

Apache Storm 3

## Using Apache Storm to Move Data

**Date of Publish:** 2019-03-08



<https://docs.hortonworks.com>

# Contents

<b>Moving Data Into and Out of Apache Storm Using Spouts and Bolts.....</b>	<b>3</b>
Ingesting Data from Kafka.....	3
KafkaSpout Integration: Core Storm APIs.....	3
KafkaSpout Integration: Trident APIs.....	7
Tuning KafkaSpout Performance.....	7
Configuring Kafka for Use with the Storm-Kafka Connector.....	8
Configuring KafkaSpout to Connect to HBase or Hive.....	8
Ingesting Data from HDFS.....	9
Configuring HDFS Spout.....	9
HDFS Spout Example.....	10
Streaming Data to Kafka.....	11
KafkaBolt Integration: Core Storm APIs.....	11
KafkaBolt Integration: Trident APIs.....	12
Writing Data to HDFS.....	14
Storm-HDFS: Core Storm APIs.....	14
Storm-HDFS: Trident APIs.....	16
Writing Data to HBase.....	16
Writing Data to Hive.....	17
Core-storm APIs.....	17
Trident APIs.....	20
Configuring Connectors for a Secure Cluster.....	21
Configuring KafkaSpout for a Secure Kafka Cluster.....	21
Configuring Storm-HDFS for a Secure Cluster.....	21
Configuring Storm-HBase for a Secure Cluster.....	23
Configuring Storm-Hive for a Secure Cluster.....	24

## Moving Data Into and Out of Apache Storm Using Spouts and Bolts

This chapter focuses on moving data into and out of Apache Storm through the use of spouts and bolts. Spouts read data from external sources to ingest data into a topology. Bolts consume input streams and process the data, emit new streams, or send results to persistent storage. This chapter focuses on bolts that move data from Storm to external sources.

The following spouts are available in HDP 2.5:

- Kafka spout based on Kafka 0.7.x/0.8.x, plus a new Kafka consumer spout available as a technical preview (not for production use)
- HDFS
- EventHubs
- Kinesis (technical preview)

The following bolts are available in HDP 2.5:

- Kafka
- HDFS
- EventHubs
- HBase
- Hive
- JDBC (supports Phoenix)
- Solr
- Cassandra
- MongoDB
- ElasticSearch
- Redis
- OpenTSDB (technical preview)

Supported connectors are located at `/usr/lib/storm/contrib`. Each contains a `.jar` file containing the connector's packaged classes and dependencies, and another `.jar` file with javadoc reference documentation.

This chapter describes how to use the Kafka spout, HDFS spout, Kafka bolt, Storm-HDFS connector, and Storm-HBase connector APIs.

### Ingesting Data from Kafka

`KafkaSpout` reads from Kafka topics. To do so, it needs to connect to the Kafka broker, locate the topic from which it will read, and store consumer offset information (using the ZooKeeper root and consumer group ID). If a failure occurs, `KafkaSpout` can use the offset to continue reading messages from the point where the operation failed.

The storm-kafka components include a core Storm spout and a fully transactional Trident spout. Storm-Kafka spouts provide the following key features:

- 'Exactly once' tuple processing with the Trident API
- Dynamic discovery of Kafka brokers and partitions

You should use the Trident API unless your application requires sub-second latency.

### KafkaSpout Integration: Core Storm APIs

The core-storm API represents a Kafka spout with the `KafkaSpout` class.

To initialize `KafkaSpout`, define a `SpoutConfig` subclass instance of the `KafkaConfig` class, representing configuration information needed to ingest data from a Kafka cluster. `KafkaSpout` requires an instance of the `BrokerHosts` interface.

#### BrokerHosts Interface

The `BrokerHost` interface maps Kafka brokers to topic partitions. Constructors for `KafkaSpout` (and, for the Trident API, `TridentKafkaConfig`) require an implementation of the `BrokerHosts` interface.

The storm-kafka component provides two implementations of `BrokerHosts`, `ZkHosts` and `StaticHosts`:

- Use `ZkHosts` if you want to track broker-to-partition mapping dynamically. This class uses Kafka's ZooKeeper entries to track mapping.

You can instantiate an object as follows:

```
public ZkHosts(String brokerZkStr, String brokerZkPath)
```

```
public ZkHosts(String brokerZkStr)
```

where:

- `brokerZkStr` is the IP:port address for the ZooKeeper host; for example, `localhost:2181`.
- `brokerZkPath` is the root directory under which topics and partition information are stored. By default this is `/brokers`, which is the default used by Kafka.

By default, broker-partition mapping refreshes every 60 seconds. If you want to change the refresh frequency, set `host.refreshFreqSecs` to your chosen value.

- Use `StaticHosts` for static broker-to-partition mapping. To construct an instance of this class, you must first construct an instance of `GlobalPartitionInformation`; for example:

```
Broker brokerForPartition0 = new Broker("localhost");//localhost:9092
Broker brokerForPartition1 = new Broker("localhost", 9092);//
localhost:9092 but we specified the port explicitly
Broker brokerForPartition2 = new Broker("localhost:9092");//localhost:9092
specified as one string.
GlobalPartitionInformation partitionInfo = new
  GlobalPartitionInformation();
partitionInfo.add(0, brokerForPartition0)//mapping form partition 0 to
  brokerForPartition0
partitionInfo.add(1, brokerForPartition1)//mapping form partition 1 to
  brokerForPartition1
partitionInfo.add(2, brokerForPartition2)//mapping form partition 2 to
  brokerForPartition2
StaticHosts hosts = new StaticHosts(partitionInfo);
```

#### KafkaConfig Class and SpoutConfig Subclass

Next, define a `SpoutConfig` subclass instance of the `KafkaConfig` class.

`KafkaConfig` contains several fields used to configure the behavior of a Kafka spout in a Storm topology; `Spoutconfig` extends `KafkaConfig`, supporting additional fields for ZooKeeper connection info and for controlling behavior specific to `KafkaSpout`.


`KafkaConfig` implements the following constructors, each of which requires an implementation of the `BrokerHosts` interface:

```
public KafkaConfig(BrokerHosts hosts, String topic)
public KafkaConfig(BrokerHosts hosts, String topic, String clientId)
```

#### KafkaConfig Parameters

##### **hosts**

One or more hosts that are Kafka ZooKeeper broker nodes (see "BrokerHosts Interface").

<b>topic</b>	Name of the Kafka topic that KafkaSpout will consume from.
<b>clientId</b>	Optional parameter used as part of the ZooKeeper path, specifying where the spout's current offset is stored.
KafkaConfig Fields	
<b>fetchSizeBytes</b>	Number of bytes to attempt to fetch in one request to a Kafka server. The default is 1MB.
<b>socketTimeoutMs</b>	Number of milliseconds to wait before a socket fails an operation with a timeout. The default value is 10 seconds.
<b>bufferSizeBytes</b>	Buffer size (in bytes) for network requests. The default is 1MB.
<b>scheme</b>	<p>The interface that specifies how a ByteBuffer from a Kafka topic is transformed into a Storm tuple.</p> <p>The default, MultiScheme, returns a tuple and no additional processing.</p> <p>The API provides many implementations of the Scheme class, including:</p> <ul style="list-style-type: none"><li>• storm.kafka.StringScheme</li><li>• storm.kafka.KeyValueSchemeAsMultiScheme</li><li>• storm.kafka.StringKeyValueScheme</li><li>• storm.kafka.KeyValueSchemeAsMultiScheme</li></ul> <p><b>Important:</b></p> <p> In Apache Storm versions prior to 1.0, MultiScheme methods accepted a byte[] parameter instead of a ByteBuffer. In Storm version 1.0, MultiScheme and related scheme APIs changed; they now accept a ByteBuffer instead of a byte[].</p> <p>As a result, Kafka spouts built with Storm versions earlier than 1.0 do not work with Storm versions 1.0 and later. When running topologies with Storm version 1.0 and later, ensure that your version of storm-kafka is at least 1.0. Rebuild pre-1.0 shaded topology .jar files that bundle storm-kafka classes with storm-kafka version 1.0 before running them in clusters with Storm 1.0 and later.</p>
<b>ignoreZKOffsets</b>	To force the spout to ignore any consumer state information stored in ZooKeeper, set ignoreZkOffsets to true. If true, the spout always begins reading from the offset defined by startOffsetTime.

<b>startOffsetTime</b>	Controls whether streaming for a topic starts from the beginning of the topic or whether only new messages are streamed. The following are valid values: <ul style="list-style-type: none"> <li>• <code>kafka.api.OffsetRequest.EarliestTime()</code> starts streaming from the beginning of the topic</li> <li>• <code>kafka.api.OffsetRequest.LatestTime()</code> streams only new messages</li> </ul>
<b>maxOffsetBehind</b>	Specifies how long a spout attempts to retry the processing of a failed tuple. If a failing tuple's offset is less than <code>maxOffsetBehind</code> , the spout stops retrying the tuple. The default is <code>LONG.MAX_VALUE</code> .
<b>useStartOffsetTimeOfOffsetOutOfRange</b>	Controls whether a spout streams messages from the beginning of a topic when the spout throws an exception for an out-of-range offset. The default value is <code>true</code> .
<b>metricsTimeBucketSizeInSecs</b>	Controls the time interval at which Storm reports spout-related metrics. The default is 60 seconds.

Instantiate `SpoutConfig` as follows:

```
public SpoutConfig(BrokerHosts hosts, String topic, String zkRoot, String
    nodeId)
```

`SpoutConfig` Parameters

<b>hosts</b>	One or more hosts that are Kafka ZooKeeper broker nodes (see "BrokerHosts Interface").
<b>topic</b>	Name of the Kafka topic that <code>KafkaSpout</code> will consume from.
<b>zkroot</b>	Root directory in ZooKeeper under which <code>KafkaSpout</code> consumer offsets are stored. The default is <code>/brokers</code> .
<b>nodeId</b>	ZooKeeper node under which <code>KafkaSpout</code> stores offsets for each topic-partition. The node ID must be unique for each Topology. The topology uses this path to recover in failure scenarios, or when there is maintenance that requires killing the topology.

`zkroot` and `nodeId` are used to construct the ZooKeeper path where Storm stores the Kafka offset. You can find offsets at `zkroot+"/"+nodeId`.

To start processing messages from where the last operation left off, use the same `zkroot` and `nodeId`. To start from the beginning of the Kafka topic, set `KafkaConfig.ignoreZKOffsets` to `true`.

Example

The following example illustrates the use of the `KafkaSpout` class and related interfaces:

```
BrokerHosts hosts = new ZkHosts(zkConnString);
SpoutConfig spoutConfig = new SpoutConfig(hosts, topicName, "/" + zkrootDir,
    nodeId);
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
```

```
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

## KafkaSpout Integration: Trident APIs

The Trident API represents a Kafka spout with the `OpaqueTridentKafkaSpout` class.

To initialize `OpaqueTridentKafkaSpout`, define a `TridentKafkaConfig` subclass instance of the `KafkaConfig` class, representing configuration information needed to ingest data from a Kafka cluster.

`KafkaConfig` Class and `TridentKafkaConfig` Subclass

Both the core-storm and Trident APIs use `KafkaConfig`, which contains several parameters and fields used to configure the behavior of a Kafka spout in a Storm topology.

Instantiate a `TridentKafkaConfig` subclass instance of the `KafkaConfig` class. Use one of the following constructors, each of which requires an implementation of the `BrokerHosts` interface.

```
public TridentKafkaConfig(BrokerHosts hosts, String topic)
public TridentKafkaConfig(BrokerHosts hosts, String topic, String id)
```

`TridentKafkaConfig` Parameters

<b>hosts</b>	One or more hosts that are Kafka ZooKeeper broker nodes (see "BrokerHosts Interface").
<b>topic</b>	Name of the Kafka topic.
<b>clientid</b>	Unique identifier for this spout.

Example

The following example illustrates the use of the `OpaqueTridentKafkaSpout` class and related interfaces:

```
TridentTopology topology = new TridentTopology();
BrokerHosts zk = new ZkHosts("localhost");
TridentKafkaConfig spoutConf = new TridentKafkaConfig(zk, "test-topic");
spoutConf.scheme = new SchemeAsMultiScheme(new StringScheme());
OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(spoutConf);
```



### Important:

In Apache Storm versions prior to 1.0, `MultiScheme` methods accepted a `byte[]` parameter instead of a `ByteBuffer`. In Storm version 1.0, `MultiScheme` and related scheme APIs changed; they now accept a `ByteBuffer` instead of a `byte[]`.

As a result, Kafka spouts built with Storm versions earlier than 1.0 do not work with Storm versions 1.0 and later. When running topologies with Storm version 1.0 and later, ensure that your version of `storm-kafka` is at least 1.0. Rebuild pre-1.0 shaded topology `.jar` files that bundle `storm-kafka` classes with `storm-kafka` version 1.0 before running them in clusters with Storm 1.0 and later.

## Tuning KafkaSpout Performance

`KafkaSpout` provides two internal parameters to control performance:

- `offset.commit.period.ms` specifies the period of time (in milliseconds) after which the spout commits to Kafka. To set this parameter, use the `KafkaSpoutConfig` set method `setOffsetCommitPeriodMs`.
- `max.uncommitted.offsets` defines the maximum number of polled offsets (records) that can be pending commit before another poll can take place. When this limit is reached, no more offsets can be polled until the next successful commit sets the number of pending offsets below the threshold. To set this parameter, use the `KafkaSpoutConfig` set method `setMaxUncommittedOffsets`.

Note that these two parameters trade off memory versus time:

- When `offset.commit.period.ms` is set to a low value, the spout commits to Kafka more often. When the spout is committing to Kafka, it is not fetching new records nor processing new tuples.
- When `max.uncommitted.offsets` increases, the memory footprint increases. Each offset uses eight bytes of memory, which means that a value of 10000000 (10MB) uses about 80MB of memory.

It is possible to achieve good performance with a low commit period and small memory footprint (a small value for `max.uncommitted.offsets`), as well as with a larger commit period and larger memory footprint. However, you should avoid using large values for `offset.commit.period.ms` with a low value for `max.uncommitted.offsets`.

Kafka consumer [configuration parameters](#) can also have an impact on the KafkaSpout performance. The following Kafka parameters are most likely to have the strongest impact on KafkaSpout performance:

- The [Kafka Consumer](#) poll timeout specifies the time (in milliseconds) spent polling if data is not available. To set this parameter, use the [KafkaSpoutConfig](#) set method `setPollTimeoutMs`.
- Kafka consumer parameter `fetch.min.bytes` specifies the minimum amount of data the server returns for a fetch request. If the minimum amount is not available, the request waits until the minimum amount accumulates before answering the request.
- Kafka consumer parameter `fetch.max.wait.ms` specifies the maximum amount of time the server will wait before answering a fetch request, when there is not sufficient data to satisfy `fetch.min.bytes`.



#### Important:

For HDP 2.5.0 clusters in production use, you should override the default values of KafkaSpout parameters `offset.commit.period` and `max.uncommitted.offsets`, and Kafka consumer parameter `poll.timeout.ms`, as follows:

- Set `poll.timeout.ms` to 200.
- Set `offset.commit.period.ms` to 30000 (30 seconds).
- Set `max.uncommitted.offsets` to 10000000 (ten million).

Performance also depends on the structure of your Kafka cluster, the distribution of the data, and the availability of data to poll.

#### Log Level Performance Impact

Storm supports several logging levels, including Trace, Debug, Info, Warn, and Error. Trace-level logging has a significant impact on performance, and should be avoided in production. The amount of log messages is proportional to the number of records fetched from Kafka, so a lot of messages are printed when Trace-level logging is enabled.

Trace-level logging is most useful for debugging pre-production environments under mild load. For debugging, if necessary, you can throttle how many messages are polled from Kafka by setting the `max.partition.fetch.bytes` parameter to a low number that is larger than the largest single message stored in Kafka.

Logs with Debug level will have slightly less performance impact than Trace-level logs, but still generate a lot of messages. This setting can be useful for assessing whether the Kafka spout is properly tuned.

## Configuring Kafka for Use with the Storm-Kafka Connector

Before using the storm-kafka connector, you must modify your Apache Kafka configuration: add a `zookeeper.connect` property, with hostnames and port numbers of HDP ZooKeeper nodes, to the Kafka `server.properties` file.

## Configuring KafkaSpout to Connect to HBase or Hive

Before connecting to HBase or Hive, add the following exclusions to your POM file for the curator framework:

```
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
</exclusion>
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
```



```
</exclusion>
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>apache-curator</artifactId>
</exclusion>
```

## Ingesting Data from HDFS

The HDFS spout actively monitors a specified HDFS directory and consumes any new files that appear in the directory, feeding data from HDFS to Storm.



### Important:

HDFS spout assumes that files visible in the monitored directory are not actively being updated. Only after a file is completely written should it be made visible to the spout. Following are two approaches for ensuring this:

- Write the file to another directory. When the write operation is finished, move the file to the monitored directory.
- Create the file in the monitored directory with an '.ignore' suffix; HDFS spout ignores files with an '.ignore' suffix. When the write operation is finished, rename the file to omit the suffix.

When the spout is actively consuming a file, it renames the file with an .inprogress suffix. After consuming all contents in the file, the file is moved to a configurable done directory and the .inprogress suffix is dropped.

### Concurrency

If multiple spout instances are used in the topology, each instance consumes a different file. Synchronization among spout instances relies on lock files created in a subdirectory called .lock (by default) under the monitored directory. A file with the same name as the file being consumed (without the .inprogress suffix) is created in the lock directory. Once the file is completely consumed, the corresponding lock file is deleted.

### Recovery from failure

Periodically, the spout records information about how much of the file has been consumed in the lock file. If the spout instance crashes or there is a force kill of topology, another spout can take over the file and resume from the location recorded in the lock file.

Certain error conditions (such as a spout crash) can leave residual lock files. Such a stale lock file indicates that the corresponding input file has not been completely processed. When detected, ownership of such stale lock files will be transferred to another spout.

The `hdfsspout.lock.timeout.sec` property specifies the duration of inactivity after which lock files should be considered stale. The default timeout is five minutes. For lock file ownership transfer to succeed, the HDFS lease on the file (from the previous lock owner) should have expired. Spouts scan for stale lock files before selecting the next file for consumption.

### Lock on .lock Directory

HDFS spout instances create a DIRLOCK file in the .lock directory to coordinate certain accesses to the .lock directory itself. A spout will try to create it when it needs access to the .lock directory, and then delete it when done. In error conditions such as a topology crash, force kill, or untimely death of a spout, this file may not be deleted. Future instances of the spout will eventually recover the file once the DIRLOCK file becomes stale due to inactivity for `hdfsspout.lock.timeout.sec` seconds.

### API Support

HDFS spout supports core Storm, but does not currently support Trident.

## Configuring HDFS Spout

The following member functions are required for `HdfsSpout`:

<b>.setReaderType()</b>	<p>Specifies which file reader to use:</p> <ul style="list-style-type: none"> <li>• To read sequence files, set this to 'seq'.</li> <li>• To read text files, set this to 'text'.</li> <li>• If you want to use a custom file reader class that implements interface <code>org.apache.storm.hdfs.spout.FileReader</code>, set this to the fully qualified class name.</li> </ul>
<b>.withOutputFields()</b>	<p>Specifies names of output fields for the spout. The number of fields depends upon the reader being used.</p> <p>For convenience, built-in reader types expose a static member called <code>defaultFields</code> that can be used for setting this.</p>
<b>.setHdfsUri()</b>	<p>Specifies the HDFS URI for HDFS NameNode; for example: <code>hdfs://namenodehost:8020</code>.</p>
<b>.setSourceDir()</b>	<p>Specifies the HDFS directory from which to read files; for example, <code>/data/inputdir</code>.</p>
<b>.setArchiveDir()</b>	<p>Specifies the HDFS directory to move a file after the file is completely processed; for example, <code>/data/done</code>.</p> <p>If this directory does not exist, it will be created automatically.</p>
<b>.setBadFilesDir()</b>	<p>Specifies a directory to move a file if there is an error parsing the contents of the file; for example, <code>/data/badfiles</code>.</p> <p>If this directory does not exist it will be created automatically.</p>

For additional configuration settings, see Apache HDFS spout [Configuration Settings](#).

## HDFS Spout Example

The following example creates an HDFS spout that reads text files from HDFS path `hdfs://localhost:54310/source`.

```
// Instantiate spout to read text files
HdfsSpout textReaderSpout = newHdfsSpout().setReaderType("text")

.withOutputFields(TextFileReader.defaultFields)

localhost:54310" ) // reqd
                    .setHdfsUri("hdfs://
                    .setSourceDir("/data/in")
                    // reqd
                    .setArchiveDir("/data/done")
                    // reqd
                    .setBadFilesDir("/data/badfiles");
                    // required

// If using Kerberos
HashMap hdfsSettings = new HashMap();
hdfsSettings.put("hdfs.keytab.file", "/path/to/keytab");
hdfsSettings.put("hdfs.kerberos.principal", "user@EXAMPLE.com");
```

```

textReaderSpout.setHdfsClientSettings(hdfsSettings);

// Create topology
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("hdfsspout", textReaderSpout, SPOUT_NUM);

// Set up bolts and wire up topology
...

// Submit topology with config
Config conf = new Config();
StormSubmitter.submitTopologyWithProgressBar("topologyName", conf,
    builder.createTopology());

```

A sample topology `HdfsSpoutTopology` is provided in the `storm-starter` module.

## Streaming Data to Kafka

Storm provides a Kafka Bolt for both the core-storm and Trident APIs that publish data to Kafka topics.

Use the following procedure to add a Storm component to your topology that writes data to a Kafka cluster:

1. Instantiate a Kafka Bolt.
2. Configure the Kafka Bolt with a Tuple-to-Message mapper.
3. Configure the Kafka Bolt with a Kafka Topic Selector.
4. Configure the Kafka Bolt with Kafka Producer properties.

The following code samples illustrate the construction of a simple Kafka bolt.

### KafkaBolt Integration: Core Storm APIs

To use `KafkaBolt`, create an instance of `org.apache.storm.kafka.bolt.KafkaBolt` and attach it as a component to your topology.

The following example shows construction of a Kafka bolt using core Storm APIs, followed by details about the code:

```

TopologyBuilder builder = new TopologyBuilder();

Fields fields = new Fields("key", "message");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
    new Values("needs", "1"),
    new Values("javadoc", "1")
);
spout.setCycle(true);
builder.setSpout("spout", spout, 5);
//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

KafkaBolt bolt = new KafkaBolt()
    .withProducerProperties(props)
    .withTopicSelector(new DefaultTopicSelector("test"))
    .withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
builder.setBolt("forwardToKafka", bolt, 8).shuffleGrouping("spout");

```

```
Config conf = new Config();

StormSubmitter.submitTopology("kafkaboltTest", conf,
    builder.createTopology());
```

### 1. Instantiate a KafkaBolt.

The core-storm API uses the `storm.kafka.bolt.KafkaBolt` class to instantiate a Kafka Bolt:

```
KafkaBolt bolt = new KafkaBolt();
```

### 2. Configure the KafkaBolt with a Tuple-to-Message Mapper.

The `KafkaBolt` maps Storm tuples to Kafka messages. By default, `KafkaBolt` looks for fields named "key" and "message." Storm provides the `storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper` class to support this default behavior and provide backward compatibility. The class is used by both the core-storm and Trident APIs.

```
KafkaBolt bolt = new KafkaBolt()
    .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
```

### 3. Configure the Kafka Bolt with a Kafka Topic Selector.



#### Note:

To ignore a message, return NULL from the `getTopics()` method.

```
KafkaBolt bolt = new KafkaBolt().withTupleToKafkaMapper(new
    FieldNameBasedTupleToKafkaMapper())
    .withTopicSelector(new DefaultTopicSelector());
```

If you need to write to multiple Kafka topics, you can write your own implementation of the `KafkaTopicSelector` interface.

### 4. Configure the Kafka Bolt with Kafka Producer properties.

You can specify producer properties in your Storm topology by calling `KafkaBolt.withProducerProperties()`. See the Apache [Producer Configs](#) documentation for more information.

## KafkaBolt Integration: Trident APIs

To use `KafkaBolt`, create an instance of `org.apache.storm.kafka.trident.TridentState` and `org.apache.storm.kafka.trident.TridentStateFactory`, and attach them to your topology.

The following example shows construction of a Kafka bolt using Trident APIs, followed by details about the code:

```
Fields fields = new Fields("word", "count");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
    new Values("storm", "1"),
    new Values("trident", "1"),
    new Values("needs", "1"),
    new Values("javadoc", "1")
);

spout.setCycle(true);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
```

```

props.put("acks", "1");
props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
    .withProducerProperties(props)
    .withKafkaTopicSelector(new DefaultTopicSelector("test"))
    .withTridentTupleToKafkaMapper(new
        FieldNameBasedTupleToKafkaMapper("word", "count"));
stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(), new
    Fields());

Config conf = new Config();
StormSubmitter.submitTopology("kafkaTridentTest", conf, topology.build());

```

### 1. Instantiate a KafkaBolt.

The Trident API uses a combination of the `storm.kafka.trident.TridentStateFactory` and `storm.kafka.trident.TridentKafkaStateFactory` classes.

```

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout");
TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory();
stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(),
    new Fields());

```

### 2. Configure the KafkaBolt with a Tuple-to-Message Mapper.

The `KafkaBolt` must map Storm tuples to Kafka messages. By default, `KafkaBolt` looks for fields named "key" and "message." Storm provides the `storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper` class to support this default behavior and provide backward compatibility. The class is used by both the core-storm and Trident APIs.

```

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
    .withTridentTupleToKafkaMapper(new
        FieldNameBasedTupleToKafkaMapper("word", "count"));

```

You must specify the field names for the Storm tuple key and the Kafka message for any implementation of the `TridentKafkaState` in the Trident API. This interface does not provide a default constructor.

However, some Kafka bolts may require more than two fields. You can write your own implementation of the `TupleToKafkaMapper` and `TridentTupleToKafkaMapper` interfaces to customize the mapping of Storm tuples to Kafka messages. Both interfaces define two methods:

```
K getKeyFromTuple(Tuple/TridentTuple tuple);
```

```
V getMessageFromTuple(Tuple/TridentTuple tuple);
```

### 3. Configure the KafkaBolt with a Kafka Topic Selector.



#### Note:

To ignore a message, return NULL from the `getTopics()` method.

```

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
    .withKafkaTopicSelector(new DefaultTopicSelector("test"))
    .withTridentTupleToKafkaMapper(new
        FieldNameBasedTupleToKafkaMapper("word", "count"));

```

If you need to write to multiple Kafka topics, you can write your own implementation of the `KafkaTopicSelector` interface; for example:

```
public interface KafkaTopicSelector {
    String getTopics(Tuple/TridentTuple tuple);
}
```

#### 4. Configure the KafkaBolt with Kafka Producer properties.

You can specify producer properties in your Storm topology by calling `TridentKafkaStateFactory.withProducerProperties()`. See the Apache [Producer Configs](#) documentation for more information.

## Writing Data to HDFS

The storm-hdfs connector supports core Storm and Trident APIs. You should use the trident API unless your application requires sub-second latency.

### Storm-HDFS: Core Storm APIs

The primary classes of the storm-hdfs connector are `HdfsBolt` and `SequenceFileBolt`, both located in the `org.apache.storm.hdfs.bolt` package. Use the `HdfsBolt` class to write text data to HDFS and the `SequenceFileBolt` class to write binary data.

For more information about the `HdfsBolt` class, refer to the Apache Storm [HdfsBolt](#) documentation.

Specify the following information when instantiating the bolt:

HdfsBolt Methods

<b>withFsUrl</b>	Specifies the target HDFS URL and port number.
<b>withRecordFormat</b>	Specifies the delimiter that indicates a boundary between data records. Storm developers can customize by writing their own implementation of the <code>org.apache.storm.hdfs.format.RecordFormat</code> interface. Use the provided <code>org.apache.storm.hdfs.format.DelimitedRecordFormat</code> class as a convenience class for writing delimited text data with delimiters such as tabs, comma-separated values, and pipes. The storm-hdfs bolt uses the <code>RecordFormat</code> implementation to convert tuples to byte arrays, so this method can be used with both text and binary data.
<b>withRotationPolicy</b>	Specifies when to stop writing to a data file and begin writing to another. Storm developers can customize by writing their own implementation of the <code>org.apache.storm.hdfs.rotation.FileSizeRotationSizePolicy</code> interface.
<b>withSyncPolicy</b>	Specifies how frequently to flush buffered data to the HDFS filesystem. This action enables other HDFS clients to read the synchronized data, even as the Storm client continues to write data. Storm developers can customize by writing their own implementation of the <code>org.apache.storm.hdfs.sync.SyncPolicy</code> interface.

**withFileNameFormat**

Specifies the name of the data file. Storm developers can customize by writing their own interface of the `org.apache.storm.hdfs.format.FileNameFormat` interface. The provided `org.apache.storm.hdfs.format.DefaultFileNameFormat` creates file names with the following naming format: `{prefix}-{componentId}-{taskId}-{rotationNum}-{timestamp}-{extension}`.

Example: `MyBolt-5-7-1390579837830.txt`.

**Example: Cluster Without High Availability ("HA")**

The following example writes pipe-delimited files to the HDFS path `hdfs://localhost:8020/foo`. After every 1,000 tuples it will synchronize with the filesystem, making the data visible to other HDFS clients. It will rotate the files when they reach 5 MB in size.

Note that the `HdfsBolt` is instantiated with an HDFS URL and port number.

```

```java
// use "|" instead of "," for field delimiter
RecordFormat format = new DelimitedRecordFormat()
    .withFieldDelimiter("|");

// Synchronize the filesystem after every 1000 tuples
SyncPolicy syncPolicy = new CountSyncPolicy(1000);

// Rotate data files when they reach 5 MB
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f,
    Units.MB);

// Use default, Storm-generated file names
FileNameFormat fileNameFormat = new DefaultFileNameFormat()
    .withPath("/foo/");

// Instantiate the HdfsBolt
HdfsBolt bolt = new HdfsBolt()
    .withFsUrl("hdfs://localhost:8020")
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(format)
    .withRotationPolicy(rotationPolicy)
    .withSyncPolicy(syncPolicy);
```

```

**Example: HA-Enabled Cluster**

The following example shows how to modify the previous example for an HA-enabled cluster.

Here the `HdfsBolt` is instantiated with a nameservice ID, instead of using an HDFS URL and port number.

```

...
HdfsBolt bolt = new HdfsBolt()
    .withFsURL("hdfs://myNameserviceID")
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(format)
    .withRotationPolicy(rotationPolicy)
    .withSyncPolicy(syncPolicy);
...

```

To obtain the nameservice ID, check the `dfs.nameservices` property in your `hdfs-site.xml` file; `nnha` in the following example:

```
<property>
  <name>dfs.nameservices</name>
  <value>nnha</value>
</property>
```

## Storm-HDFS: Trident APIs

The Trident API implements a `StateFactory` class with an API that resembles the methods from the `storm-code` API, as shown in the following code sample:

```
...
Fields hdfsFields = new Fields("field1", "field2");

FileNameFormat fileNameFormat = new DefaultFileNameFormat()
    .withPrefix("trident")
    .withExtension(".txt")
    .withPath("/trident");

RecordFormat recordFormat = new DelimitedRecordFormat()
    .withFields(hdfsFields);

FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f,
    FileSizeRotationPolicy.Units.MB);

HdfsState.Options options = new HdfsState.HdfsFileOptions()
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(recordFormat)
    .withRotationPolicy(rotationPolicy)
    .withFsUrl("hdfs://localhost:8020");

StateFactory factory = new HdfsStateFactory().withOptions(options);

TridentState state = stream.partitionPersist(factory, hdfsFields, new
    HdfsUpdater(), new Fields());
```

See the javadoc for the Trident API, included with the `storm-hdfs` connector, for more information.

### Limitations

Directory and file names changes are limited to a prepackaged file name format based on a timestamp.

## Writing Data to HBase

The `storm-hbase` connector enables Storm developers to collect several PUTS in a single operation and write to multiple HBase column families and counter columns. A PUT is an HBase operation that inserts data into a single HBase cell.

Use the HBase client's write buffer to automatically batch: `hbase.client.write.buffer`.

The primary interface in the `storm-hbase` connector is the `org.apache.storm.hbase.bolt.mapper.HBaseMapper` interface. However, the default implementation, `SimpleHBaseMapper`, writes a single column family. Storm developers can implement the `HBaseMapper` interface themselves or extend `SimpleHBaseMapper` if they want to change or override this behavior.

### SimpleHBaseMapper Methods



<b>withRowKeyField</b>	Specifies the row key for the target HBase row. A row key uniquely identifies a row in HBase
<b>withColumnFields</b>	Specifies the target HBase column.
<b>withCounterFields</b>	Specifies the target HBase counter.
<b>withColumnFamily</b>	Specifies the target HBase column family.

### Example

The following example specifies the 'word' tuple as the row key, adds an HBase column for the tuple 'word' field, adds an HBase counter column for the tuple 'count' field, and writes data to the 'cf' column family.

```
SimpleHBaseMapper mapper = new SimpleHBaseMapper()
    .withRowKeyField("word")
    .withColumnFields(new Fields("word"))
    .withCounterFields(new Fields("count"))
    .withColumnFamily("cf");
```

## Writing Data to Hive

Core Storm and Trident APIs support streaming data directly to Apache Hive using Hive transactions. Data committed in a transaction is immediately available to Hive queries from other Hive clients. You can stream data to existing table partitions, or configure the streaming Hive bolt to dynamically create desired table partitions.

Use the following steps to perform this procedure:

1. Instantiate an implementation of the HiveMapper Interface.
2. Instantiate a HiveOptions class with the HiveMapper implementation.
3. Instantiate a HiveBolt with the HiveOptions class.



### Note:

Currently, data may be streamed only into bucketed tables using the ORC file format.

## Core-storm APIs

The following example constructs a Kafka bolt using core Storm APIs:

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames));
HiveOptions hiveOptions = new
HiveOptions(metaStoreURI, dbName, tblName, mapper);
HiveBolt hiveBolt = new HiveBolt(hiveOptions);
```

1. Instantiate an Implementation of HiveMapper Interface.

The storm-hive streaming bolt uses the HiveMapper interface to map the names of tuple fields to the names of Hive table columns. Storm provides two implementations: DelimitedRecordHiveMapper and JsonRecordHiveMapper. Both implementations take the same arguments.

**Table 1: HiveMapper Arguments**

Argument	Data Type	Description
withColumnFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table column names.

Argument	Data Type	Description
withPartitionFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table partitions.
withTimeAsPartitionField	String	Requests that table partitions be created with names set to system time. Developers can specify any Java-supported date format, such as "YYYY/MM/DD".

The following sample code illustrates how to use DelimitedRecordHiveMapper:

```

...
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withPartitionFields(new Fields(partNames));

DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withTimeAsPartitionField("YYYY/MM/DD");
...

```

2. Instantiate a HiveOptions Class with the HiveMapper Implementation. The HiveOptions class configures transactions used by Hive to ingest the streaming data:

```

...
HiveOptions hiveOptions = new
    HiveOptions(metaStoreURI, dbName, tblName, mapper)
    .withTxnsPerBatch(10)
    .withBatchSize(1000)
    .withIdleTimeout(10);
...

```

The following table describes all configuration properties for the HiveOptions class.

**Table 2: HiveOptions Class Configuration Properties**

HiveOptions Configuration Property	Data Type	Description
metaStoreURI	String	Hive Metastore URI. Storm developers can find this value in hive-site.xml.
dbName	String	Database name
tblName	String	Table name
mapper	Mapper	Two properties that start with "org.apache.storm.hive.bolt.": mapper.DelimitedRecordHiveMapper mapper.JsonRecordHiveMapper

HiveOptions Configuration Property	Data Type	Description
withTxnsPerBatch	Integer	Configures the number of desired transactions per transaction batch. Data from all transactions in a single batch form a single compaction file. Storm developers use this property in conjunction with the withBatchSize property to control the size of compaction files. The default value is 100.  Hive stores data in base files that cannot be updated by HDFS. Instead, Hive creates a set of delta files for each transaction that alters a table or partition and stores them in a separate delta directory. Occasionally, Hive compacts, or merges, the base and delta files. Hive performs all compactions in the background without affecting concurrent reads and writes of other Hive clients.
withMaxOpenConnections	Integer	Specifies the maximum number of open connections. Each connection is to a single Hive table partition. The default value is 500. When Hive reaches this threshold, an idle connection is terminated for each new connection request. A connection is considered idle if no data is written to the table partition to which the connection is made.
withBatchSize	Integer	Specifies the maximum number of Storm tuples written to Hive in a single Hive transaction. The default value is 15000 tuples.
withCallTimeout	Integer	Specifies the interval in seconds between consecutive heartbeats sent to Hive. Hive uses heartbeats to prevent expiration of unused transactions. Set this value to 0 to disable heartbeats. The default value is 240.
withAutoCreatePartitions	Boolean	Indicates whether HiveBolt should automatically create the necessary Hive partitions needed to store streaming data. The default value is true.
withKerberosPrincipal	String	Kerberos user principal for accessing a secured Hive installation.
withKerberosKeytab	String	Kerberos keytab for accessing a secured Hive installation.

3. Instantiate the HiveBolt with the HiveOptions class:

```
...
HiveBolt hiveBolt = new HiveBolt(hiveOptions);
...
```

4. Before building your topology code, add the following dependency to your topology pom.xml file:

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.3.3</version>
</dependency>
```

## Trident APIs

The following example shows construction of a Kafka bolt using core Storm APIs, followed by details about the code:

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withTimeAsPartitionField("YYYY/MM/DD");

HiveOptions hiveOptions = new
    HiveOptions(metaStoreURI, dbName, tblName, mapper)
    .withTxnsPerBatch(10)
    .withBatchSize(1000)
    .withIdleTimeout(10);

StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
TridentState state = stream.partitionPersist(factory, hiveFields, new
    HiveUpdater(),
    new Fields());
```

### 1. Instantiate an Implementation of HiveMapper Interface

The storm-hive streaming bolt uses the HiveMapper interface to map the names of tuple fields to the names of Hive table columns. Storm provides two implementations: DelimitedRecordHiveMapper and JsonRecordHiveMapper. Both implementations take the same arguments.

**Table 3: HiveMapper Arguments**

Argument	Data Type	Description
withColumnFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table column names.
withPartitionFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table partitions.
withTimeAsPartitionField	String	Requests that table partitions be created with names set to system time. Developers can specify any Java-supported date format, such as "YYYY/MM/DD".

The following sample code illustrates how to use DelimitedRecordHiveMapper:

```
...
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withPartitionFields(new Fields(partNames));

DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
    .withColumnFields(new Fields(colNames))
    .withTimeAsPartitionField("YYYY/MM/DD");
...
```

### 2. Instantiate a HiveOptions class with the HiveMapper Implementation

Use the HiveOptions class to configure the transactions used by Hive to ingest the streaming data, as illustrated in the following code sample.

```
...
HiveOptions hiveOptions = new
    HiveOptions(metaStoreURI, dbName, tblName, mapper)
    .withTxnsPerBatch(10)
    .withBatchSize(1000)
```

```
.withIdleTimeout(10);
...
```

See "HiveOptions Class Configuration Properties" for a list of configuration properties for the HiveOptions class.

3. Instantiate the HiveBolt with the HiveOptions class:

```
...
StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
TridentState state = stream.partitionPersist(factory, hiveFields, new
    HiveUpdater(),
    new Fields());
...
```

4. Before building your topology code, add the following dependency to your topology pom.xml file:

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpClient</artifactId>
  <version>4.3.3</version>
</dependency>
```

## Configuring Connectors for a Secure Cluster

If your topology uses KafkaSpout, Storm-HDFS, Storm-HBase, or Storm-Hive to access components on a Kerberos-enabled cluster, complete the associated configuration steps listed in this subsection.

### Configuring KafkaSpout for a Secure Kafka Cluster

To connect to a Kerberized Kafka topic:

#### Procedure

1. Code: Add `spoutConfig.securityProtocol=PLAINTEXTSASL` to your Kafka Spout configuration.
2. Configuration: Add a `KafkaClient` section (excerpted from `/usr/hdp/current/kafka-broker/config/kafka_jaas.conf`) to `/usr/hdp/current/storm-supervisor/conf/storm_jaas.conf`:

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    keyTab="/etc/security/keytabs/stormusr.service.keytab"
    storeKey=true
    useTicketCache=false
    serviceName="kafka"
    principal="stormusr/host.name@EXAMPLE.COM";
};
```

3. Setup: Add a Kafka ACL for the topic. For example:

```
bin/kafka-acls.sh --authorizer kafka.security.auth.SimpleAclAuthorizer --authorizer-properties
zookeeper.connect=localhost:2181 --add --allow-principal user:stormusr --allow-hosts * --operations Read --topic
TEST
```

### Configuring Storm-HDFS for a Secure Cluster

To use the `storm-hdfs` connector in topologies that run on secure clusters:

#### Procedure

1. Provide your own Kerberos keytab and principal name to the connectors. The Config object that you pass into the topology must contain the storm keytab file and principal name.

- Specify an HdfsBolt configKey, using the method HdfsBolt.withConfigKey("somekey"). The value map of this key should have the following two properties:

```
hdfs.keytab.file: "<path-to-keytab>"
```

```
hdfs.kerberos.principal: "<principal>@<host>"
```

where

<path-to-keytab> specifies the path to the keytab file on the supervisor hosts

<principal>@<host> specifies the user and domain; for example, storm-admin@EXAMPLE.com.

For example:

```
Config config = new Config();
config.put(HdfsSecurityUtil.STORM_KEYTAB_FILE_KEY, "$keytab");
config.put(HdfsSecurityUtil.STORM_USER_NAME_KEY, "$principal");

StormSubmitter.submitTopology("$topologyName", config,
    builder.createTopology());
```

On worker hosts the bolt/trident-state code will use the keytab file and principal to authenticate with the NameNode. Make sure that all workers have the keytab file, stored in the same location.



**Note:**

For more information about the HdfsBolt class, refer to the Apache Storm [HdfsBolt API documentation](#).

- Distribute the keytab file that the Bolt is using in the Config object, to all supervisor nodes. This is the keytab that is being used to authenticate to HDFS, typically the Storm service keytab, storm. The user ID that the Storm worker is running under should have access to it.

On an Ambari-managed cluster this is /etc/security/keytabs/storm.service.keytab (the "path-to-keytab"), where the worker runs under storm.

- If you set supervisor.run.worker.as.user to true (see [Running Workers as Users](#) in Configuring Storm for Kerberos over Ambari), make sure that the user that the workers are running under (typically the storm keytab) has read access on those keytabs. This is a manual step; an admin needs to go to each supervisor node and run chmod to give file system permissions to the users on these keytab files.



**Note:**

You do not need to create separate keytabs or principals; the general guideline is to create a principal and keytab for each group of users that requires the same access to these resources, and use that single keytab.

All of these connectors accept topology configurations. You can specify the keytab location on the host and the principal through which the connector will login to that system.

- Configure the connector(s). Here is a sample configuration for the Storm-HDFS connector (see [Writing Data to HDFS](#) for a more extensive example):

```
HdfsBolt bolt = new HdfsBolt()
    .withFsUrl("hdfs://localhost:8020")
    .withFileNameFormat(fileNameFormat)
    .withRecordFormat(format)
    .withRotationPolicy(rotationPolicy)
    .withSyncPolicy(syncPolicy);
bolt.withConfigKey("hdfs.config");

Map<String, Object> map = new HashMap<String, Object>();
map.put("hdfs.keytab.file", "/etc/security/keytabs/storm.service.keytab");
map.put("hdfs.kerberos.principal", "storm@TEST.HORTONWORKS.COM");

Config config = new Config();
config.put("hdfs.config", map);
```

```
StormSubmitter.submitTopology("$topologyName", config, builder.createTopology());
```

**Important:**

For the Storm-HDFS connector, you must package `hdfs-site.xml` and `core-site.xml` (from your cluster configuration) in the `topology.jar` file.

In addition, include any configuration files for HDP components used in your Storm topology, such as `hive-site.xml` and `hbase-site.xml`. This fulfills the requirement that all related configuration files appear in the CLASSPATH of your Storm topology at runtime.

## Configuring Storm-HBase for a Secure Cluster

To use the `storm-hbase` connector in topologies that run on secure clusters:

### Procedure

1. Provide your own Kerberos keytab and principal name to the connectors. The Config object that you pass into the topology must contain the storm keytab file and principal name.
2. Specify an `HBaseBolt` `configKey`, using the method `HBaseBolt.withConfigKey("somekey")`. The value map of this key should have the following two properties:

```
storm.keytab.file: "<path-to-keytab-file>"
```

```
storm.kerberos.principal: "<principal>@<host>"
```

For example:

```
Config config = new Config();
config.put(HBaseSecurityUtil.STORM_KEYTAB_FILE_KEY, "$keytab");
config.put(HBaseSecurityUtil.STORM_USER_NAME_KEY, "$principal");

StormSubmitter.submitTopology("$topologyName", config,
    builder.createTopology());
```

On worker hosts the bolt/trident-state code will use the keytab file and principal to authenticate with the NameNode. Make sure that all workers have the keytab file, stored in the same location.

**Note:**

For more information about the `HBaseBolt` class, refer to the Apache Storm [HBaseBolt API documentation](#).

3. Distribute the keytab file that the Bolt is using in the Config object, to all supervisor nodes. This is the keytab that is being used to authenticate to HBase, typically the Storm service keytab, `storm`. The user ID that the Storm worker is running under should have access to it.

**Note:**

You do not need to create separate keytabs or principals; the general guideline is to create a principal and keytab for each group of users that requires the same access to these resources, and use that single keytab.

All of these connectors accept topology configurations. You can specify the keytab location on the host and the principal through which the connector will login to that system.

4. If you set `supervisor.run.worker.as.user` to `true`, make sure that the user that the workers are running under (typically the storm keytab) has read access on those keytabs. This is a manual step; an admin needs to go to each supervisor node and run `chmod` to give file system permissions to the users on these keytab files.

**Note:**

You do not need to create separate keytabs or principals; the general guideline is to create a principal and keytab for each group of users that requires the same access to these resources, and use that single keytab.

All of these connectors accept topology configurations. You can specify the keytab location on the host and the principal through which the connector will login to that system.

5. Configure the connector(s). Here is a sample configuration for the Storm-HBase connector:

```
HBaseBolt hbase = new HBaseBolt("WordCount",
    mapper).withConfigKey("hbase.config");

Map<String, Object> mapHbase = new HashMap<String, Object>();
mapHbase.put("storm.keytab.file", "/etc/security/keytabs/
storm.service.keytab");
mapHbase.put("storm.kerberos.principal", "storm@TEST.HORTONWORKS.COM");

Config config = new Config();
config.put("hbase.config", mapHbase);

StormSubmitter.submitTopology("$topologyName", config, builder.createTopology());
```

### What to do next

For the Storm-HBase connector, you must package `hdfs-site.xml`, `core-site.xml`, and `hbase-site.xml` (from your cluster configuration) in the topology `.jar` file.

In addition, include any other configuration files for HDP components used in your Storm topology, such as `hive-site.xml`. This fulfills the requirement that all related configuration files appear in the `CLASSPATH` of your Storm topology at runtime.

## Configuring Storm-Hive for a Secure Cluster

The Storm-Hive connector accepts configuration settings as part of the `HiveOptions` class. For more information about the `HiveBolt` and `HiveOptions` classes, see the Apache Storm [HiveOptions](#) and [HiveBolt](#) API documentation.

### About this task

There are two required settings for accessing secure Hive:

### Procedure

1. `withKerberosPrincipal`, the Kerberos principal for accessing Hive:

```
public HiveOptions withKerberosPrincipal(String kerberosPrincipal)
```

2. `withKerberosKeytab`, the Kerberos keytab for accessing Hive:

```
public HiveOptions withKerberosKeytab(String kerberosKeytab)
```