Hortonworks Data Platform

Apache Kafka Component Guide

(April 20, 2017)

docs.cloudera.com

Hortonworks Data Platform: Apache Kafka Component Guide

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.

Table of Contents

1. Building a High-Throughput Messaging System with Apache Kafka 1
2. What's New 2
    2.1. Apache Kafka 2
    2.2. Content Updates 3
3. Apache Kafka Concepts 4
4. Installing Kafka 6
    4.1. Prerequisites 6
    4.2. Installing Kafka Using Ambari 6
5. Configuring Kafka for a Production Environment 13
    5.1. Preparing the Environment 13
        5.1.1. Operating System Settings 13
        5.1.2. File System Selection 13
        5.1.3. Disk Drive Considerations 14
        5.1.4. Java Version 14
        5.1.5. Ethernet Bandwidth 15
    5.2. Customizing Kafka Settings on an Ambari-Managed Cluster 15
    5.3. Kafka Broker Settings 17
        5.3.1. Connection Settings 17
        5.3.2. Topic Settings 18
        5.3.3. Log Settings 19
        5.3.4. Compaction Settings 21
        5.3.5. General Broker Settings 21
    5.4. Kafka Producer Settings 23
        5.4.1. Important Producer Settings 23
    5.5. Kafka Consumer Settings 25
    5.6. Configuring ZooKeeper for Use with Kafka 25
    5.7. Enabling Audit to HDFS for a Secure Cluster 26
6. Mirroring Data Between Clusters: Using the MirrorMaker Tool 27
    6.1. Running MirrorMaker 27
    6.2. Checking Mirroring Progress 29
    6.3. Avoiding Data Loss 30
    6.4. Running MirrorMaker on Kerberos-Enabled Clusters 30
7. Creating a Kafka Topic 32
8. Developing Kafka Producers and Consumers 33

List of Tables

6.1. MirrorMaker Options 28
6.2. Consumer Offset Checker Options 29

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:

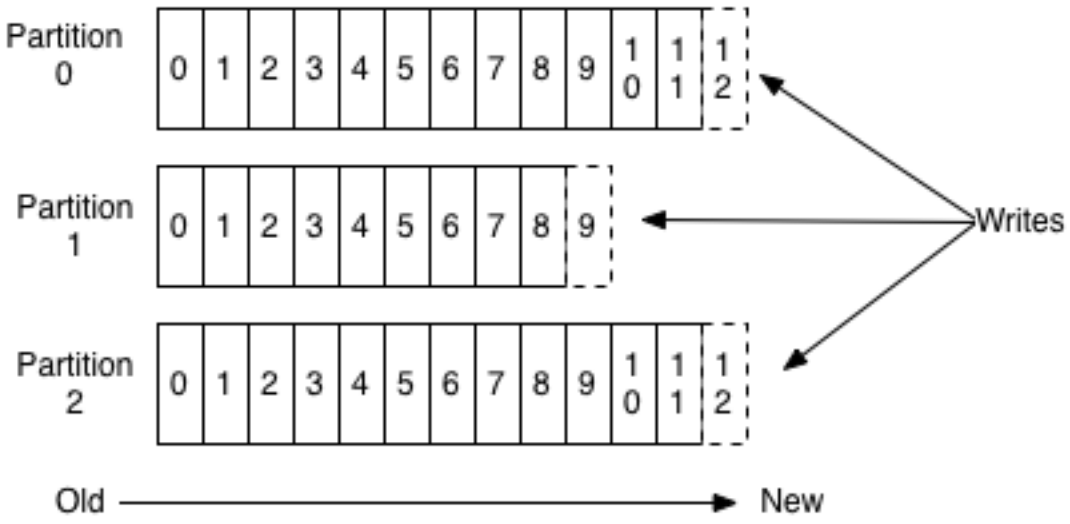
- Topics
- Producers
- Consumers
- Brokers

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.

For each topic, the Kafka cluster maintains a structured commit log with 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 for topics provides parallelism. In addition, because writes to a partition are sequential, the number of hard disk seeks is minimized. This reduces latency and increases performance.

Producers

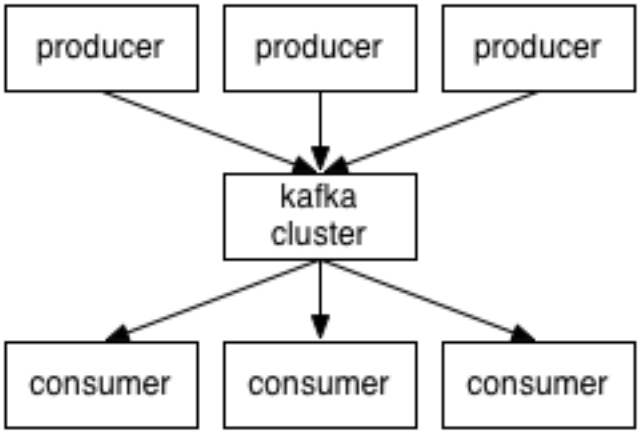
Producers are processes that publish messages to one or more Kafka topics. The producer is responsible for choosing which message to assign to which partition within a topic. Assignment can be done in a round-robin fashion to balance load, or it can be based on a semantic partition function.

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

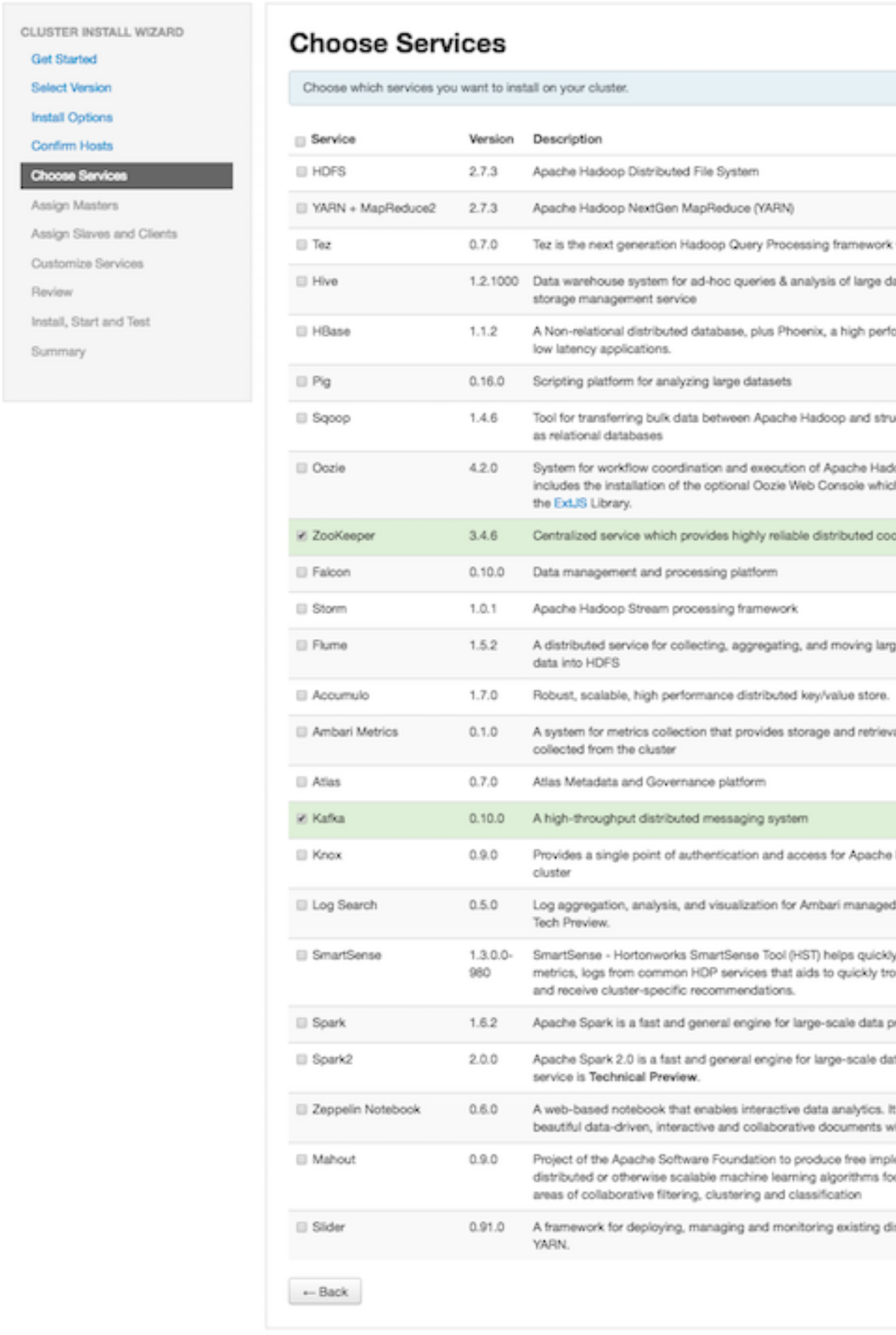
Encrypted file systems such as SafenetFS are not supported for Kafka. Index file corruption can occur.

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.

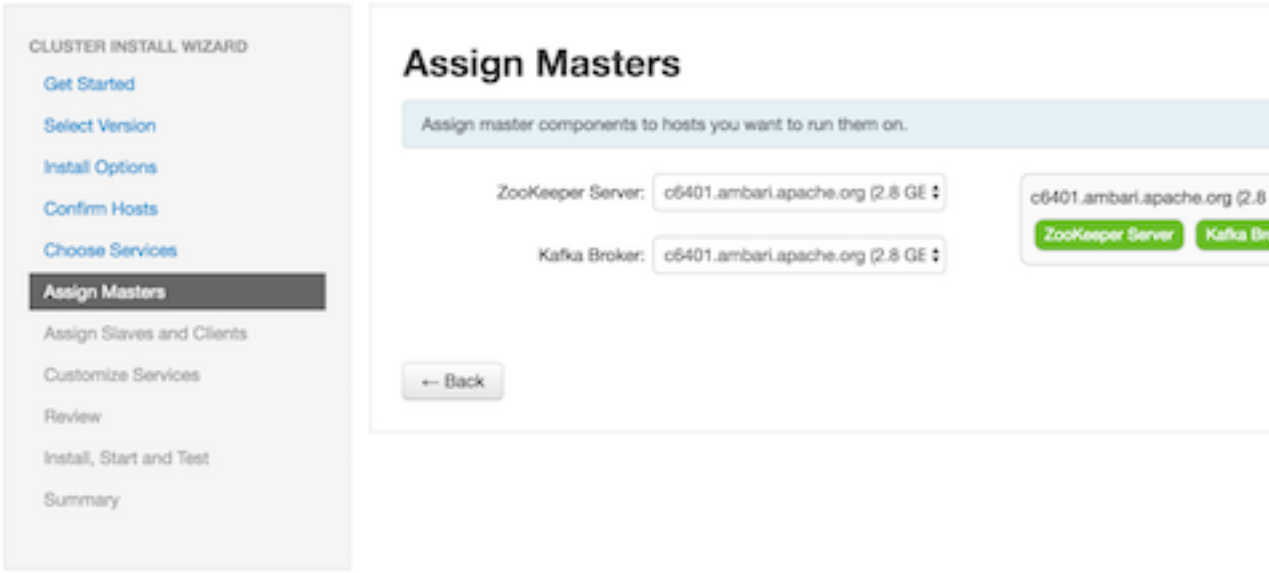
1. Click the Ambari "Services" tab.
2. In the Ambari "Actions" menu, select "Add Service." This starts the Add Service wizard, displaying the Choose Services page. Some of the services are enabled by default.
3. Scroll through the alphabetic list of components on the Choose Services page, and select "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 whic
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 da
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 w
Mahout 0.9.0 Project of the Apache Software Foundation to produce free impl
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.
← Back

4. Click "Next" to continue.

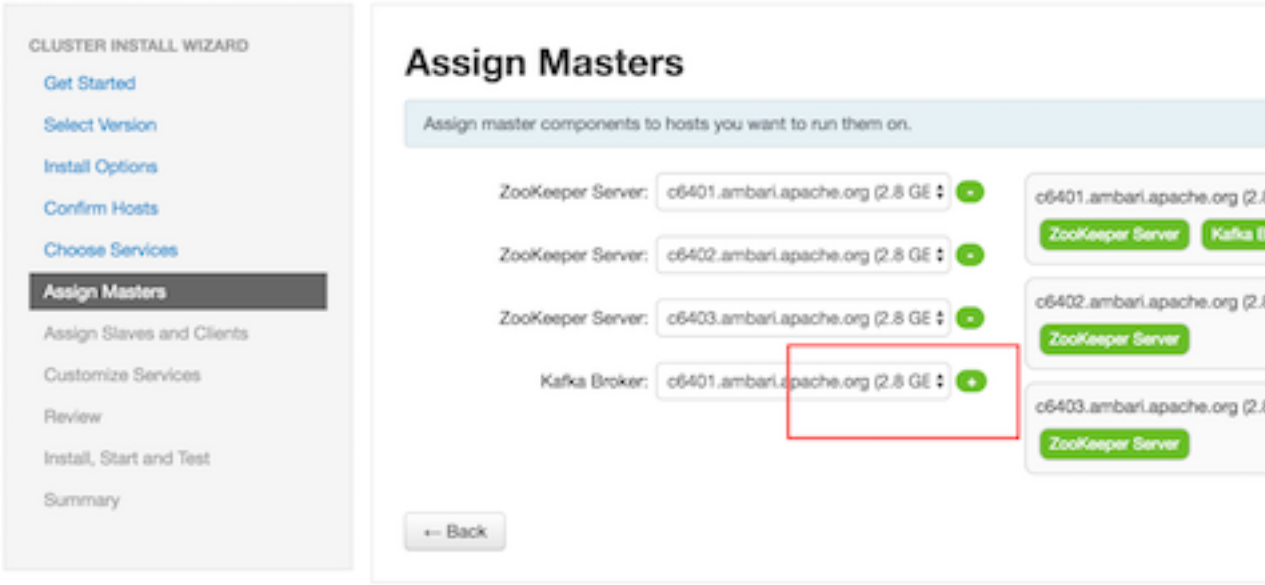
5. On the Assign Masters page, review the node assignments for Kafka nodes.

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

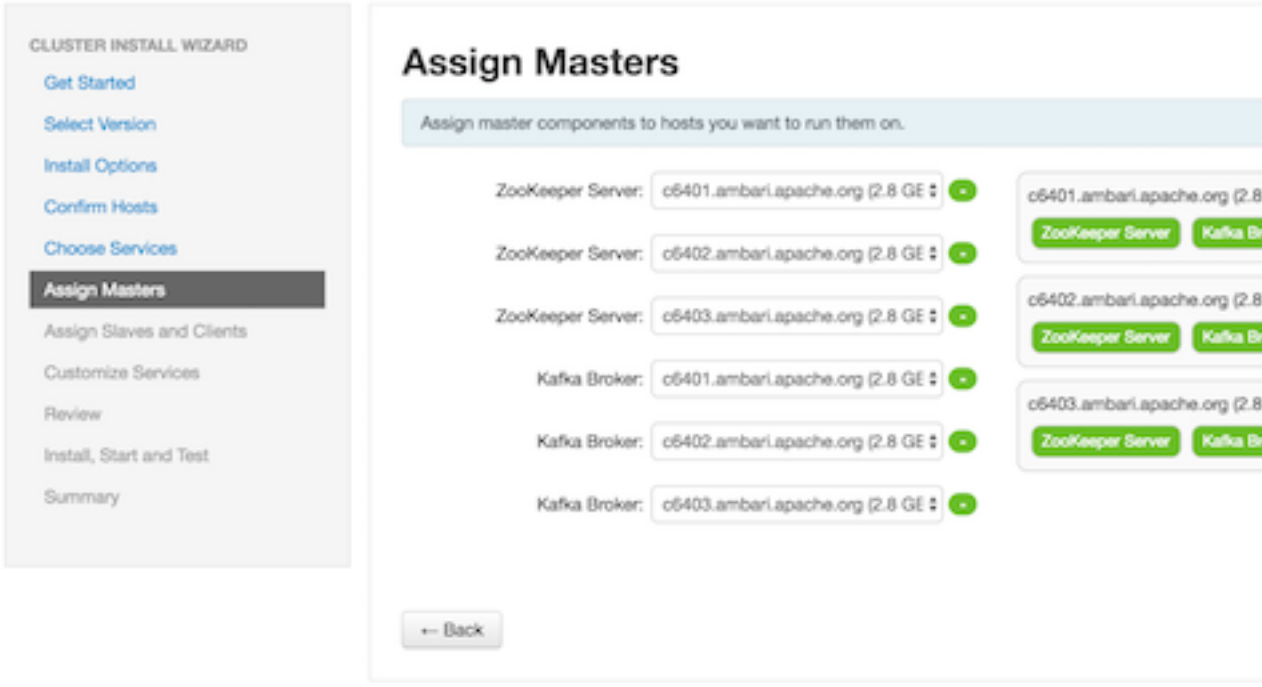


6. If you want Kafka to run with high availability, you must assign more than one node for Kafka brokers, resulting in Kafka brokers running on multiple nodes.

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

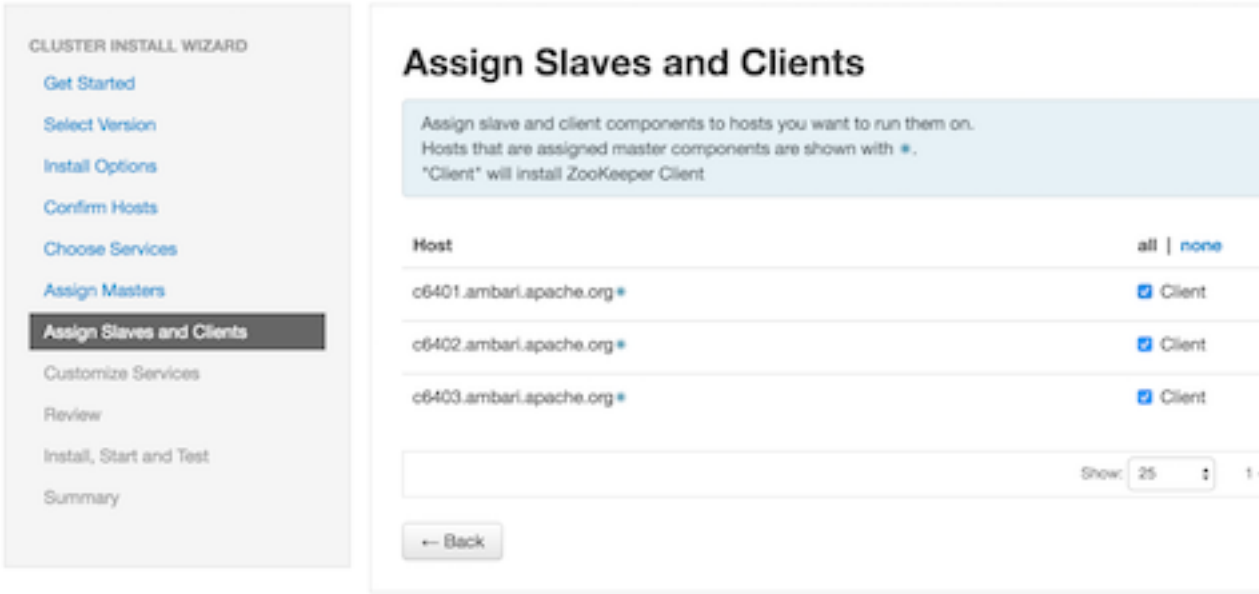


   The following screen shows node assignment for a multi-node Kafka cluster:



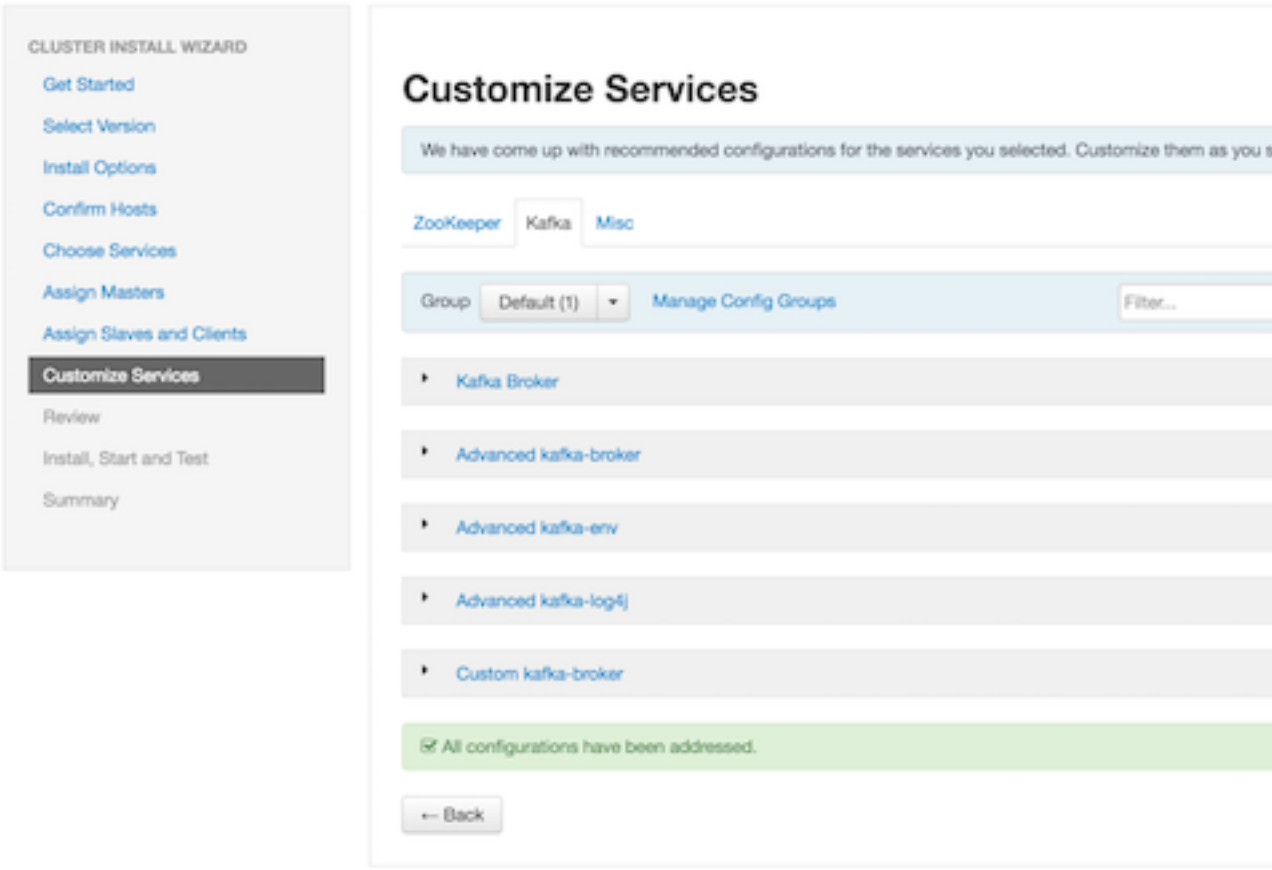
7. Click "Next" to continue.

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



9. Click "Next" to continue.

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

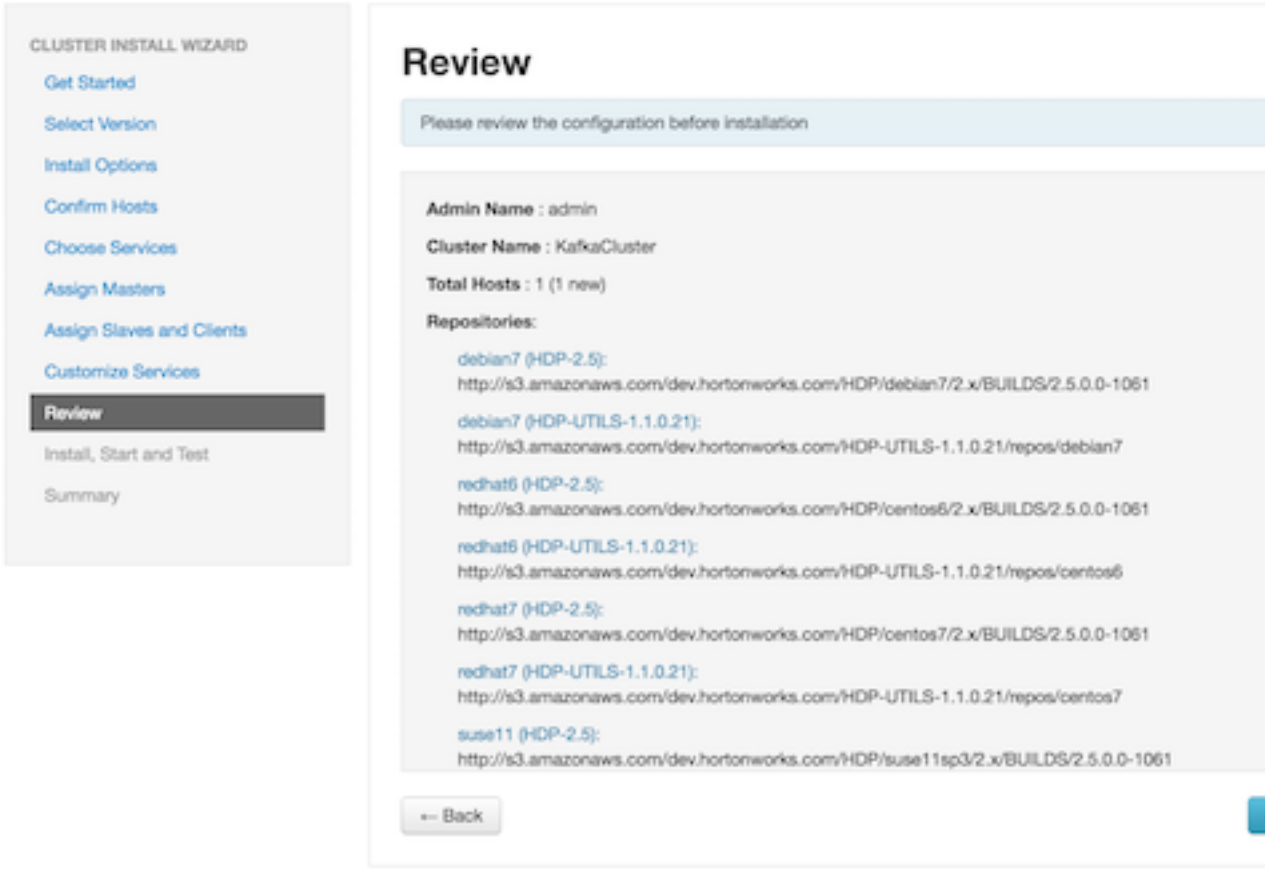


For your initial configuration you should use the default values set by Ambari. If Ambari prompts you with the message "Some configurations need your attention before you can proceed," review the list of properties and provide the required information.

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

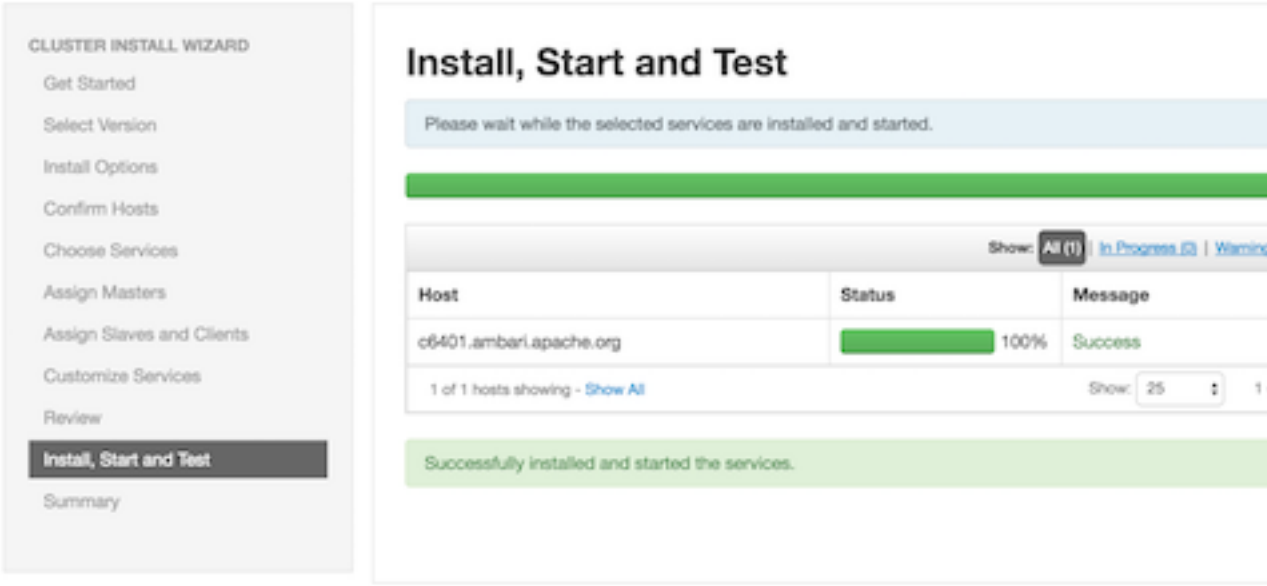
11. Click "Next" to continue.

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

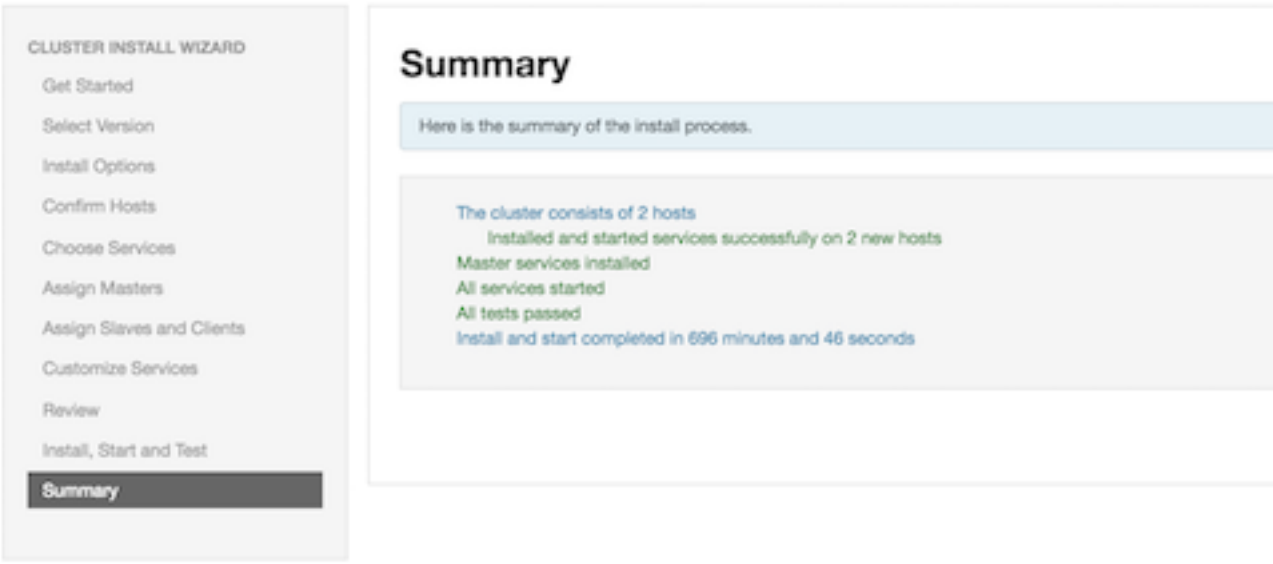


13. Click "Deploy" to begin installation.

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



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



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, including:

- Preparing the environment
- Customizing settings for brokers, producers, and consumers
- Configuring ZooKeeper for use with Kafka
- Enabling audit to HDFS when running Kafka on a secure cluster

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

5.1. Preparing the Environment

The following factors can affect Kafka performance:

- Operating system settings
- File system selection
- Disk drive configuration
- Java version
- Ethernet bandwidth

5.1.1. Operating System Settings

Consider the following when configuring Kafka:

- Kafka uses page cache memory as a buffer for active writers and readers, so after you specify JVM size (using `-Xmx` and `-Xms` Java options), leave the remaining RAM available to the operating system for page caching.
- Kafka needs open file descriptors for files and network connections. You should set the file descriptor limit to at least 128000.
- You can increase the maximum socket buffer size to enable high-performance data transfer.

5.1.2. File System Selection

Kafka uses regular Linux disk files for storage. We recommend using the EXT4 or XFS file system. Improvements to the XFS file system show improved performance characteristics for Kafka workloads without compromising stability.

Caution

- Do not use mounted shared drives or any network file systems with Kafka, due to the risk of index failures and (in the case of network file systems) issues related to the use of MemoryMapped files to store the offset index.
- Encrypted file systems such as SafenetFS are not supported for Kafka. Index file corruption can occur.

5.1.3. Disk Drive Considerations

For throughput, we recommend dedicating multiple drives to Kafka data. More drives typically perform better with Kafka than fewer. Do not share these Kafka drives with any other application or use them for Kafka application logs.

You can configure multiple drives by specifying a comma-separated list of directories for the `log.dirs` property in the `server.properties` file. Kafka uses a round-robin approach to assign partitions to directories specified in `log.dirs`; the default value is `/tmp/kafka-logs`.

The `num.io.threads` property should be set to a value equal to or greater than the number of disks dedicated for Kafka. Recommendation: start by setting this property equal to the number of disks.

Depending on how you configure flush behavior (see "Log Flush Management"), a faster disk drive is beneficial if the `log.flush.interval.messages` property is set to flush the log file after every 100,000 messages (approximately).

Kafka performs best when data access loads are balanced among partitions, leading to balanced loads across disk drives. In addition, data distribution across disks is important. If one disk becomes full and other disks have available space, this can cause performance issues. To avoid slowdowns or interruptions to Kafka services, you should create usage alerts that notify you when available disk space is low.

RAID can potentially improve load balancing among the disks, but RAID can cause performance bottleneck due to slower writes. In addition, it reduces available disk space. Although RAID can tolerate disk failures, rebuilding RAID array is I/O-intensive and effectively disables the server. Therefore, RAID does not provide substantial improvements in availability.

5.1.4. Java Version

With Apache Kafka on HDP 2.5, you should use the latest update for Java version 1.8 and make sure that G1 garbage collection support is enabled. (G1 support is enabled by default in recent versions of Java.) If you prefer to use Java 1.7, make sure that you use update u51 or later.

Here are several recommended settings for the JVM:

```
-Xmx6g
-Xms6g
-XX:MetaspaceSize=96m
-XX:+UseG1GC
-XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35
-XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80
```

To set JVM heap size for the Kafka broker, export KAFKA_HEAP_OPTS; for example:

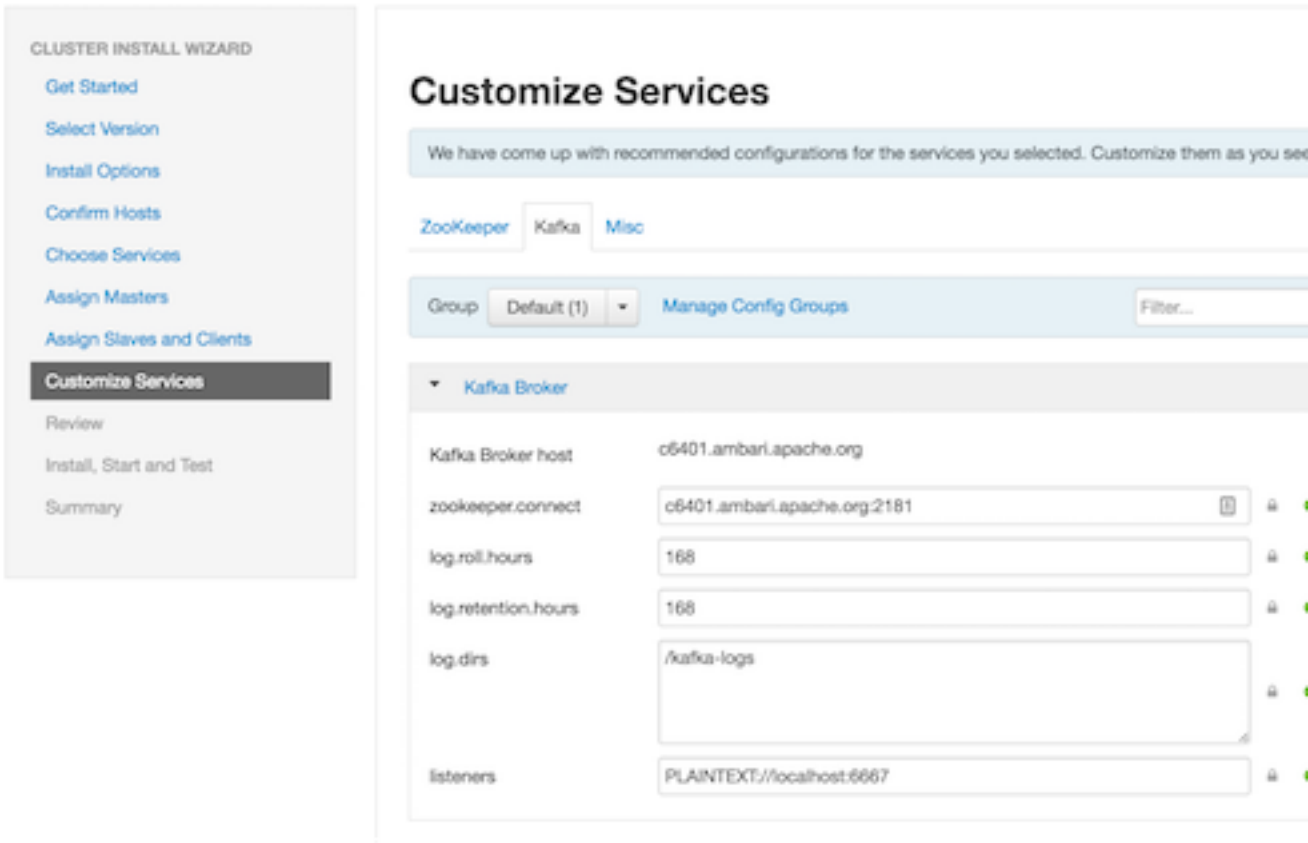
```
export KAFKA_HEAP_OPTS="-Xmx2g -Xms2g"
./kafka-server-start.sh
```

5.1.5. Ethernet Bandwidth

Ethernet bandwidth can have an impact on Kafka performance; make sure it is sufficient for your throughput requirements.

5.2. Customizing Kafka Settings on an Ambari-Managed Cluster

To customize configuration settings during the Ambari installation process, click the "Kafka" tab on the Customize Services page:

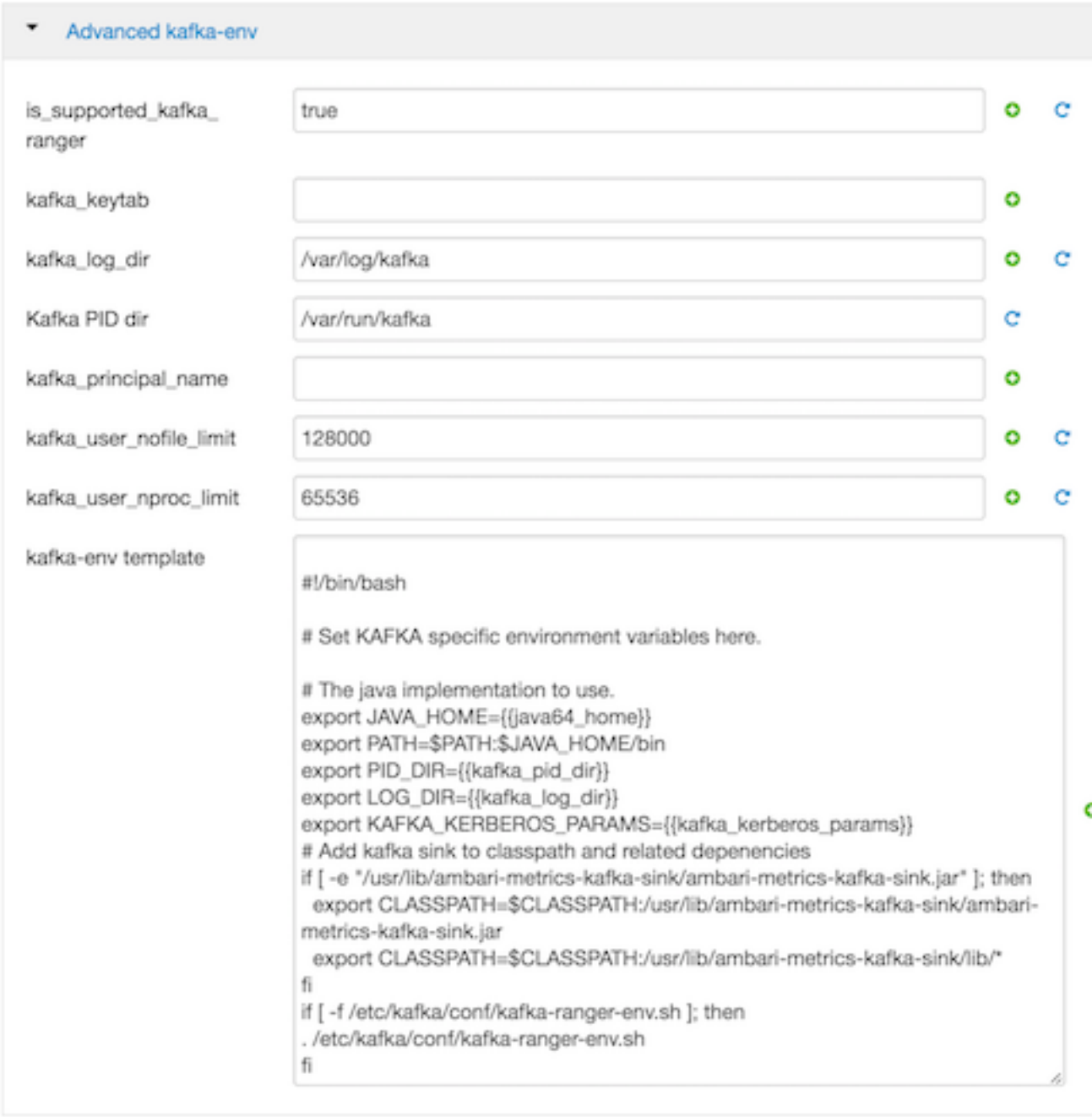


If you want to access configuration settings after installing Kafka using Ambari:

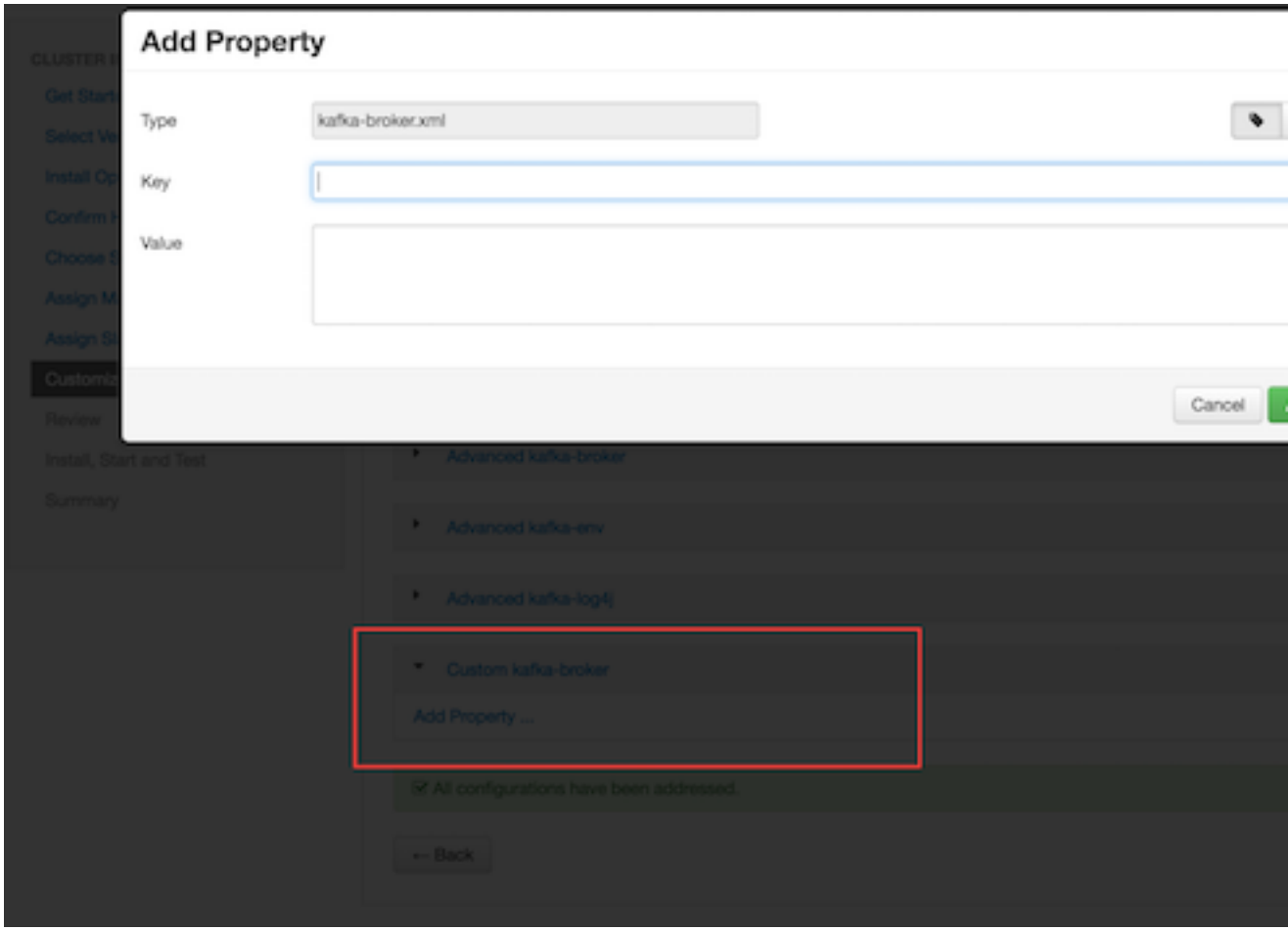
1. Click Kafka on the Ambari dashboard.
2. Choose Configs.

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.

Settings in the Advanced kafka-env category are configured by Ambari; you should not modify these settings:



To add configuration properties that are not listed by default in Ambari, navigate to the Custom kafka-broker category:



5.3. Kafka Broker Settings

The following subsections describe configuration settings that influence the performance of Kafka brokers.

5.3.1. 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 names 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.

5.3.2. Topic Settings

For each topic, Kafka maintains a structured commit log with one or more partitions. These topic partitions form the basic unit of parallelism in Kafka. In general, the more partitions there are in a Kafka cluster, the more parallel consumers can be added, resulting in higher throughput.

You can calculate the number of partitions based on your throughput requirements. If throughput from a producer to a single partition is P and throughput from a single partition to a consumer is C, and if your target throughput is T, the minimum number of required partitions is

`max (T/P, T/C).`

Note also that more partitions can increase latency:

- End-to-end latency in Kafka is defined as the difference in time from when a message is published by the producer to when the message is read by the consumer.
- Kafka only exposes a message to a consumer after it has been committed, after the message is replicated to all in-sync replicas.
- Replication of one thousand partitions from one broker to another can take up 20ms. This is too long for some real-time applications.
- In the new Kafka producer, messages are accumulated on the producer side; producers buffer the message per partition. This approach allows users to set an upper bound on the amount of memory used for buffering incoming messages. After enough data is accumulated or enough time has passed, accumulated messages are removed and sent to the broker. If you define more partitions, messages are accumulated for more partitions on the producer side.
- Similarly, the consumer fetches batches of messages per partition. Consumer memory requirements are proportional to the number of partitions that the consumer subscribes to.

Important Topic Properties

Review the following settings 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` Allows users to delete a topic from Kafka using the admin tool, for Kafka versions 0.9 and later. Deleting a topic through the admin tool will have no effect if this setting is turned off.

By default this feature is turned off (set to `false`).

5.3.3. 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).

When setting this value, take into account your disk space and how long you would like messages to be available. An active consumer can read quickly and deliver messages to their destination.

The higher the retention setting, 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.

Log Flush Management

Kafka writes topic messages to a log file immediately upon receipt, but the data is initially buffered in page cache. A log flush forces Kafka to flush topic messages from page cache, writing the messages to disk.

We recommend using the default flush settings, which rely on background flushes done by Linux and Kafka. Default settings provide high throughput and low latency, and they guarantee recovery through the use of replication.

If you decide to specify your own flush settings, you can force a flush after a period of time, or after a specified number of messages, or both (whichever limit is reached first). You can set property values globally and override them on a per-topic basis.

There are several important considerations related to log file flushing:

- Durability: unflushed data is at greater risk of loss in the event of a crash. A failed broker can recover topic partitions from its replicas, but if a follower does not issue a fetch request or consume from the leader's log-end offset within the time specified by `replica.lag.time.max.ms` (which defaults to 10 seconds), the leader removes the follower from the in-sync replica ("ISR"). When this happens there is a slight chance of message loss if you do not explicitly set `log.flush.interval.messages`. If the leader broker fails and the follower is not caught up with the leader, the follower can still be under ISR for those 10 seconds and messages during leader transition to follower can be lost.
- Increased latency: data is not available to consumers until it is flushed (the `fsync` implementation in most Linux filesystems blocks writes to the file system).
- Throughput: a flush operation is typically an expensive operation.
- Disk usage patterns are less efficient.
- Page-level locking in background flushing is much more granular.

`log.flush.interval.messages` specifies the number of messages to accumulate on a log partition before Kafka forces a flush of data to disk.

`log.flush.scheduler.interval.ms` specifies the amount of time (in milliseconds) after which Kafka checks to see if a log needs to be flushed to disk.

`log.segment.bytes` specifies the size of the log file. Kafka flushes the log file to disk whenever a log file reaches its maximum size.

`log.roll.hours` specifies the maximum length of time before a new log segment is rolled out (in hours); this value is secondary to `log.roll.ms`. Kafka flushes the log file to disk whenever a log file reaches this time limit.

5.3.4. 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).

5.3.5. General 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 allows you to specify a preference of availability or durability. *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 property to `false`.

This setting operates as follows:

- If `unclean.leader.election.enable` is set to `true` (enabled), an out-of-sync replica will be elected

	as leader when there is no live in-sync replica (ISR). This preserves the availability of the partition, but there is a chance of data loss. • If `unclean.leader.election.enable` is set to `false` and there are no live in-sync replicas, Kafka returns an error and the partition will be unavailable. This property is set to `true` by default, which favors availability. If durability is preferable to availability, set `unclean.leader.election` 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.bytes` setting in the new consumer, or smaller than the `fetch.message.max.bytes` setting in the old consumer. In addition, the value must be smaller than `replica.fetch.max.bytes`.
`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.

5.4. Kafka Producer Settings

If performance is important and you have not yet upgraded to the new Kafka producer (client version 0.9.0.1 or later), consider doing so. The new producer is generally faster and more fully featured than the previous client.

To use the new producer client, add the associated maven dependency on the client jar; for example:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.9.0.0</version>
</dependency>
```

For more information, see the KafkaProducer javadoc.

The following subsections describe several types of configuration settings that influence the performance of Kafka producers.

5.4.1. Important Producer Settings

The lifecycle of a request from producer to broker involves several configuration settings:

1. The producer polls for a batch of messages from the batch queue, one batch per partition. A batch is ready when one of the following is true:
   - `batch.size` is reached. **Note**: Larger batches typically have better compression ratios and higher throughput, but they have higher latency.
   - `linger.ms` (time-based batching threshold) is reached. Note: There is no simple guideilne for setting `linger.ms` values; you should test settings on specific use cases. For small events (100 bytes or less), this setting does not appear to have much impact.
   - Another batch to the same broker is ready.
   - The producer calls `flush()` or `close()`.
2. The producer groups the batch based on the leader broker.
3. The producer sends the grouped batch to the broker.

The following paragraphs list additional settings related to the request lifecycle:

`max.in.flight.requests.per.connection` (pipelining)
: The maximum number of unacknowledged requests the client will send on a single connection before blocking. If this setting is greater than 1, pipelining is used when the producer sends the grouped batch to the broker. This improves throughput, but if there are failed sends

there is a risk of out-of-order delivery due to retries (if retries are enabled). Note also that excessive pipelining reduces throughput.

`compression.type`

Compression is an important part of a producer's work, and the speed of different compression types differs a lot.

To specify compression type, use the `compression.type` property. It accepts standard compression codecs ('gzip', 'snappy', 'lz4'), as well as 'uncompressed' (the default, equivalent to no compression), and 'producer' (uses the compression codec set by the producer).

Compression is handled by the user thread. If compression is slow it can help to add more threads. In addition, batching efficiency impacts the compression ratio: more batching leads to more efficient compression.

`acks`

The `acks` setting specifies acknowledgments that the producer requires the leader to receive before considering a request complete. This setting defines the durability level for the producer.

Acks	Throughput	Latency	Durability
0	High	Low	No Guarantee. The producer does not wait for acknowledgment from the server.
1	Medium	Medium	Leader writes the record to its local log, and responds without awaiting full acknowledgment from all followers.
-1	Low	High	Leader waits for the full set of in-sync replicas (ISRs) to acknowledge the record. This guarantees that the record is not lost as long as at least one IRS is active.

`flush()`

The new Producer API supports an optional `flush()` call, which makes all buffered records immediately available to send (even if `linger.ms` is greater than 0).

When using `flush()`, the number of bytes between two `flush()` calls is an important factor for performance.

- In microbenchmarking tests, a setting of approximately 4MB performed well for events 1KB in size.
- A general guideline is to set `batch.size` equal to the total bytes between `flush()`calls divided by number of partitions:

  (total bytes between `flush()`calls) / (partition count)

Additional Considerations

A producer thread going to the same partition is faster than a producer thread that sends messages to multiple partitions.

If a producer reaches maximum throughput but there is spare CPU and network capacity on the server, additional producer processes can increase overall throughput.

Performance is sensitive to event size: larger events are more likely to have better throughput. In microbenchmarking tests, 1KB events streamed faster than 100-byte events.

5.5. Kafka Consumer Settings

You can usually obtain good performance from consumers without tuning configuration settings. In microbenchmarking tests, consumer performance was not as sensitive to event size or batch size as was producer performance. Both 1KG and 100B events showed similar throughput.

One basic guideline for consumer performance is to keep the number of consumer threads equal to the partition count.

5.6. Configuring ZooKeeper for Use with Kafka

Here are several recommendations for ZooKeeper configuration with Kafka:

- Do not run ZooKeeper on a server where Kafka is running.
- When using ZooKeeper with Kafka you should dedicate ZooKeeper to Kafka, and not use ZooKeeper for any other components.
- Make sure you allocate sufficient JVM memory. A good starting point is 4GB.
- To monitor the ZooKeeper instance, use JMX metrics.

Configuring ZooKeeper for Multiple Applications

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

The following example shows a sample `chroot` path:

```
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.

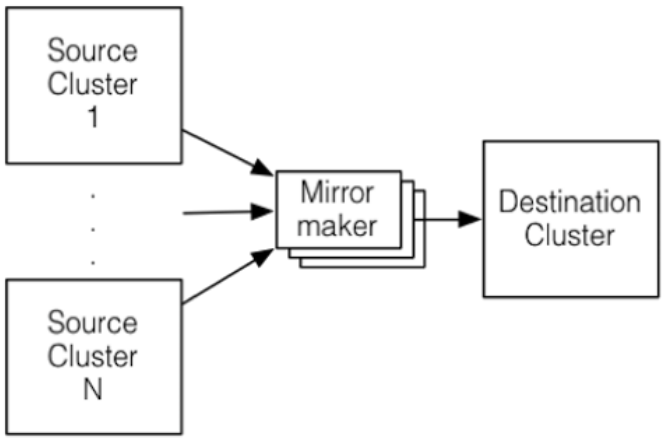
5.7. Enabling Audit to HDFS for a Secure Cluster

To enable audit to HDFS when running Kafka 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. Creating a Kafka Topic

As described in Apache Kafka Concepts, Kafka maintains feeds of messages in categories called *topics*. Producers write data to topics and consumers read from topics. Since Kafka is a distributed system, topics are partitioned and replicated across multiple nodes. Kafka treats each topic partition as a log (an ordered set of messages). Each message in a partition is assigned a unique offset.

Each topic has a user-defined category (or feed name), to which messages are published.

To create a Kafka topic, run `kafka-topics.sh` and specify topic name, replication factor, and other attributes:

```
/bin/kafka-topics.sh --create \
    --zookeeper <hostname>:<port> \
    --topic <topic-name> \
    --partitions <number-of-partitions> \
    --replication-factor <number-of-replicating-servers>
```

The following example creates a topic named "test", with one partition and one replica:

```
bin/kafka-topics.sh --create \
    --zookeeper localhost:2181 \
    --replication-factor 1 \
    --partitions 1 \
    --topic test
```

To view the topic, run the `list topic` command:

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

To create topics on a cluster with Kerberos enabled, see Creating Kafka Topics in the *HDP Security Guide*.

The `auto.create.topics.enable` property, when set to true, automatically creates topics when applications attempt to produce, consume, or fetch metadata for a nonexistent topic. For more information, see Kafka Broker Settings.

8. 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
```