# Configuring a Streams Application

Apache Kafka® and Kafka Streams configuration options must be configured before using Streams. You can configure Kafka Streams by specifying parameters in a `java.util.Properties` instance.

1. Create a `java.util.Properties` instance.
2. Set the parameters. For example:

```
import java.util.Properties;
import org.apache.kafka.streams.StreamsConfig;

Properties props = new Properties();
// Set a few key parameters
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-first-streams-application");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");
// Any further settings
props.put(... , ...);
```

## Configuration parameter reference

This section contains the most common Streams configuration parameters. For a full reference, see the Streams and Client Javadocs.

### Required configuration parameters

Here are the required Streams configuration parameters.

| Parameter Name | Importance | Description | Default Value |
|---|---|---|---|
| application.id | Required | An identifier for the stream processing application. Must be unique within the Kafka cluster. | None |
| bootstrap.servers | Required | A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. | None |

#### application.id

(Required) The application ID. Each stream processing application must have a unique ID. The same ID must be given to all instances of the application. It is recommended to use only alphanumeric characters, `.` (dot), `-` (hyphen), and `_` (underscore). Examples: `"hello_world"`, `"hello_world-v1.0.0"`

This ID is used in the following places to isolate resources used by the application from others:

- As the default Kafka consumer and producer `client.id` prefix
- As the Kafka consumer `group.id` for coordination
- As the name of the subdirectory in the state directory (cf. `state.dir`)
- As the prefix of internal Kafka topic names

**Tip:**

When an application is updated, the `application.id` should be changed unless you want to reuse the existing data in internal topics and state stores. For example, you could embed the version information within `application.id`, as `my-app-v1.0.0` and `my-app-v1.0.`

Expand
Content

### bootstrap.servers

(Required) The Kafka bootstrap servers. This is the samesetting that is used by the underlying producer and consumer clients to connect to the Kafka cluster. Example: `"kafka-broker1:9092,kafka-broker2:9092"`.

**Tip:**

Kafka Streams applications can only communicate with a single Kafka cluster specified by this config value. Future versions of Kafka Streams will support connecting to different Kafka clusters for reading input streams and writing output streams.

## Optional configuration parameters

Here are the optionalStreams configuration parameters, sorted by level of importance:

- High: These parameters can have a significant impact on performance. Take care when deciding the values of these parameters.
- Medium: These parameters can have some impact on performance. Your specific environment will determine how much tuning effort should be focused on these parameters.
- Low: These parameters have a less general or less significant impact on performance.

| Parameter Name | Importance | Description | Default Value |
|---|---|---|---|
| application.server | Low | A host:port pair pointing to an embedded user defined endpoint that can be used for discovering the locations of state stores within a single Kafka Streams application. The value of this must be different for each instance of the application. | the empty string |
| buffered.records.per.partition | Low | The maximum number of records to buffer per partition. | 1000 |
| cache.max.bytes.buffering | Medium | Maximum number of memory bytes to be used for record caches across all threads. | 10485760 bytes |
| client.id | Medium | An ID string to pass to the server when making requests. (This setting is passed to the consumer/producer clients used internally by Kafka Streams.) | the empty string |
| commit.interval.ms | Low | The frequency with which to save the position (offsets in source topics) of tasks. | 30000 milliseco milliseconds (ex |
| default.deserialization.exception.handler | Medium | Exception handling class that implements the `DeserializationExceptionHandler` interface. | See default.deseriali |
| default.production.exception.handler | Medium | Exception handling class that implements the `ProductionExceptionHandler` interface. | See default.prod |
| default.key.serde | Medium | Default serializer/deserializer class for record keys, | `Serdes.ByteArra` |

| Parameter Name | Importance | Description | Default Value |
|---|---|---|---|
| | | implements the `Serde` interface (see also value.serde). | |
| default.value.serde | Medium | Default serializer/deserializer class for record values, implements the `Serde` interface (see also key.serde). | `Serdes.ByteArra` |
| default.windowed.key.serde.inner | Medium | Default inner serializer/deserializer class for record keys, implements the `Serde` interface. Only affective if `default.key.serde` is a windowed serde. | `Serdes.ByteArra` |
| default.windowed.value.serde.inner | Medium | Default inner serializer/deserializer class for record values, implements the `Serde` interface Only affective if `default.value.serde` is a windowed serde. | `Serdes.ByteArra` |
| default.timestamp.extractor | Medium | Default timestamp extractor class that implements the `TimestampExtractor` interface. | See Timestamp |
| max.task.idle.ms | Medium | Maximum amount of time a stream task will stay idle when not all of its partition buffers contain records. | 0 milliseconds |
| metric.reporters | Low | A list of classes to use as metrics reporters. | the empty list |
| metrics.num.samples | Low | The number of samples maintained to compute metrics. | 2 |
| metrics.recording.level | Low | The highest recording level for metrics. | `INFO` |
| metrics.sample.window.ms | Low | The window of time a metrics sample is computed over. | 30000 milliseco |
| num.standby.replicas | Medium | The number of standby replicas for each task. | 0 |
| num.stream.threads | Medium | The number of threads to execute stream processing. | 1 |
| partition.grouper | Low | Partition grouper class that implements the `PartitionGrouper` interface. | See Partition Gr |
| poll.ms | Low | The amount of time in milliseconds to block waiting for input. | 100 millisecond |
| processing.guarantee | Medium | The processing mode. Can be either at-least-once (default) or exactly-once. | See Processing |

| Parameter Name | Importance | Description | Default Value |
|---|---|---|---|
| replication.factor | High | The replication factor for changelog topics and repartition topics created by the application. | 1 |
| retries | Medium | The number of retries for broker requests that return a retryable error. | 0 |
| retry.backoff.ms | Medium | The amount of time in milliseconds, before a request is retried. This applies if the `retries` parameter is configured to be greater than 0. | 100 |
| state.cleanup.delay.ms | Low | The amount of time in milliseconds to wait before deleting state when a partition has migrated. | 6000000 millise |
| state.dir | High | Directory location for state stores. | `/var/lib/kafka-` |
| topology.optimization | Low | Enables/Disables topology optimization. Accepts strings `none` or `all`. | `none` |
| windowstore.changelog.additional.retention.ms | Low | Added to a windows maintainMs to ensure data is not deleted from the log prematurely. Allows for clock drift. | 86400000 millis |

## default.deserialization.exception.handler

The default deserialization exception handler allows you to manage record exceptions that fail to deserialize. This can be caused by corrupt data, incorrect serialization logic, or unhandled record types. The implemented exception handler needs to return a `FAIL` or `CONTINUE` depending on the record and the exception thrown. Returning `FAIL` will signal that Streams should shut down and `CONTINUE` will signal that Streams should ignore the issue and continue processing. The default implemention class is LogAndFailExceptionHandler. These exception handlers are available:

- LogAndContinueExceptionHandler: This handler logs the deserialization exception and then signals the processing pipeline to continue processing more records. This log-and-skip strategy allows Kafka Streams to make progress instead of failing if there are records that fail to deserialize.
- LogAndFailExceptionHandler. This handler logs the deserialization exception and then signals the processing pipeline to stop processing more records.

You can also provide your own customized exception handler besides the library provided ones to meet your needs. For an example customized exception handler implementation, please read the Failure and exception handling FAQ.

## default.production.exception.handler

The default production exception handler allows you to manage exceptions triggered when trying to interact with a broker such as attempting to produce a record that is too large. By default, Kafka provides and uses the DefaultProductionExceptionHandler that always fails when these exceptions occur.

Each exception handler can return a `FAIL` or `CONTINUE` depending on the record and the exception thrown. Returning `FAIL` will signal that

Streams should shut down and `CONTINUE` will signal that Streams should ignore the issue and continue processing. If you want to provide an exception handler that always ignores records that are too large, you could implement something like the following:

```
import java.util.Properties;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.common.errors.RecordTooLargeException;
import org.apache.kafka.streams.errors.ProductionExceptionHandler;
import org.apache.kafka.streams.errors.ProductionExceptionHandler.ProductionExceptionHandlerResponse;

class IgnoreRecordTooLargeHandler implements ProductionExceptionHandler {
    public void configure(Map&lt;String, Object&gt; config) {}

    public ProductionExceptionHandlerResponse handle(final ProducerRecord&lt;byte[], byte[]&gt; record,
                                                     final Exception exception) {
        if (exception instanceof RecordTooLargeException) {
            return ProductionExceptionHandlerResponse.CONTINUE;
        } else {
            return ProductionExceptionHandlerResponse.FAIL;
        }
    }
}

Properties settings = new Properties();

// other various kafka streams settings, e.g. bootstrap servers, application ID, etc

settings.put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_CLASS_CONFIG,
             IgnoreRecordTooLargeHandler.class);
```

## default.key.serde

The default Serializer/Deserializer class for record keys. Serialization and deserialization in Kafka Streams happens whenever data needs to be materialized, for example:

- Whenever data is read from or written to a*Kafka topic*(e.g., via the `StreamsBuilder#stream()` and `KStream#to()` methods).
- Whenever data is read from or written to a*state store*.

This is discussed in more detail inData types and serialization.

## default.value.serde

The default Serializer/Deserializer class for record values. Serialization and deserialization in Kafka Streams happens whenever data needs to be materialized, for example:

- Whenever data is read from or written to a*Kafka topic*(e.g., via the `KStreamBuilder#stream()` and `KStream#to()` methods).
- Whenever data is read from or written to a*state store*.

This is discussed in more detail inData types and serialization.

## default.timestamp.extractor

A timestamp extractor pulls a timestamp from an instance ofConsumerRecord. Timestamps are used to control the progress of streams.

The default extractor isFailOnInvalidTimestamp. This extractor retrieves built-in timestamps that are automatically embedded into Kafka messages by the Kafka producer client since Kafka version 0.10. Depending on the setting of Kafka's server-side `log.message.timestamp.type` broker and `message.timestamp.type` topic parameters, this extractor provides you with:

- **event-time** processing semantics if `log.message.timestamp.type` is set to `CreateTime` aka "producer time" (which is the default). This represents the time when a Kafka producer sent the original message. If you use Kafka's official producer client or one of Confluent's producer clients, the timestamp represents milliseconds since the epoch.
- **ingestion-time** processing semantics if `log.message.timestamp.type` is set to `LogAppendTime` aka "broker time". This represents the time when the Kafka broker received the original message, in milliseconds since the epoch.

The `FailOnInvalidTimestamp` extractor throws an exception if a record contains an invalid (i.e. negative) built-in timestamp, because Kafka

Streams would not process this record but silently drop it. Invalid built-in timestamps can occur for various reasons: if for example, you consume a topic that is written to by pre-0.10 Kafka producer clients or by third-party producer clients that don't support the new Kafka 0.10 message format yet; another situation where this may happen is after upgrading your Kafka cluster from `0.9` to `0.10`, where all the data that was generated with `0.9` does not include the `0.10` message timestamps.

If you have data with invalid timestamps and want to process it, then there are two alternative extractors available. Both work on built-in timestamps, but handle invalid timestamps differently.

- LogAndSkipOnInvalidTimestamp: This extractor logs a warn message and returns the invalid timestamp to Kafka Streams, which will not process but silently drop the record. This log-and-skip strategy allows Kafka Streams to make progress instead of failing if there are records with an invalid built-in timestamp in your input data.
- UsePreviousTimeOnInvalidTimestamp. This extractor returns the record's built-in timestamp if it is valid (i.e. not negative). If the record does not have a valid built-in timestamps, the extractor returns the previously extracted valid timestamp from a record of the same topic partition as the current record as a timestamp estimation. In case that no timestamp can be estimated, it throws an exception.

Another built-in extractor is WallclockTimestampExtractor. This extractor does not actually "extract" a timestamp from the consumed record but rather returns the current time in milliseconds from the system clock (think: `System.currentTimeMillis()`), which effectively means Streams will operate on the basis of the so-called **processing-time** of events.

You can also provide your own timestamp extractors, for instance to retrieve timestamps embedded in the payload of messages. If you cannot extract a valid timestamp, you can either throw an exception, return a negative timestamp, or estimate a timestamp. Returning a negative timestamp will result in data loss -- the corresponding record will not be processed but silently dropped. If you want to estimate a new timestamp, you can use the value provided via `previousTimestamp` (i.e., a Kafka Streams timestamp estimation). Here is an example of a custom `TimestampExtractor` implementation:

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.streams.processor.TimestampExtractor;

// Extracts the embedded timestamp of a record (giving you "event-time" semantics).
public class MyEventTimeExtractor implements TimestampExtractor {

  @Override
  public long extract(final ConsumerRecord<Object, Object> record, final long previousTimestamp) {
    // `Foo` is your own custom class, which we assume has a method that returns
    // the embedded timestamp (milliseconds since midnight, January 1, 1970 UTC).
    long timestamp = -1;
    final Foo myPojo = (Foo) record.value();
    if (myPojo != null) {
      timestamp = myPojo.getTimestampInMillis();
    }
    if (timestamp < 0) {
      // Invalid timestamp!  Attempt to estimate a new timestamp,
      // otherwise fall back to wall-clock time (processing-time).
      if (previousTimestamp >= 0) {
        return previousTimestamp;
      } else {
        return System.currentTimeMillis();
      }
    }
    return timestamp;
  }

}
```

You would then define the custom timestamp extractor in your Streams configuration as follows:

```
import java.util.Properties;
import org.apache.kafka.streams.StreamsConfig;

Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, MyEventTimeExtractor.class);
```

### num.standby.replicas

The number of standby replicas. Standby replicas are shadow copies of local state stores. Kafka Streams attempts to create the specified number of replicas per store and keep them up to date as long as there are enough instances running. Standby replicas are used to minimize the latency of task failover. A task that was previously running on a failed instance is preferred to restart on an instance that has standby replicas so that the local state store restoration process from its changelog can be minimized. Details about how Kafka Streams makes use of the standby replicas to minimize the cost of resuming tasks on failover can be found in the State section.

**Note**

If you configure *n* standby replicas, you need to provision *n+1* `KafkaStreams` instances.

### num.stream.threads

This specifies the number of stream threads in an instance of the Kafka Streams application. The stream processing code runs in these threads. For more info about the Kafka Streams threading model, see Threading Model.

### partition.grouper

A partition grouper creates a list of stream tasks from the partitions of source topics, where each created task is assigned with a group of source topic partitions. The default implementation provided by Kafka Streams is DefaultPartitionGrouper. It assigns each task with one partition for each of the source topic partitions. The generated number of tasks equals the largest number of partitions among the input topics. Usually an application does not need to customize the partition grouper.

### processing.guarantee

The processing guarantee that should be used. Possible values are `"at_least_once"` (default) and `"exactly_once"`. Note that if exactly-once processing is enabled, the default for parameter `commit.interval.ms` changes to `100ms`. Additionally, consumers are configured with `isolation.level="read_committed"` and producers are configured with `retries=Integer.MAX_VALUE` and `enable.idempotence=true` per default. Note that by default exactly-once processing requires a cluster of at least three brokers what is the recommended setting for production. For development you can change this, by adjusting broker setting `transaction.state.log.replication.factor` to the number of broker you want to use. For more details see Processing Guarantees.

### replication.factor

This specifies the replication factor of internal topics that Kafka Streams creates when local states are used or a stream is repartitioned for aggregation. Replication is important for fault tolerance. Without replication even a single broker failure may prevent progress of the stream processing application. It is recommended to use a similar replication factor as source topics.

**Recommendation:**

Increase the replication factor to 3 to ensure that the internal Kafka Streams topic can tolerate up to 2 broker failures. Note that you will require more storage space as well (3 times more with the replication factor of 3).

### state.dir

The state directory. Kafka Streams persists local states under the state directory. Each application has a subdirectory on its hosting machine that is located under the state directory. The name of the subdirectory is the application ID. The state stores associated with the application are created under this subdirectory.

### max.task.idle.ms

Maximum amount of time a stream task will stay idle when not all of its partition buffers contain records, to avoid potential out-of-order record processing across multiple input streams. When only a subset of a certain task's input topic partitions have data available to be processed, the task would not know what's the timestamp of the next record from those empty partitions, and hence continue processing those available partitions' records have a risk of out-of-order data processing: i.e. records with older timestamp may be received later and get processed after other records with newer timestamp. Setting this config to a larger value would allow application to trade some processing latency to reduce likelihood of out-of-order data processing by holding on processing the existing available records but keep fetching for the empty topic partitions.

## Kafka consumers, producer, and admin client configuration parameters

You can specify parameters for the Kafkaconsumers, producers, and admin client that are used internally. The consumer, producer, and admin client settings are defined by specifying parameters in a `StreamsConfig` instance.

In this example, the Kafkaconsumer session timeout is configured to be 60000 milliseconds in the Streams settings:

```
Properties streamsSettings = new Properties();
// Example of a "normal" setting for Kafka Streams
streamsSettings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker-01:9092");
// Customize the Kafka consumer settings of your Streams application
streamsSettings.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 60000);
```

### Naming

Some consumer, producer, and admin client configuration parameters use the same parameter name. For example `send.buffer.bytes` and `receive.buffer.bytes` are used to configure TCP buffers; `request.timeout.ms` and `retry.backoff.ms` control retries for client request. You can avoid duplicate names by prefix parameter names with `consumer.`, `producer.`, or `admin.` (e.g., `consumer.send.buffer.bytes` or `producer.send.buffer.bytes`).

```
Properties streamsSettings = new Properties();
// same value for consumer and producer
streamsSettings.put("PARAMETER_NAME", "value");
// different values for consumer, producer, and admin client
streamsSettings.put("consumer.PARAMETER_NAME", "consumer-value");
streamsSettings.put("producer.PARAMETER_NAME", "producer-value");
streamsSettings.put("admin.PARAMETER_NAME", "admin-value");
// alternatively, you can use
streamsSettings.put(StreamsConfig.consumerPrefix("PARAMETER_NAME"), "consumer-value");
streamsSettings.put(StreamsConfig.producerPrefix("PARAMETER_NAME"), "producer-value");
streamsSettings.put(StreamsConfig.adminClientPrefix("PARAMETER_NAME"), "admin-value");
```

### Default Values

Kafka Streams uses different default values for some of the underlying client configs, which are summarized below. For detailed descriptions of these configs, see Producer Configurations and Consumer Configurations.

| Parameter Name | Corresponding Client | Streams Default |
|---|---|---|
| auto.offset.reset | Global Consumer | none (cannot be changed) |
| auto.offset.reset | Restore Consumer | none (cannot be changed) |
| auto.offset.reset | Consumer | earliest |
| enable.auto.commit | Consumer | false (cannot be changed) |
| linger.ms | Producer | 100 |
| max.poll.interval.ms | Consumer | 300000 |
| max.poll.records | Consumer | 1000 |
| retries | Producer | 10 |
| rocksdb.config.setter | Consumer | |

### enable.auto.commit

The consumer auto commit. To guarantee at-least-once processing semantics and turn off auto commits, Kafka Streams overrides this consumer config value to `false`. Consumers will only commit explicitly via *commitSync* calls when the Kafka Streams library or a user decides to commit the current processing state.

### rocksdb.config.setter

The RocksDB configuration. Kafka Streams uses RocksDB as the default storage engine for persistent stores. To change the default configuration for RocksDB, implement `RocksDBConfigSetter` and provide your custom class via rocksdb.config.setter.

Here is an example that adjusts the memory size consumed by RocksDB.

```
public static class CustomRocksDBConfig implements RocksDBConfigSetter {

  // This object should be a member variable so it can be closed in RocksDBConfigSetter#close.
  private org.rocksdb.Cache cache = new org.rocksdb.LRUCache(16 * 1024L * 1024L);

  @Override
  public void setConfig(final String storeName, final Options options, final Map<String, Object> configs) {
    // See #1 below.
    BlockBasedTableConfig tableConfig = (BlockBasedTableConfig) options.tableFormatConfig();
    tableConfig.setBlockCache(cache);
    // See #2 below.
    tableConfig.setBlockSize(16 * 1024L);
    // See #3 below.
    tableConfig.setCacheIndexAndFilterBlocks(true);
    options.setTableFormatConfig(tableConfig);
    // See #4 below.
    options.setMaxWriteBufferNumber(2);
  }

  @Override
  public void close(final String storeName, final Options options) {
    // See #5 below.
    cache.close();
  }
}

Properties streamsSettings = new Properties();
streamsConfig.put(StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG, CustomRocksDBConfig.class);
```

**Notes for example:**

1. `BlockBasedTableConfig tableConfig = (BlockBasedTableConfig) options.tableFormatConfig();` Get a reference to the existing `TableFormatConfig` rather than create a new one so you don't accidentally overwrite defaults such as the `BloomFilter`, an important optimization.
2. `tableConfig.setBlockSize(16 * 1024L);` Modify the default per these instructions from the RocksDB GitHub (indexes and filter blocks).
3. `tableConfig.setCacheIndexAndFilterBlocks(true);` Do not let the index and filter blocks grow unbounded. For more information, see the RocksDB GitHub (caching index and filter blocks).
4. `options.setMaxWriteBufferNumber(2);` See the advanced options in the RocksDB GitHub.
5. `cache.close();` To avoid memory leaks, you must close any objects you constructed that extend org.rocksdb.RocksObject. See RocksJava docs for more details.

## Recommended configuration parameters for resiliency

There are several Kafka and Kafka Streams configuration options that need to be configured explicitly for resiliency in face of broker failures:

| Parameter Name | Corresponding Client | Default value | Consider setting to |
|---|---|---|---|
| acks | Producer | `acks=1` | `acks=all` |
| replication.factor | Streams | `1` | `3` |
| min.insync.replicas | Broker | `1` | `2` |

Increasing the replication factor to 3 ensures that the internal Kafka Streams topic can tolerate up to 2 broker failures. Changing the acks setting to "all" guarantees that a record will not be lost as long as one replica is alive. The tradeoff from moving to the default values to the recommended ones is that some performance and more storage space (3x with the replication factor of 3) are sacrificed for more resiliency.

### acks

The number of acknowledgments that the leader must have received before considering a request complete. This controls the durability of records that are sent. The possible values are:

- `acks=0` The producer does not wait for acknowledgment from the server and the record is immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the `retries` configuration will not take effect (as the client won't generally know of any failures). The offset returned for each record will always be set to `-1`.
- `acks=1` The leader writes the record to its local log and responds without waiting for full acknowledgement from all followers. If the leader immediately fails after acknowledging the record, but before the followers have replicated it, then the record will be lost.
- `acks=all` The leader waits for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost if there is at least one in-sync replica alive. This is the strongest available guarantee.

For more information, see the Kafka Producer documentation.

### replication.factor

See the description here.

You define these settings via `StreamsConfig`:

```
Properties streamsSettings = new Properties();
streamsSettings.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 3);
streamsSettings.put(StreamsConfig.producerPrefix(ProducerConfig.ACKS_CONFIG), "all");
```

**Note**

A future version of Kafka Streams will allow developers to set their own app-specific configuration settings through the `Properties` instance as well, which can then be accessed through ProcessorContext.

Last updated on Sep 10, 2019.