the feeds of published messages from those topics. Kafka consumers keep track of which messages have already been consumed by storing the current offset. Because Kafka retains all messages on disk for a configurable amount of time, consumers can use the offset to rewind or skip to any point in a partition.

Brokers

A Kafka cluster consists of one or more servers, each of which is called a broker. Producers send messages to the Kafka cluster, which in turn serves them to consumers. Each broker manages the persistence and replication of message data.



Kafka Brokers scale and perform well in part because Brokers are not responsible for keeping track of which messages have been consumed. Instead, the message consumer is responsible for this. This design feature eliminates the potential for back-pressure when consumers process messages at different rates.

4. Installing Kafka

Although you can install Kafka on a cluster not managed by Ambari (see Installing and Configuring Apache Kafka in the *Non-Ambari Cluster Installation Guide*), this chapter describes how to install Kafka on an Ambari-managed cluster.

4.1. Prerequisites

Before installing Kafka, ZooKeeper must be installed and running on your cluster.

Note that the following underlying file systems are supported for use with Kafka:

- EXT4: supported and recommended
- EXT3: supported

Caution

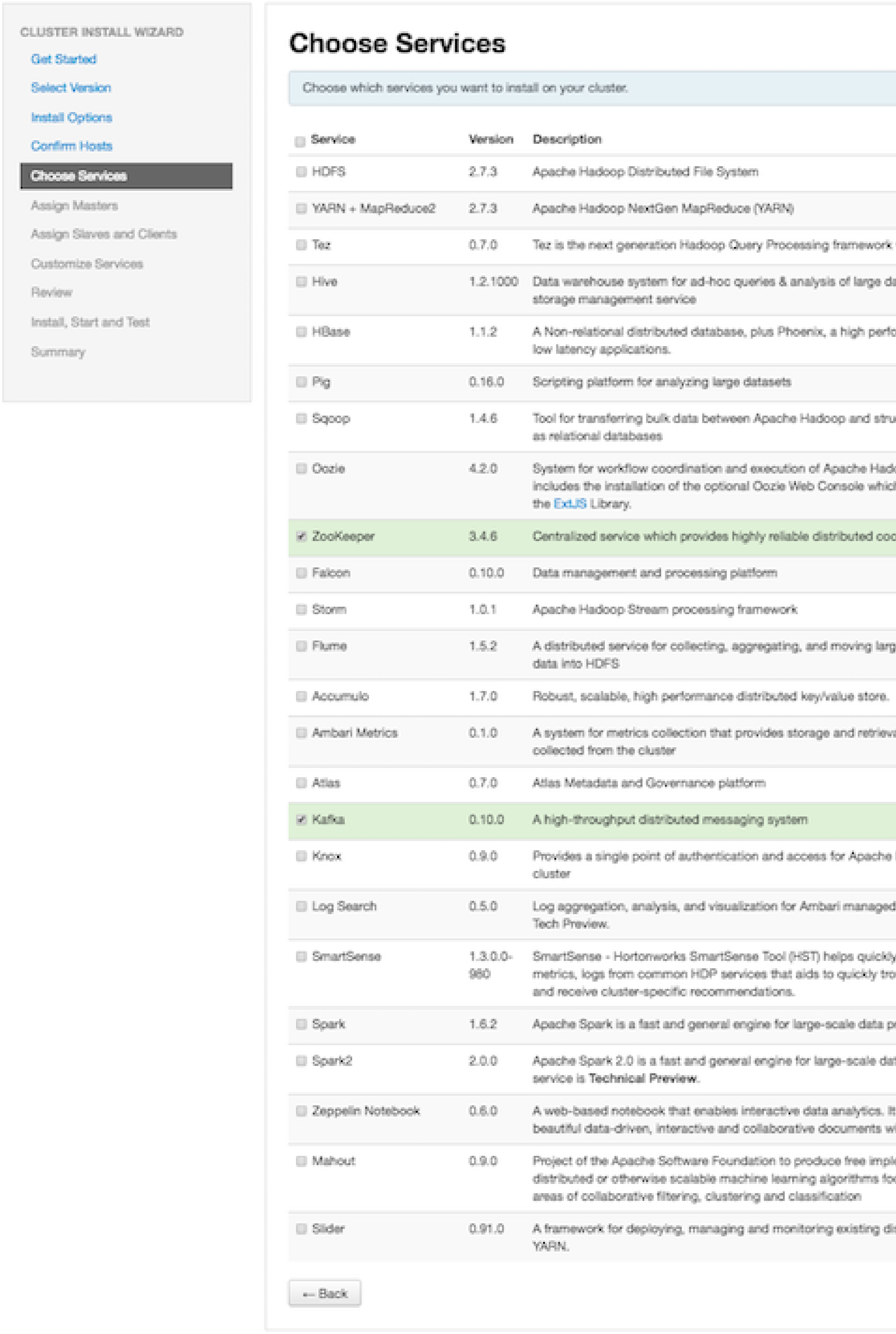
Encrypted file systems such as SafenetFS are not supported for Kafka. Index file corruption can occur.

4.2. Installing Kafka Using Ambari

Before you install Kafka using Ambari, refer to Adding a Service for background information about how to install Hortonworks Data Platform (HDP) components using Ambari.

To install Kafka using Ambari, complete the following steps.

1. Click the Ambari "Services" tab.
2. In the Ambari "Actions" menu, select "Add Service." This starts the Add Service wizard, displaying the Choose Services page. Some of the services are enabled by default.
3. Scroll through the alphabetic list of components on the Choose Services page, and select "Kafka".



CLUSTER INSTALL WIZARD

- Get Started
- Select Version
- Install Options
- Confirm Hosts
- **Choose Services**
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Review
- Install, Start and Test
- Summary

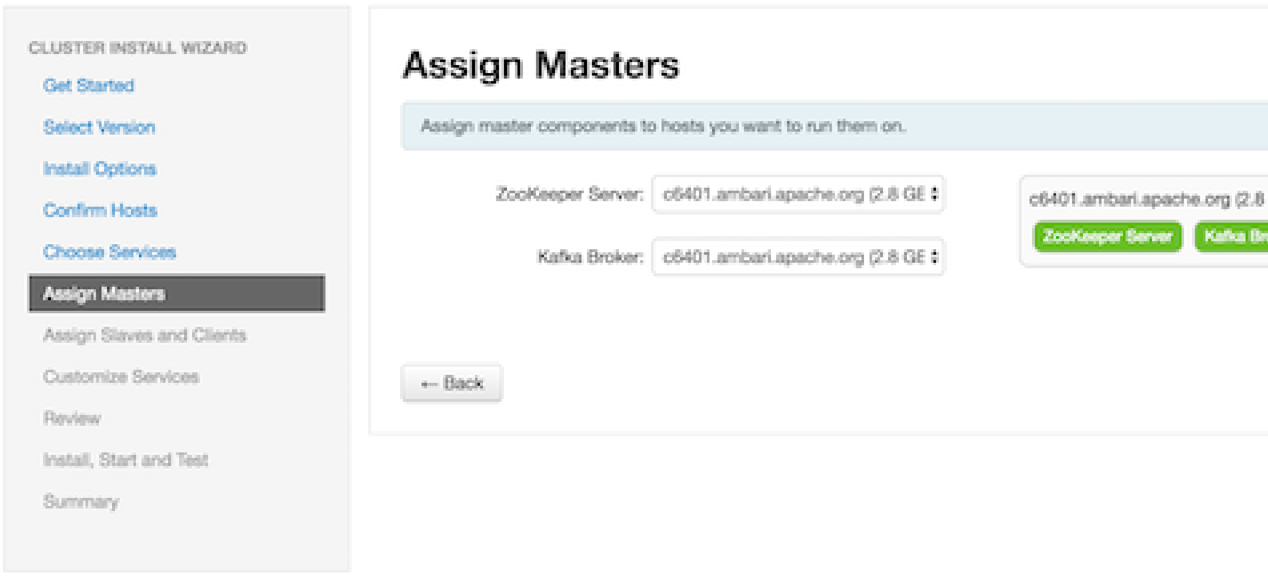
Choose Services

Choose which services you want to install on your cluster.

	Service	Version	Description
	HDFS	2.7.3	Apache Hadoop Distributed File System
	YARN + MapReduce2	2.7.3	Apache Hadoop NextGen MapReduce (YARN)
	Tez	0.7.0	Tez is the next generation Hadoop Query Processing framework
	Hive	1.2.1000	Data warehouse system for ad-hoc queries & analysis of large da storage management service
	HBase	1.1.2	A Non-relational distributed database, plus Phoenix, a high perfo low latency applications.
	Pig	0.16.0	Scripting platform for analyzing large datasets
	Sqoop	1.4.6	Tool for transferring bulk data between Apache Hadoop and stru as relational databases
	Oozie	4.2.0	System for workflow coordination and execution of Apache Hado includes the installation of the optional Oozie Web Console which the ExtJS Library.
✓	ZooKeeper	3.4.6	Centralized service which provides highly reliable distributed coo
	Falcon	0.10.0	Data management and processing platform
	Storm	1.0.1	Apache Hadoop Stream processing framework
	Flume	1.5.2	A distributed service for collecting, aggregating, and moving larg data into HDFS
	Accumulo	1.7.0	Robust, scalable, high performance distributed key/value store.
	Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieva collected from the cluster
	Atlas	0.7.0	Atlas Metadata and Governance platform
✓	Kafka	0.10.0	A high-throughput distributed messaging system
	Knox	0.9.0	Provides a single point of authentication and access for Apache cluster
	Log Search	0.5.0	Log aggregation, analysis, and visualization for Ambari managed Tech Preview.
	SmartSense	1.3.0.0-980	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly metrics, logs from common HDP services that aids to quickly tro and receive cluster-specific recommendations.
	Spark	1.6.2	Apache Spark is a fast and general engine for large-scale data p
	Spark2	2.0.0	Apache Spark 2.0 is a fast and general engine for large-scale dat service is **Technical Preview**.
	Zeppelin Notebook	0.6.0	A web-based notebook that enables interactive data analytics. It beautiful data-driven, interactive and collaborative documents wi
	Mahout	0.9.0	Project of the Apache Software Foundation to produce free imple distributed or otherwise scalable machine learning algorithms fo areas of collaborative filtering, clustering and classification
	Slider	0.91.0	A framework for deploying, managing and monitoring existing di YARN.

4. Click "Next" to continue.
5. On the Assign Masters page, review the node assignments for Kafka nodes.

   The following screen shows node assignment for a single-node Kafka cluster:



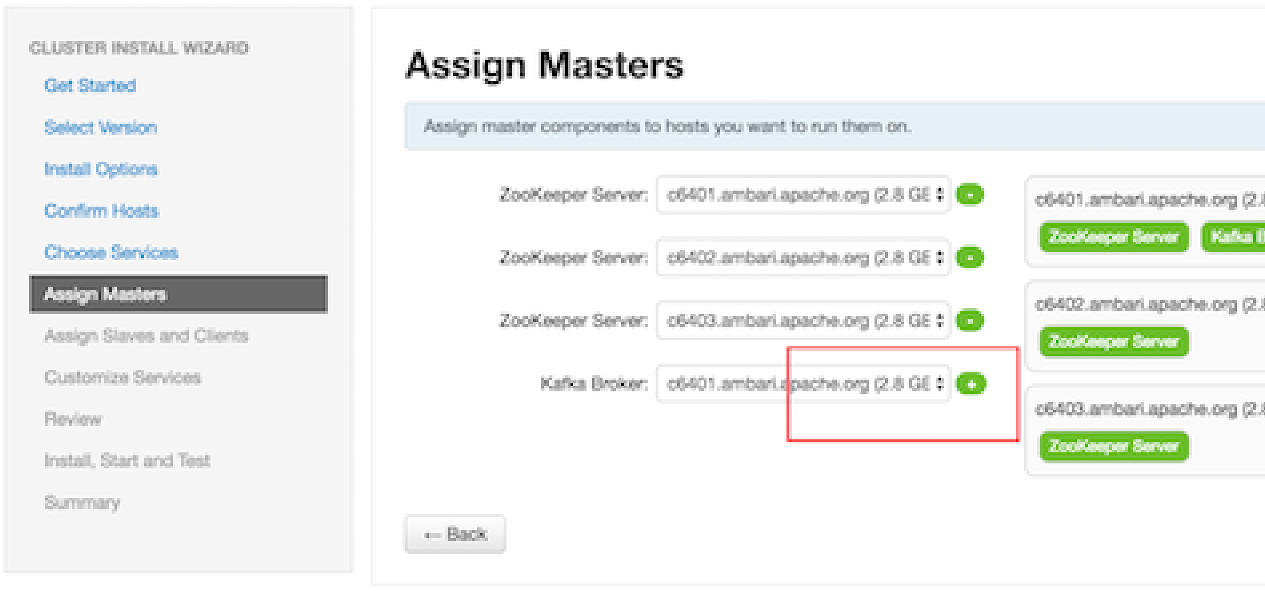
Assign master components to hosts you want to run them on.

ZooKeeper Server: c6401.ambari.apache.org (2.8 GE

Kafka Broker: c6401.ambari.apache.org (2.8 GE

6. If you want Kafka to run with high availability, you must assign more than one node for Kafka brokers, resulting in Kafka brokers running on multiple nodes.

   Click the "+" symbol to add more broker nodes to the cluster:



Assign master components to hosts you want to run them on.

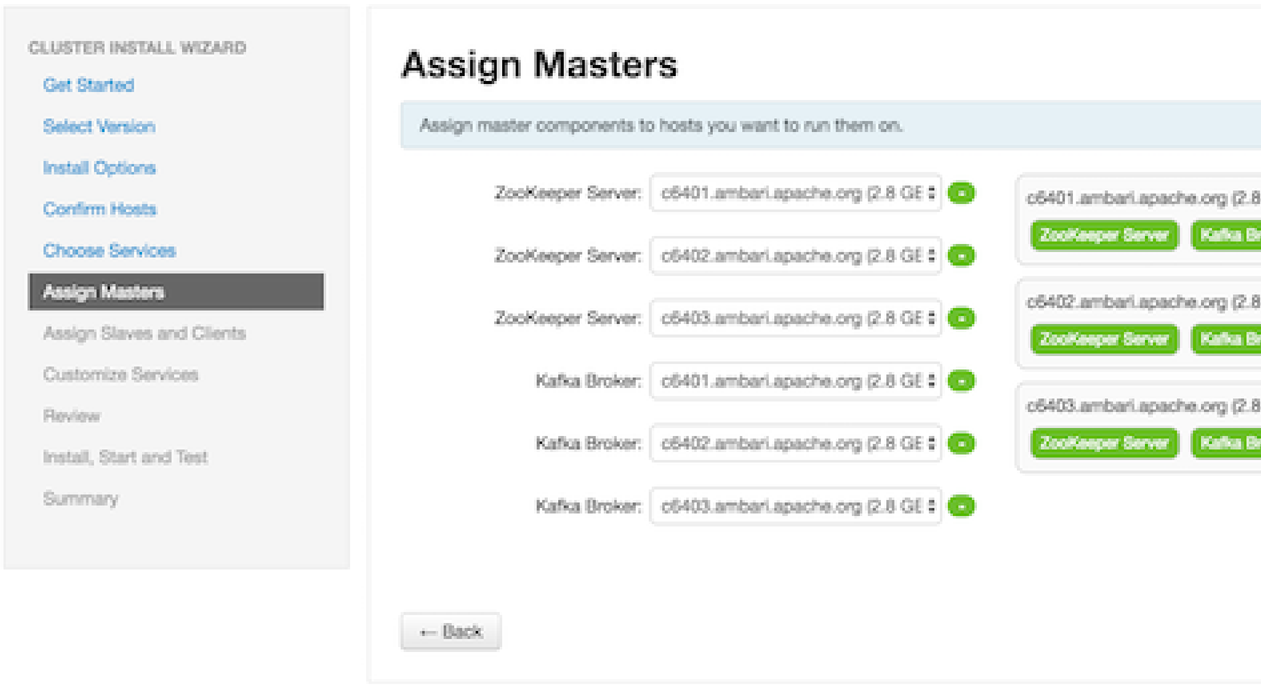
ZooKeeper Server: c6401.ambari.apache.org (2.8 GE

ZooKeeper Server: c6402.ambari.apache.org (2.8 GE

ZooKeeper Server: c6403.ambari.apache.org (2.8 GE

Kafka Broker: c6401.ambari.apache.org (2.8 GE

The following screen shows node assignment for a multi-node Kafka cluster:



CLUSTER INSTALL WIZARD

- Get Started
- Select Version
- Install Options
- Confirm Hosts
- Choose Services
- **Assign Masters**
- Assign Slaves and Clients
- Customize Services
- Review
- Install, Start and Test
- Summary

Assign Masters

Assign master components to hosts you want to run them on.

ZooKeeper Server:	c6401.ambari.apache.org (2.8 GE
ZooKeeper Server:	c6402.ambari.apache.org (2.8 GE
ZooKeeper Server:	c6403.ambari.apache.org (2.8 GE
Kafka Broker:	c6401.ambari.apache.org (2.8 GE
Kafka Broker:	c6402.ambari.apache.org (2.8 GE
Kafka Broker:	c6403.ambari.apache.org (2.8 GE

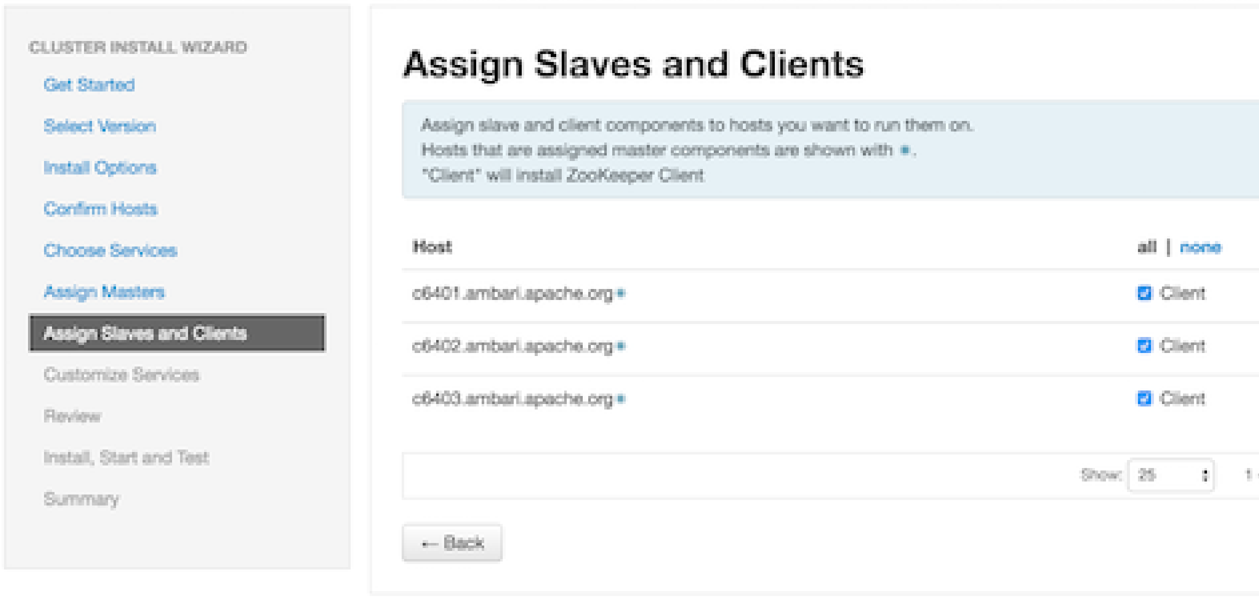
c6401.ambari.apache.org (2.8
ZooKeeper Server | Kafka Br

c6402.ambari.apache.org (2.8
ZooKeeper Server | Kafka Br

c6403.ambari.apache.org (2.8
ZooKeeper Server | Kafka Br

7. Click "Next" to continue.

8. On the Assign Slaves and Clients page, choose the nodes that you want to run ZooKeeper clients:



CLUSTER INSTALL WIZARD

- Get Started
- Select Version
- Install Options
- Confirm Hosts
- Choose Services
- Assign Masters
- **Assign Slaves and Clients**
- Customize Services
- Review
- Install, Start and Test
- Summary

Assign Slaves and Clients

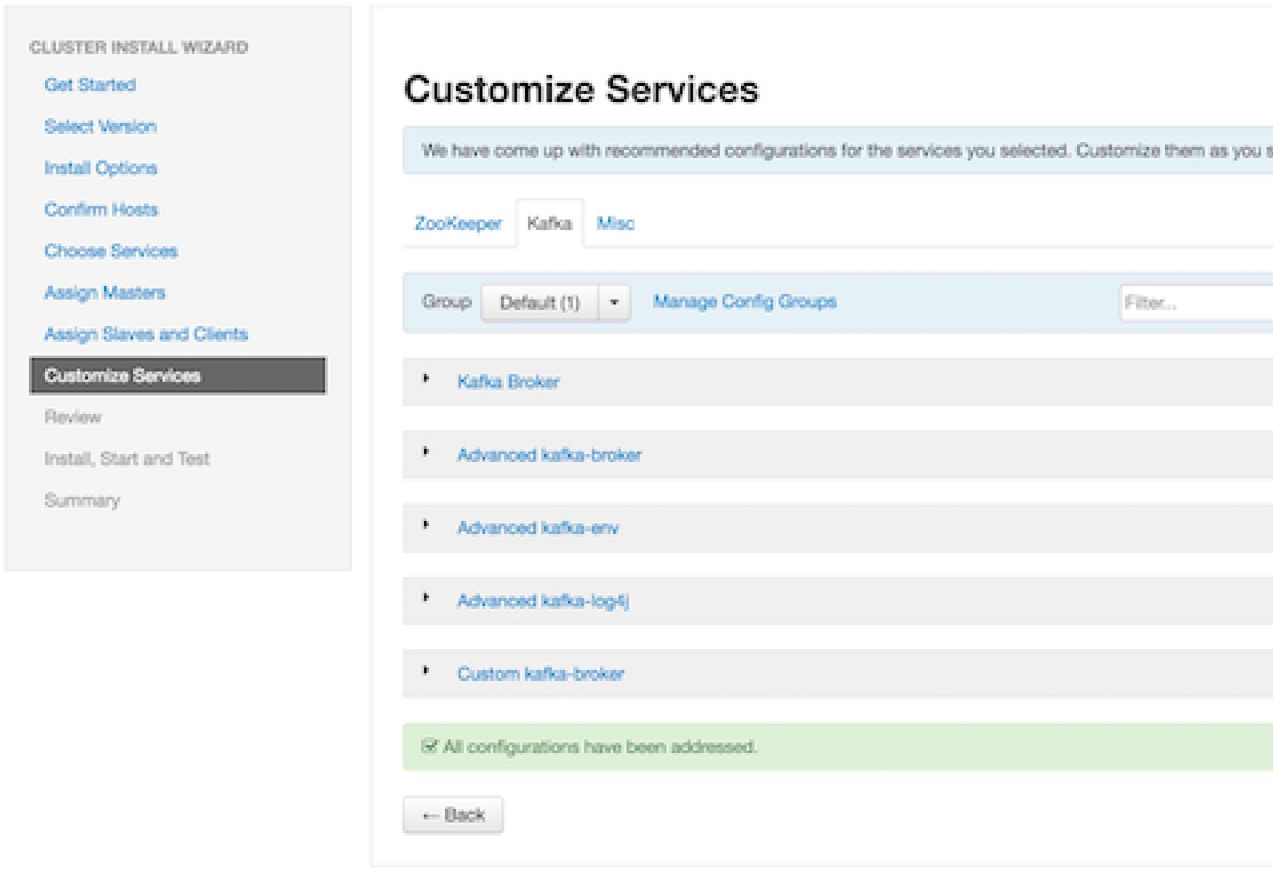
Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.
"Client" will install ZooKeeper Client

Host	all | none
c6401.ambari.apache.org✱	☑ Client
c6402.ambari.apache.org✱	☑ Client
c6403.ambari.apache.org✱	☑ Client

Show: 25

9. Click "Next" to continue.

10. Ambari displays the Customize Services page, which lists a series of services:



Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you s

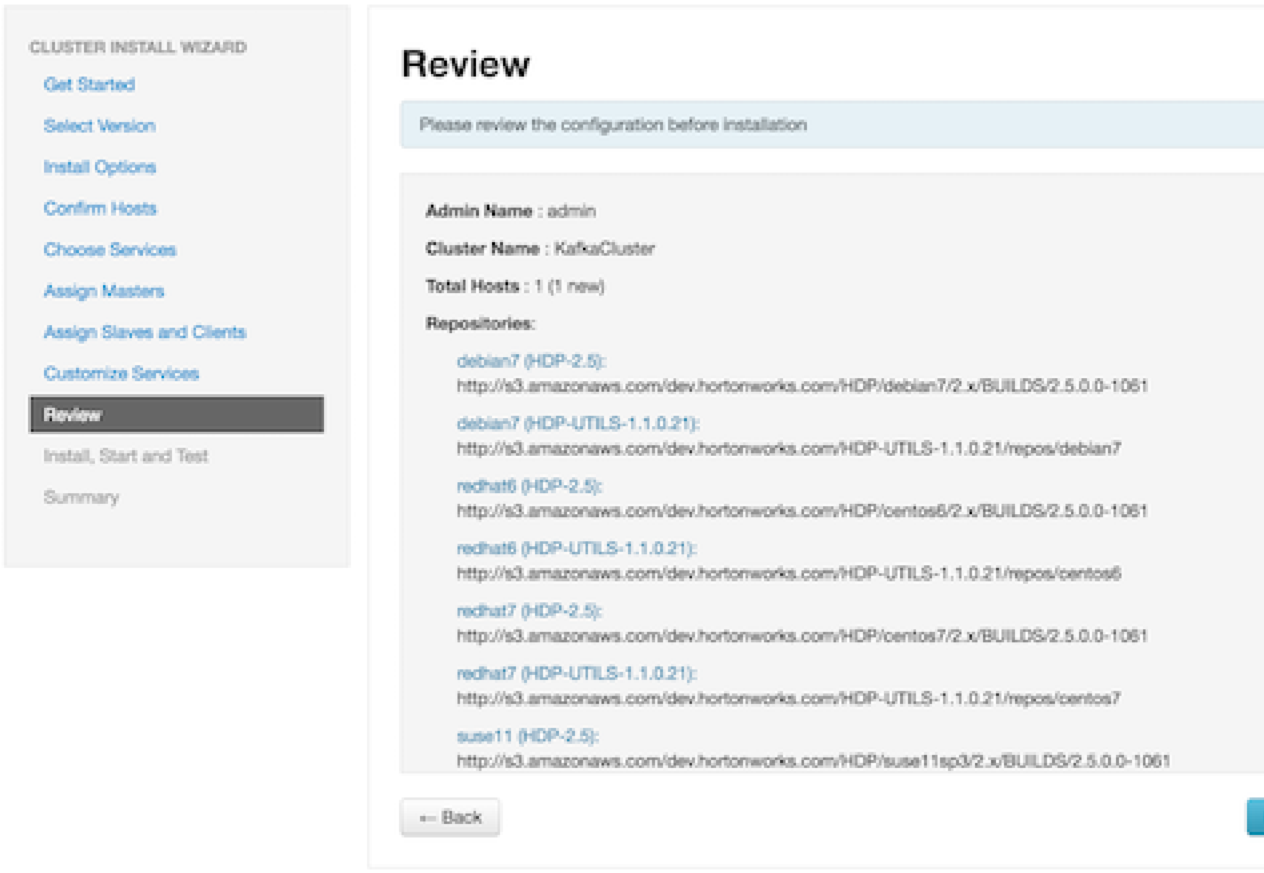
All configurations have been addressed.

For your initial configuration you should use the default values set by Ambari. If Ambari prompts you with the message "Some configurations need your attention before you can proceed," review the list of properties and provide the required information.

For information about optional settings that are useful in production environments, see Configuring Apache Kafka for a Production Environment.

11. Click "Next" to continue.

12. When the wizard displays the Review page, ensure that all HDP components correspond to HDP 2.5 or later:



Review

Please review the configuration before installation

Admin Name : admin

Cluster Name : KafkaCluster

Total Hosts : 1 (1 new)

Repositories:

debian7 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/debian7/2.x/BUILDS/2.5.0.0-1061

debian7 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/debian7

redhat6 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos6/2.x/BUILDS/2.5.0.0-1061

redhat6 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos6

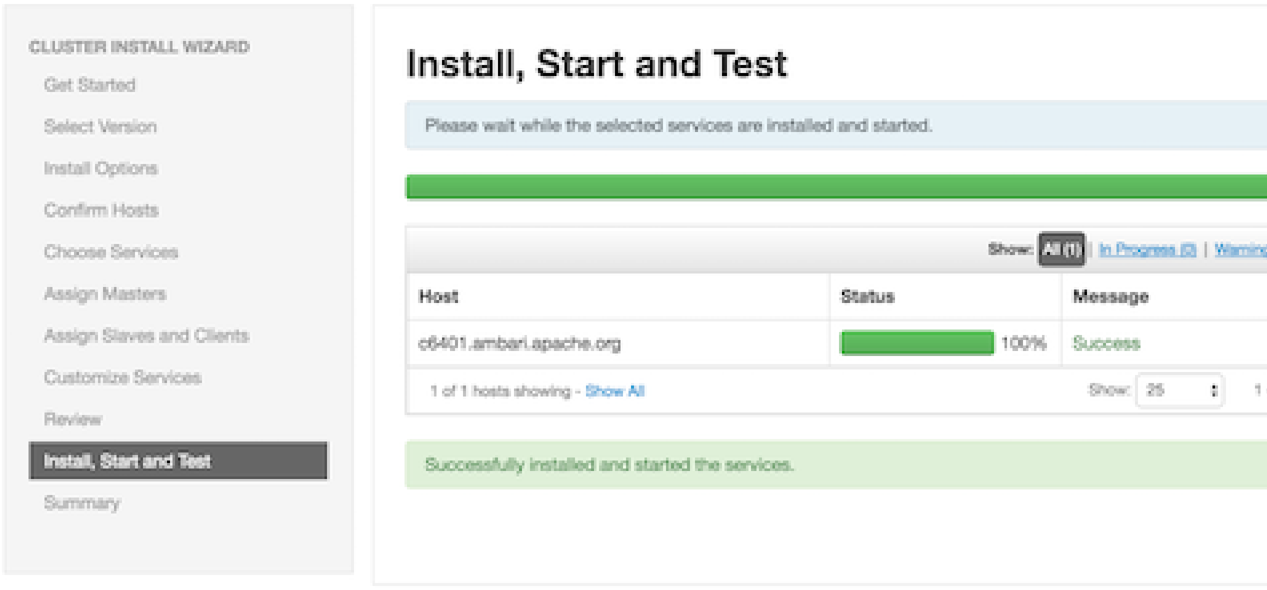
redhat7 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos7/2.x/BUILDS/2.5.0.0-1061

redhat7 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos7

suse11 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/suse11sp3/2.x/BUILDS/2.5.0.0-1061

13.Click "Deploy" to begin installation.

14.Ambari displays the Install, Start and Test page. Monitor the status bar and messages for progress updates:



Install, Start and Test

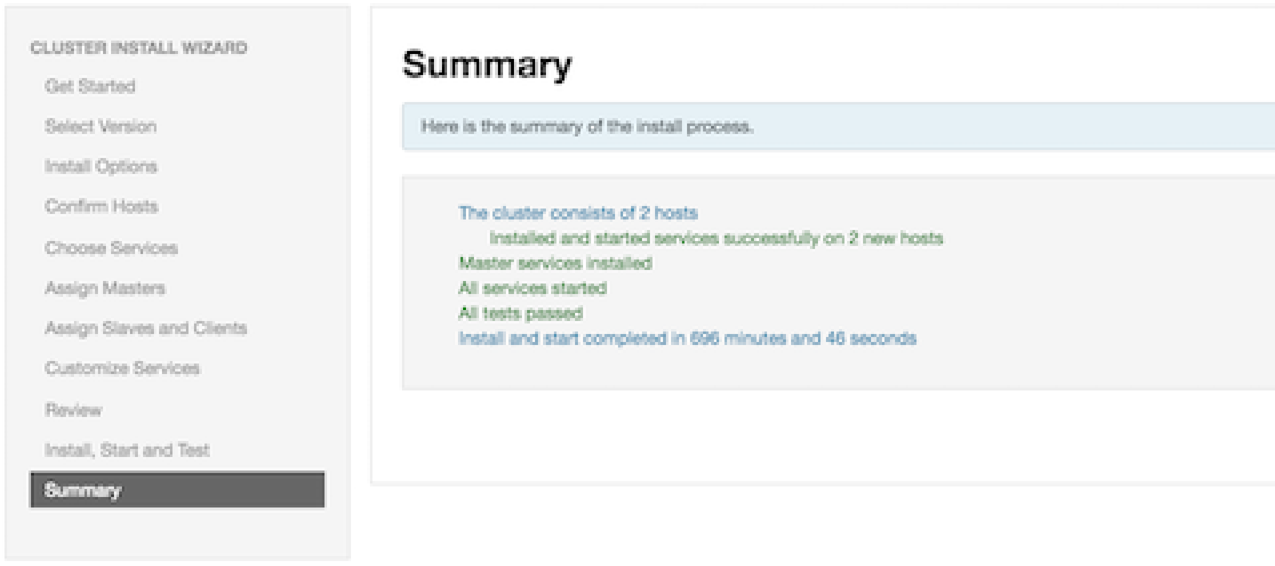
Please wait while the selected services are installed and started.

Host	Status	Message
c6401.ambari.apache.org	100%	Success

1 of 1 hosts showing - Show All   Show: 25

Successfully installed and started the services.

15. When the wizard presents a summary of results, click "Complete" to finish installing Kafka:



Summary

Here is the summary of the install process.

The cluster consists of 2 hosts
Installed and started services successfully on 2 new hosts
Master services installed
All services started
All tests passed
Install and start completed in 696 minutes and 46 seconds

After Kafka is deployed and running, validate the installation. You can use the command-line interface to create a Kafka topic, send test messages, and consume the messages. For more information, see Validate Kafka in the *Non-Ambari Cluster Installation Guide*.

5. Configuring Kafka for a Production Environment

This chapter covers topics related to Kafka configuration:

- Customizing configuration settings for Kafka in a production environment
- Configuring a ZooKeeper cluster for running multiple applications, such as Kafka and HBase
- Enabling audit to HDFS for a secure cluster

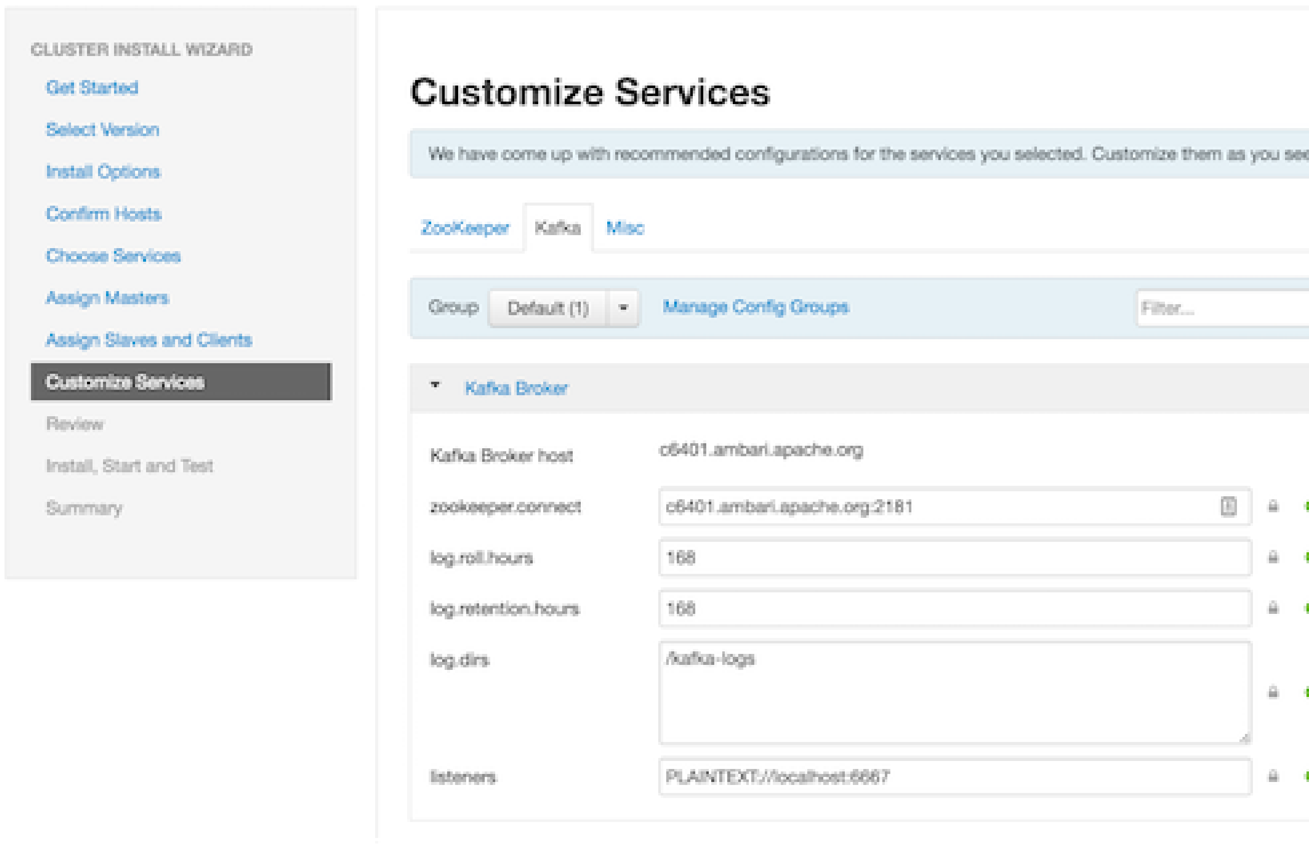
Instructions are for Ambari-managed clusters.

To configure Kafka for Kerberos security on an Ambari-managed cluster, see Configuring Kafka for Kerberos Using Ambari in the *Security Guide*.

For information about Kafka properties that are not described in this chapter, see Apache Kafka Configuration documentation.

5.1. Customizing Kafka Configuration Settings

To customize configuration settings during installation—while using the Ambari wizard—click the "Kafka" tab on the Customize Services page:



Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you see

Kafka Broker host	c6401.ambari.apache.org
zookeeper.connect	c6401.ambari.apache.org:2181
log.roll.hours	168
log.retention.hours	168
log.dirs	/kafka-logs
listeners	PLAINTEXT://localhost:6667

If you want to access configuration settings after installing Kafka using Ambari:

1. Click Kafka on the Ambari dashboard.

2. Choose Configs.

To view and modify settings, either scroll through categories and expand a category (such as "Kafka Broker", as shown in the graphic), or use the "Filter" box to search for a property.

5.1.1. Connection Settings

Review the following connection setting in the Advanced kafka-broker category, and modify as needed:

`zookeeper.session.timeout.ms`	Specifies Zookeeper session timeout, in milliseconds. The default value is 30000 ms. If the server fails to signal heartbeat to ZooKeeper within this period of time, the server is considered to be dead. If you set this value too low, the server might be falsely considered dead; if you set it too high it may take too long to recognize a truly dead server. If you see frequent disconnection from the ZooKeeper server, review this setting. If long garbage collection pauses cause Kafka to lose its ZooKeeper session, you might need to configure longer timeout values.

Important

Do not change the following connection settings:

`listeners`	A comma-separated list of URIs that Kafka will listen on, and their protocols. Ambari sets this value to the hostnames of nodes where Kafka is being installed.. Do not change this setting.
`zookeeper.connect`	A comma-separated list of ZooKeeper `hostname:port` pairs. Ambari sets this value. Do not change this setting.

5.1.2. Topic Settings

Review the following log setting in the Advanced kafka-broker category, and modify as needed:

`auto.create.topics.enable`	Enable automatic creation of topics on the server. If this property is set to true, then attempts to produce, consume, or fetch metadata for a nonexistent topic automatically create the topic with the default replication factor and number of partitions. The default is `enabled`.

`default.replication.factor`	Specifies default replication factors for automatically created topics. For high availability production systems, you should set this value to at least 3.
`num.partitions`	Specifies the default number of log partitions per topic, for automatically created topics. The default value is 1. Change this setting based on the requirements related to your topic and partition design.
`delete.topic.enable`	Enables delete topic functionality. Enable this if you want to delete topics through the admin tool. Deleting a topic through the admin tool will have no effect if this config is turned off. By default this feature is `disabled`.

5.1.3. Log Settings

Review the following settings in the Kafka Broker category, and modify as needed:

`log.roll.hours`	The maximum time, in hours, before a new log segment is rolled out. The default value is 168 hours (seven days). This setting controls the period of time after which Kafka will force the log to roll, even if the segment file is not full. This ensures that the retention process is able to delete or compact old data.
`log.retention.hours`	The number of hours to keep a log file before deleting it. The default value is 168 hours (seven days). The higher the number, the longer the data will be preserved. Higher settings generate larger log files, so increasing this setting might reduce your overall storage capacity.
`log.dirs`	A comma-separated list of directories in which log data is kept. If you have multiple disks, list all directories under each disk.

Review the following setting in the Advanced kafka-broker category, and modify as needed:

`log.retention.bytes`	The amount of data to retain in the log for each topic partition. By default, log size is unlimited. Note that this is the limit for each partition, so multiply this value by the number of partitions to calculate the total data retained for the topic. If `log.retention.hours` and `log.retention.bytes` are both set, Kafka deletes a segment when either limit is exceeded.
`log.segment.bytes`	The log for a topic partition is stored as a directory of segment files. This setting controls the maximum size of a segment file

before a new segment is rolled over in the log. The default is 1 GB.

Broker Settings

Review the following settings in the Advanced kafka-broker category, and modify as needed:

`auto.leader.rebalance.enable`	Enables automatic leader balancing. A background thread checks and triggers leader balancing (if needed) at regular intervals. The default is `enabled`.
`unclean.leader.election.enable`	This property operates as follows: • If `unclean.leader.election.enable` is set to `true` (enabled), one of the out-of-sync replicas will be elected as leader when there is no live in-sync replica (ISR). This will lead to data loss. • If `unclean.leader.election.enable` is set to `false` and there are no live in-sync replicas (ISR), Kafka returns an error and the partition will be unavailable. This property is enabled by default. This is an important setting: If availability is more important than avoiding data loss, ensure that this property is set to true. If preventing data loss is more important than availability, set this to false.
`controlled.shutdown.enable`	Enables controlled shutdown of the server. The default is `enabled`.
`min.insync.replicas`	When a producer sets `acks` to "all", `min.insync.replicas` specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception. When used together, `min.insync.replicas` and producer `acks` allow you to enforce stronger durability guarantees. You should set `min.insync.replicas` to 2 for replication factor equal to 3.
`message.max.bytes`	Specifies the maximum size of message that the server can receive. It is important that this property be set with consideration for the maximum fetch size used by your consumers, or a producer could publish messages too large for consumers to consume.

	Note that there are currently two versions of consumer and producer APIs. The value of `message.max.bytes` must be smaller than the `max.partition.fetch.byte` setting in the new consumer, or smaller than the `fetch.message.max.bytes` setting in the old consumer.
`replica.fetch.max.bytes`	Specifies the number of bytes of messages to attempt to fetch. This value must be larger than `message.max.bytes`.
`broker.rack`	The rack awareness feature distributes replicas of a partition across different racks. You can specify that a broker belongs to a particular rack through the "Custom kafka-broker" menu option. For more information about the rack awareness feature, see http://kafka.apache.org/documentation.html#basic_ops_racks.

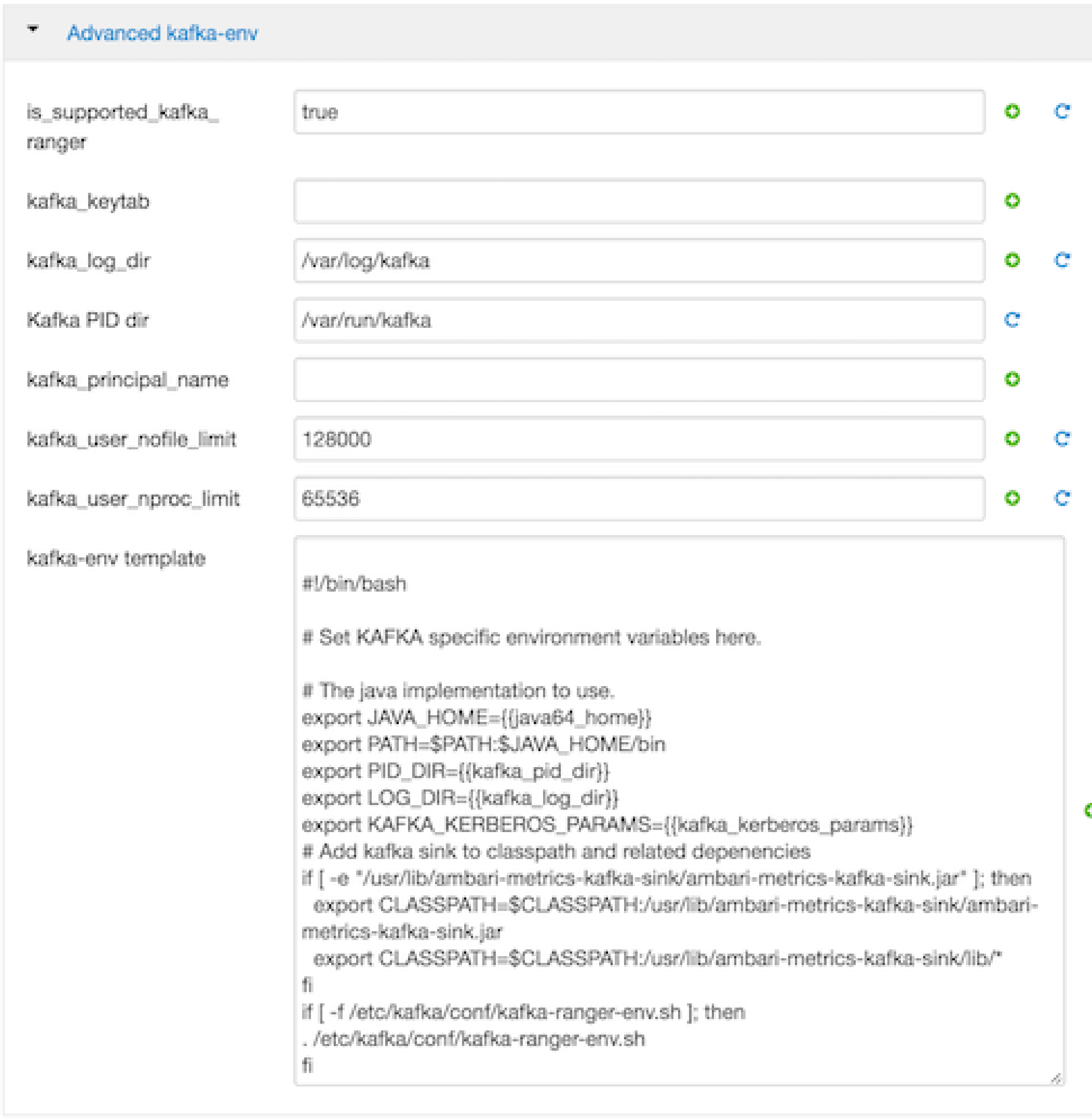
5.1.4. Compaction Settings

Review the following settings in the Advanced kafka-broker category, and modify as needed:

`log.cleaner.dedupe.buffer.size`	Specifies total memory used for log deduplication across all cleaner threads. By default, 128 MB of buffer is allocated. You may want to review this and other `log.cleaner` configuration values, and adjust settings based on your use of compacted topics (`__consumer_offsets` and other compacted topics).
`log.cleaner.io.buffer.size`	Specifies the total memory used for log cleaner I/O buffers across all cleaner threads. By default, 512 KB of buffer is allocated. You may want to review this and other `log.cleaner` configuration values, and adjust settings based on your usage of compacted topics (`__consumer_offsets` and other compacted topics).

5.1.5. Advanced kafka-env Settings

Advanced kafka-env environment settings are configured by Ambari; you should not modify these settings:



is_supported_kafka_ranger	true
kafka_keytab	
kafka_log_dir	/var/log/kafka
Kafka PID dir	/var/run/kafka
kafka_principal_name	
kafka_user_nofile_limit	128000
kafka_user_nproc_limit	65536

kafka-env template

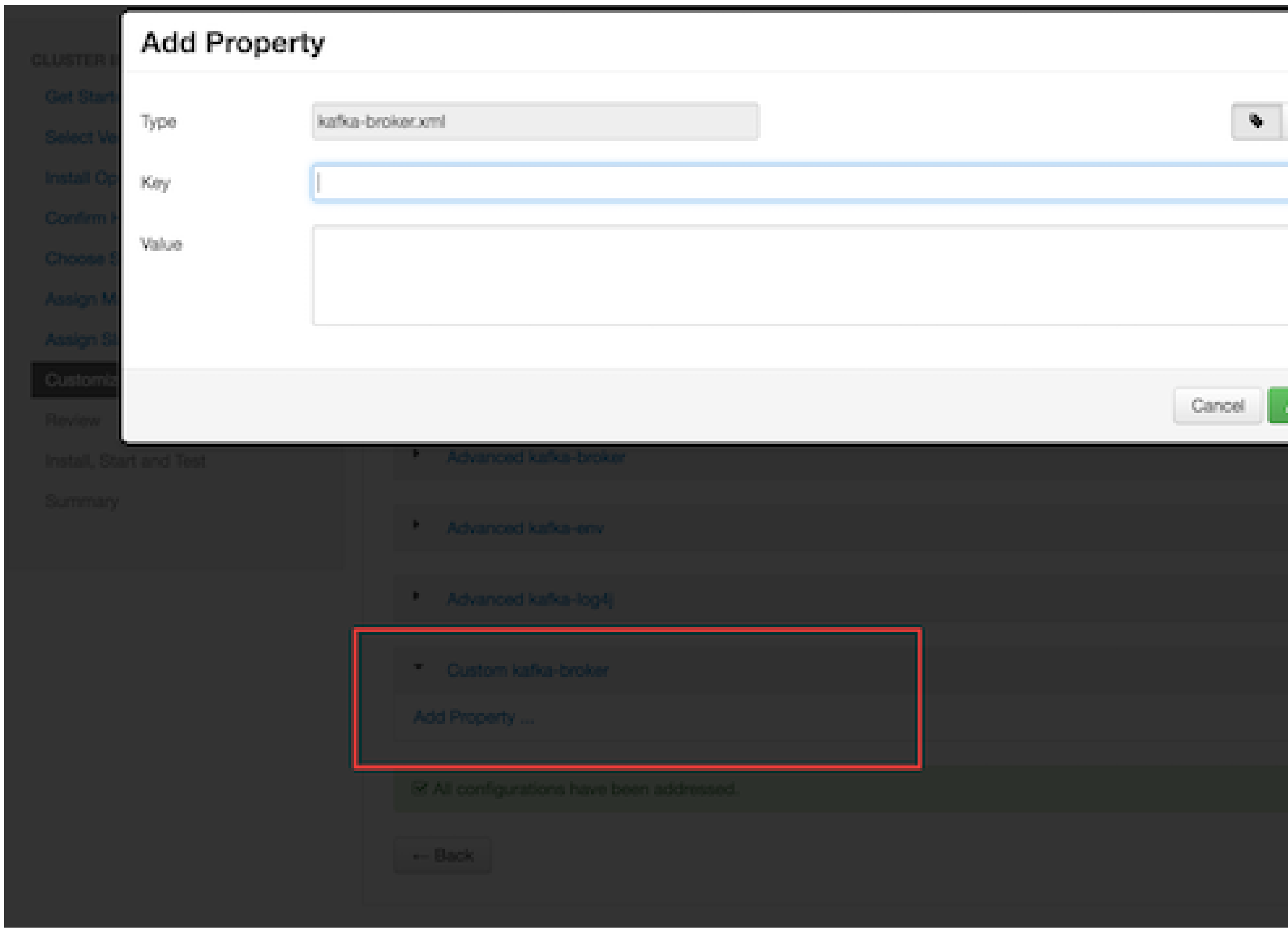
```
#!/bin/bash

# Set KAFKA specific environment variables here.

# The java implementation to use.
export JAVA_HOME={{java64_home}}
export PATH=$PATH:$JAVA_HOME/bin
export PID_DIR={{kafka_pid_dir}}
export LOG_DIR={{kafka_log_dir}}
export KAFKA_KERBEROS_PARAMS={{kafka_kerberos_params}}
# Add kafka sink to classpath and related depenencies
if [ -e "/usr/lib/ambari-metrics-kafka-sink/ambari-metrics-kafka-sink.jar" ]; then
  export CLASSPATH=$CLASSPATH:/usr/lib/ambari-metrics-kafka-sink/ambari-metrics-kafka-sink.jar
  export CLASSPATH=$CLASSPATH:/usr/lib/ambari-metrics-kafka-sink/lib/*
fi
if [ -f /etc/kafka/conf/kafka-ranger-env.sh ]; then
. /etc/kafka/conf/kafka-ranger-env.sh
fi
```

5.1.6. Adding Configuration Properties

You can add other configuration properties by navigating to the Custom kafka-broker category:



5.2. Configuring ZooKeeper for Multiple Applications

If you plan to use the same ZooKeeper cluster for different applications (such as Kafka cluster1, Kafka cluster2, HBase, and so on), you should add a `chroot` path so that all Kafka data for a cluster appears under a specific path. Here is an sample definition:

```
c6401.ambari.apache.org:2181:/kafka-root,
c6402.ambari.apache.org:2181:/kafka-root
```

You must create this `chroot` path yourself before starting the broker, and consumers must use the same connection string.

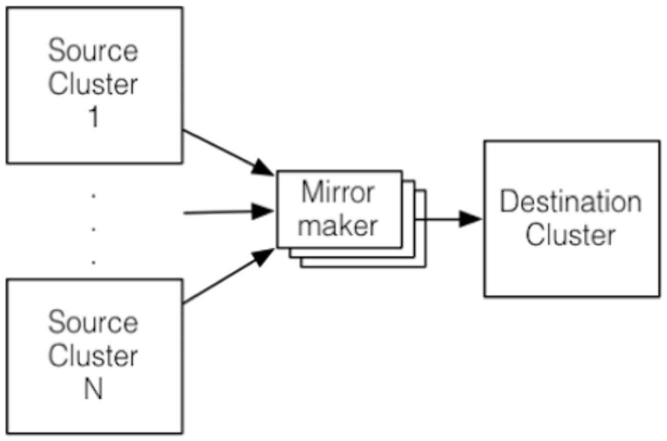
5.3. Enabling Audit to HDFS for a Secure Cluster

To enable audit to HDFS when running Storm on a secure cluster, perform the steps listed at the bottom of Manually Updating Ambari HDFS Audit Settings in the *HDP Security Guide*.

6. Mirroring Data Between Clusters: Using the MirrorMaker Tool

The process of replicating data between Kafka clusters is called "mirroring", to differentiate cross-cluster replication from replication among nodes within a single cluster. A common use for mirroring is to maintain a separate copy of a Kafka cluster in another data center.

Kafka's MirrorMaker tool reads data from topics in one or more source Kafka clusters, and writes corresponding topics to a destination Kafka cluster (using the same topic names):



To mirror more than one source cluster, start at least one MirrorMaker instance for each source cluster.

You can also use multiple MirrorMaker processes to mirror topics within the same consumer group. This can increase throughput and enhance fault-tolerance: if one process dies, the others will take over the additional load.

The source and destination clusters are completely independent, so they can have different numbers of partitions and different offsets. The destination (mirror) cluster is not intended to be a mechanism for fault-tolerance, because the consumer position will be different. (The MirrorMaker process will, however, retain and use the message key for partitioning, preserving order on a per-key basis.) For fault tolerance we recommend using standard within-cluster replication.

6.1. Running MirrorMaker

Prerequisite: The source and destination clusters must be deployed and running.

To set up a mirror, run `kafka.tools.MirrorMaker`. The following table lists configuration options.

At a minimum, MirrorMaker requires one or more consumer configuration files, a producer configuration file, and either a whitelist or a blacklist of topics. In the consumer and producer configuration files, point the consumer to the ZooKeeper process on the source cluster, and point the producer to the ZooKeeper process on the destination (mirror) cluster, respectively.

Table 6.1. MirrorMaker Options

Parameter	Description	Examples
`--consumer.config`	Specifies a file that contains configuration settings for the source cluster. For more information about this file, see the "Consumer Configuration File" subsection.	`--consumer.config hdp1-consumer.properties`
`--producer.config`	Specifies the file that contains configuration settings for the target cluster. For more information about this file, see the "Producer Configuration File" subsection.	`--producer.config hdp1-producer.properties`
`--whitelist` `--blacklist`	(Optional) For a partial mirror, you can specify exactly one comma-separated list of topics to include (–whitelist) or exclude (–blacklist). In general, these options accept Java regex patterns. For caveats, see the note after this table.	`--whitelist my-topic`
`--num.streams`	Specifies the number of consumer stream threads to create.	`--num.streams 4`
`--num.producers`	Specifies the number of producer instances. Setting this to a value greater than one establishes a producer pool that can increase throughput.	`--num.producers 2`
`--queue.size`	Queue size: number of messages that are buffered, in terms of number of messages between the consumer and producer. Default = 10000.	`--queue.size 2000`
`--help`	List MirrorMaker command-line options.	

Note

- A comma (',') is interpreted as the regex-choice symbol ('|') for convenience.
- If you specify `--white-list=".*"`, MirrorMaker tries to fetch data from the system-level topic `__consumer-offsets` and produce that data to the target cluster. This can result in the following error:

  ```
  Producer cannot send requests to __consumer-offsets
  ```

  Workaround: Specify topic names, or to replicate all topics, specify `--blacklist="__consumer-offsets"`.

The following example replicates `topic1` and `topic2` from `sourceClusterConsumer` to `targetClusterProducer`:

```
/usr/hdp/current/kafka-broker/bin/kafka-run-class.sh kafka.tools.MirrorMaker
 --consumer.config sourceClusterConsumer.properties  --producer.config
 targetClusterProducer.properties --whitelist="topic1, topic"
```

Consumer Configuration File

The consumer configuration file must specify the ZooKeeper process in the source cluster.

Here is a sample consumer configuration file:

```
zk.connect=hdp1:2181/kafka
zk.connectiontimeout.ms=1000000
consumer.timeout.ms=-1
groupid=dp-MirrorMaker-test-datap1
shallow.iterator.enable=true
mirror.topics.whitelist=app_log
```

Producer Configuration File

The producer configuration should point to the target cluster's ZooKeeper process (or use the broker.list parameter to specify a list of brokers on the destination cluster).

Here is a sample producer configuration file:

```
zk.connect=hdp1:2181/kafka-test
producer.type=async
compression.codec=0
serializer.class=kafka.serializer.DefaultEncoder
max.message.size=10000000
queue.time=1000
queue.enqueueTimeout.ms=-1
```

6.2. Checking Mirroring Progress

You can use Kafka's Consumer Offset Checker command-line tool to assess how well your mirror is keeping up with the source cluster. The Consumer Offset Checker checks the number of messages read and written, and reports the lag for each consumer in a specified consumer group.

The following command runs the Consumer Offset Checker for group `KafkaMirror`, topic `test-topic`. The `--zkconnect` argument points to the ZooKeeper host and port on the source cluster.

```
/usr/hdp/current/kafka/bin/kafka-run-class.sh kafka.tools.
ConsumerOffsetChecker --group KafkaMirror --zkconnect source-cluster-
zookeeper:2181 --topic test-topic

Group        Topic          Pid Offset     logSize     Lag     Owner
KafkaMirror  test-topic     0   5          5           0       none
KafkaMirror  test-topic     1   3          4           1       none
KafkaMirror  test-topic     2   6          9           3       none
```

Table 6.2. Consumer Offset Checker Options

`--group`	(Required) Specifies the consumer group.
`--zkconnect`	Specifies the ZooKeeper connect string. The default is `localhost:2181`.
`--broker-info`	Lists broker information
`--help`	Lists offset checker options.

`--topic`	Specifies a comma-separated list of consumer topics. If you do not specify a topic, the offset checker will display information for all topics under the given consumer group.

6.3. Avoiding Data Loss

If for some reason the producer cannot deliver messages that have been consumed and committed by the consumer, it is possible for a MirrorMaker process to lose data.

To prevent data loss, use the following settings. (Note: these are the default settings.)

- For consumers:
  - `auto.commit.enabled=false`
- For producers:
  - `max.in.flight.requests.per.connection=1`
  - `retries=Int.MaxValue`
  - `acks=-1`
  - `block.on.buffer.full=true`
- Specify the `--abortOnSendFail` option to MirrorMaker

The following actions will be taken by MirrorMaker:

- MirrorMaker will send only one request to a broker at any given point.
- If any exception is caught in the MirrorMaker thread, MirrorMaker will try to commit the acked offsets and then exit immediately.
- On a `RetriableException` in the producer, the producer will retry indefinitely. If the retry does not work, MirrorMaker will eventually halt when the producer buffer is full.
- On a non-retriable exception, if `--abort.on.send.fail` is specified, MirrorMaker will stop.

  If `--abort.on.send.fail` is not specified, the producer callback mechanism will record the message that was not sent, and MirrorMaker will continue running. In this case, the message will not be replicated in the target cluster.

6.4. Running MirrorMaker on Kerberos-Enabled Clusters

To run MirrorMaker on a Kerberos/SASL-enabled cluster, configure producer and consumer properties as follows:

1. Choose or add a new principal for MirrorMaker. Do not use `kafka` or any other service accounts. The following example uses principal `mirrormaker`.

2. Create client-side Kerberos keytabs for your MirrorMaker principal. For example:

   ```
   sudo kadmin.local -q "ktadd -k /tmp/mirrormaker.keytab mirrormaker/
   HOSTNAME@EXAMPLE.COM"
   ```

3. Add a new Jaas configuration file to the node where you plan to run MirrorMaker:

   ```
   -Djava.security.auth.login.config=/usr/hdp/current/kafka-broker/config/
   kafka_mirrormaker_jaas.conf
   ```

4. Add the following settings to the KafkaClient section of the new Jaas configuration file. Make sure the principal has permissions on both the source cluster and the target cluster.

   ```
   KafkaClient {
        com.sun.security.auth.module.Krb5LoginModule required
        useKeyTab=true
        keyTab="/tmp/mirrormaker.keytab"
        storeKey=true
        useTicketCache=false
        serviceName="kafka"
        principal="mirrormaker/HOSTNAME@EXAMPLE.COM";
       };
   ```

5. Run the following ACL command on the source and destination Kafka clusters:

   ```
   bin/kafka-acls.sh --topic test-topic --add --allow-principal
    user:mirrormaker --operation ALL --config /usr/hdp/current/kafka-broker/
   config/server.properties
   ```

6. In your MirrorMaker `consumer.config` and `producer.config` files, specify `security.protocol=SASL_PLAINTEXT`.

7. Start MirrorMaker. Specify the `new.consumer` option in addition to your other options. For example:

   ```
   /usr/hdp/current/kafka-broker/bin/kafka-run-class.sh kafka.tools.MirrorMaker
    --consumer.config consumer.properties --producer.config target-cluster-
   producer.properties --whitelist my-topic --new.consumer
   ```

7. Developing Kafka Producers and Consumers

The examples in this chapter contain code for a basic Kafka producer and consumer, and similar examples for an SSL-enabled cluster. (To configure Kafka for SSL, see Enable SSL for Kafka Clients in the *HDP Security Guide*.)

For examples of Kafka producers and consumers that run on a Kerberos-enabled cluster, see Producing Events/Messages to Kafka on a Secured Cluster and Consuming Events/ Messages from Kafka on a Secured Cluster, in the *Security Guide*.

Basic Producer Example

```
package com.hortonworks.example.kafka.producer;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import java.util.Properties;
import java.util.Random;

public class BasicProducerExample {

   public static void main(String[] args){

       Properties props = new Properties();
       props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");
       props.put(ProducerConfig.ACKS_CONFIG, "all");
       props.put(ProducerConfig.RETRIES_CONFIG, 0);
       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");
       props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");

       Producer<String, String> producer = new KafkaProducer<String,
 String>(props);
       TestCallback callback = new TestCallback();
       Random rnd = new Random();
       for (long i = 0; i < 100 ; i++) {
           ProducerRecord<String, String> data = new ProducerRecord<String,
 String>(
                   "test-topic", "key-" + i, "message-"+i );
           producer.send(data, callback);
       }

       producer.close();
   }


   private static class TestCallback implements Callback {
       @Override
       public void onCompletion(RecordMetadata recordMetadata, Exception e) {
           if (e != null) {
               System.out.println("Error while producing message to topic :" +
 recordMetadata);
               e.printStackTrace();
           } else {
               String message = String.format("sent message to topic:%s
 partition:%s  offset:%s", recordMetadata.topic(), recordMetadata.partition(),
 recordMetadata.offset());
               System.out.println(message);
           }
       }
   }

}
```

To run the producer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```

Producer Example for an SSL-Enabled Cluster

The following example adds three important configuration settings for SSL encryption and three for SSL authentication. The two sets of configuration settings are prefaced by comments.

```
package com.hortonworks.example.kafka.producer;

import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.config.SslConfigs;

import java.util.Properties;
import java.util.Random;

public class BasicProducerExample {

   public static void main(String[] args){

       Properties props = new Properties();
       props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");

       //configure the following three settings for SSL Encryption
       props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
       props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.truststore.jks");
       props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,  "test1234");

       // configure the following three settings for SSL Authentication
       props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.keystore.jks");
       props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "test1234");
       props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "test1234");
       props.put(ProducerConfig.ACKS_CONFIG, "all");
       props.put(ProducerConfig.RETRIES_CONFIG, 0);
       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");
       props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringSerializer");

       Producer<String, String> producer = new KafkaProducer<String,
 String>(props);
       TestCallback callback = new TestCallback();
       Random rnd = new Random();
       for (long i = 0; i < 100 ; i++) {
           ProducerRecord<String, String> data = new ProducerRecord<String,
 String>(
                   "test-topic", "key-" + i, "message-"+i );
           producer.send(data, callback);
       }

       producer.close();
   }


   private static class TestCallback implements Callback {
       @Override
       public void onCompletion(RecordMetadata recordMetadata, Exception e) {
           if (e != null) {
               System.out.println("Error while producing message to topic :" +
 recordMetadata);
               e.printStackTrace();
           } else {
               String message = String.format("sent message to topic:%s
 partition:%s  offset:%s", recordMetadata.topic(), recordMetadata.partition(),
 recordMetadata.offset());
               System.out.println(message);
           }
       }
   }

}
```

To run the producer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```

Basic Consumer Example

```
package com.hortonworks.example.kafka.consumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;

import java.util.Collection;
import java.util.Collections;
import java.util.Properties;

public class BasicConsumerExample {
   public static void main(String[] args) {

       Properties consumerConfig = new Properties();
       consumerConfig.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.
example.com:6667");
       consumerConfig.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
       consumerConfig.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
 "earliest");
       consumerConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 "org.apache.kafka.common.serialization.StringDeserializer");
       consumerConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.
apache.kafka.common.serialization.StringDeserializer");
       KafkaConsumer<byte[], byte[]> consumer = new
 KafkaConsumer<>(consumerConfig);
       TestConsumerRebalanceListener rebalanceListener = new
 TestConsumerRebalanceListener();
       consumer.subscribe(Collections.singletonList("test-topic"),
 rebalanceListener);

       while (true) {
           ConsumerRecords<byte[], byte[]> records = consumer.poll(1000);
           for (ConsumerRecord<byte[], byte[]> record : records) {
               System.out.printf("Received Message topic =%s, partition =%s,
 offset = %d, key = %s, value = %s\n", record.topic(), record.partition(),
 record.offset(), record.key(), record.value());
           }

           consumer.commitSync();
       }

   }

   private static class  TestConsumerRebalanceListener implements
 ConsumerRebalanceListener {
       @Override
       public void onPartitionsRevoked(Collection<TopicPartition> partitions)
 {
           System.out.println("Called onPartitionsRevoked with partitions:" +
 partitions);
       }

       @Override
       public void onPartitionsAssigned(Collection<TopicPartition> partitions)
 {
           System.out.println("Called onPartitionsAssigned with partitions:" +
 partitions);
       }
   }

}
```

To run the consumer example, use the following command:

```
# java com.hortonworks.example.kafka.consumer.BasicConsumerExample
```

Consumer Example for an SSL-Enabled Cluster

The following example adds three important configuration settings for SSL encryption and three for SSL authentication. The two sets of configuration settings are prefaced by comments.

```
package com.hortonworks.example.kafka.consumer;

import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRebalanceListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.config.SslConfigs;

import java.util.Collection;
import java.util.Collections;
import java.util.Properties;

public class BasicConsumerExample {

   public static void main(String[] args) {

       Properties props = new Properties();
       props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka.example.
com:6667");

       //configure the following three settings for SSL Encryption
       props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
       props.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.truststore.jks");
       props.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,  "test1234");

       //configure the following three settings for SSL Authentication
       props.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, "/var/private/ssl/
kafka.client.keystore.jks");
       props.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "test1234");
       props.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG, "test1234");

       props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
       props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
       props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringDeserializer");
       props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.
kafka.common.serialization.StringDeserializer");
       KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props);
       TestConsumerRebalanceListener rebalanceListener = new
 TestConsumerRebalanceListener();
       consumer.subscribe(Collections.singletonList("test-topic"),
 rebalanceListener);

       while (true) {
           ConsumerRecords<byte[], byte[]> records = consumer.poll(1000);
           for (ConsumerRecord<byte[], byte[]> record : records) {
               System.out.printf("Received Message topic =%s, partition =%s,
 offset = %d, key = %s, value = %s\n", record.topic(), record.partition(),
 record.offset(), record.key(), record.value());
           }

           consumer.commitSync();
```

```
        }

    }

    private static class  TestConsumerRebalanceListener implements
 ConsumerRebalanceListener {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> partitions)
 {
            System.out.println("Called onPartitionsRevoked with partitions:" +
 partitions);
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> partitions)
 {
            System.out.println("Called onPartitionsAssigned with partitions:" +
 partitions);
        }
    }

}
```

To run the consumer example, use the following command:

```
$ java com.hortonworks.example.kafka.producer.BasicProducerExample
```