Spring Cloud Stream Kafka Binder Reference Guide

1.1.0.BUILD-SNAPSHOT

Copyright © 2013-2016Pivotal Software, Inc.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

Part I. Reference Guide

This guide describes the Apache Kafka implementation of the Spring Cloud Stream Binder. It contains information about its design, usage and configuration options, as well as information on how the Stream Cloud Stream concepts map into Apache Kafka specific constructs.

1. Usage

For using the Apache Kafka binder, you just need to add it to your Spring Cloud Stream application, using the following Maven coordinates:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

Alternatively, you can also use the Spring Cloud Stream Kafka Starter.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

2. Apache Kafka Binder Overview

A simplified diagram of how the Apache Kafka binder operates can be seen below.

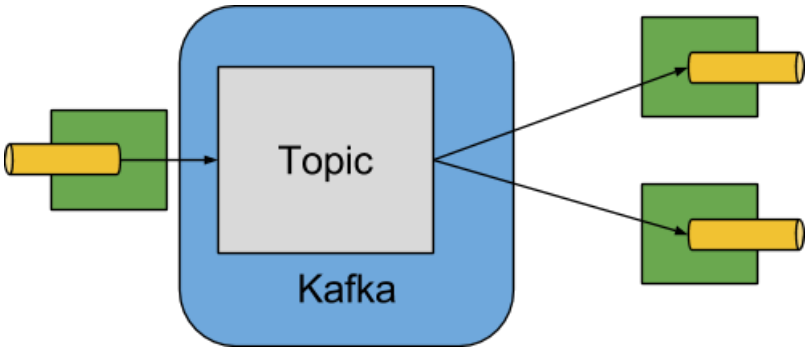

Figure 2.1. Kafka Binder

The Apache Kafka Binder implementation maps each destination to an Apache Kafka topic. The consumer group maps directly to the same Apache Kafka concept. Partitioning also maps directly to Apache Kafka partitions as well.

3. Configuration Options

This section contains the configuration options used by the Apache Kafka binder.

For common configuration options and properties pertaining to binder, refer to the core docs.

3.1 Kafka Binder Properties

spring.cloud.stream.kafka.binder.brokers
: A list of brokers to which the Kafka binder will connect.

  Default: `localhost`.

spring.cloud.stream.kafka.binder.defaultBrokerPort
: `brokers` allows hosts specified with or without port information (e.g., `host1,host2:port2`). This sets the default port when no port is configured in the broker list.

  Default: `9092`.

spring.cloud.stream.kafka.binder.zkNodes
: A list of ZooKeeper nodes to which the Kafka binder can connect.

  Default: `localhost`.

spring.cloud.stream.kafka.binder.defaultZkPort
: `zkNodes` allows hosts specified with or without port information (e.g., `host1,host2:port2`). This sets the default port when no port is configured in the node list.

  Default: `2181`.

spring.cloud.stream.kafka.binder.configuration
: Key/Value map of client properties (both producers and consumer) passed to all clients created by the binder. Due to the fact that these properties will be used by both producers and consumers, usage should be restricted to common properties, especially security settings.

  Default: Empty map.

spring.cloud.stream.kafka.binder.headers
: The list of custom headers that will be transported by the binder.

  Default: empty.

spring.cloud.stream.kafka.binder.offsetUpdateTimeWindow
: The frequency, in milliseconds, with which offsets are saved. Ignored if `0`.

  Default: `10000`.

spring.cloud.stream.kafka.binder.offsetUpdateCount
: The frequency, in number of updates, which which consumed offsets are persisted. Ignored if `0`. Mutually exclusive with `offsetUpdateTimeWindow`.

  Default: `0`.

spring.cloud.stream.kafka.binder.requiredAcks
: The number of required acks on the broker.

  Default: `1`.

spring.cloud.stream.kafka.binder.minPartitionCount
: Effective only if `autoCreateTopics` or `autoAddPartitions` is set. The global minimum number of partitions that the binder will configure on topics on which it produces/consumes data. It can be superseded by the `partitionCount` setting of the producer or by the value of `instanceCount` * `concurrency` settings of the producer (if either is larger).

  Default: `1`.

spring.cloud.stream.kafka.binder.replicationFactor
: The replication factor of auto-created topics if `autoCreateTopics` is active.

  Default: `1`.

spring.cloud.stream.kafka.binder.autoCreateTopics
: If set to `true`, the binder will create new topics automatically. If set to `false`, the binder will rely on the topics being already configured. In the latter case, if the topics do not exist, the binder will fail to start. Of note, this setting is independent of the `auto.topic.create.enable` setting of the broker and it does not influence it: if the server is set to auto-create topics, they may be created as part of the metadata retrieval request, with default broker settings.

  Default: `true`.

spring.cloud.stream.kafka.binder.autoAddPartitions
: If set to `true`, the binder will create add new partitions if required. If set to `false`, the binder will rely on the partition size of the topic being already configured. If the partition count of the target topic is smaller than the expected value, the binder will fail to start.

  Default: `false`.

spring.cloud.stream.kafka.binder.socketBufferSize
: Size (in bytes) of the socket buffer to be used by the Kafka consumers.

  Default: `2097152`.

3.2 Kafka Consumer Properties

The following properties are available for Kafka consumers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.consumer.`.

autoRebalanceEnabled
: When `true`, topic partitions will be automatically rebalanced between the members of a consumer group. When `false`, each consumer will be assigned a fixed set of partitions based on `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex`. This requires both `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties to be set appropriately on each launched instance. The property `spring.cloud.stream.instanceCount` must typically be greater than 1 in this case.

  Default: `true`.

autoCommitOffset
: Whether to autocommit offsets when a message has been processed. If set to `false`, a header with the key `kafka_acknowledgment` of the type `org.springframework.kafka.support.Acknowledgment` header will be present in the inbound message. Applications may use this header for acknowledging messages. See the examples section for details. When this property is set to `false`, Kafka binder will set the ack mode to `org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL`.

  Default: `true`.

autoCommitOnError
: Effective only if `autoCommitOffset` is set to `true`. If set to `false` it suppresses auto-commits for messages that result in errors, and will commit only for successful messages, allows a stream to automatically replay from the last successfully processed message, in case of persistent failures. If set to `true`, it will always auto-commit (if auto-commit is enabled). If not set (default), it effectively has the same value as `enableDlq`, auto-committing erroneous messages if they are sent to a DLQ, and not committing them otherwise.

  Default: not set.

recoveryInterval
: The interval between connection recovery attempts, in milliseconds.

  Default: `5000`.

resetOffsets
: Whether to reset offsets on the consumer to the value provided by `startOffset`.

  Default: `false`.

startOffset
: The starting offset for new groups, or when `resetOffsets` is `true`. Allowed values: `earliest`, `latest`.

  Default: null (equivalent to `earliest`).

enableDlq
: When set to true, it will send enable DLQ behavior for the consumer. Messages that result in errors will be forwarded to a topic named `error.<destination>.<group>`. This provides an alternative option to the more common Kafka replay scenario for the case when the number of errors is relatively small and replaying the entire original topic may be too cumbersome.

  Default: `false`.

configuration
: Map with a key/value pair containing generic Kafka consumer properties.

  Default: Empty map.

3.3 Kafka Producer Properties

The following properties are available for Kafka producers only and must be prefixed with `spring.cloud.stream.kafka.bindings.<channelName>.producer.`.

bufferSize
: Upper limit, in bytes, of how much data the Kafka producer will attempt to batch before sending.

  Default: `16384`.

sync
: Whether the producer is synchronous.

  Default: `false`.

batchTimeout
: How long the producer will wait before sending in order to allow more messages to accumulate in the same batch. (Normally the producer does not wait at all, and simply sends all the messages that accumulated while the previous send was in progress.) A non-zero value may increase throughput at the expense of latency.

  Default: `0`.

configuration
: Map with a key/value pair containing generic Kafka producer properties.

  Default: Empty map.

> **Note**
>
> The Kafka binder will use the `partitionCount` setting of the producer as a hint to create a topic with the given partition count (in conjunction with the `minPartitionCount`, the maximum of the two being the value being used). Exercise caution when configuring both `minPartitionCount` for a binder and `partitionCount` for an application, as the larger value will be used. If a topic already exists with a smaller partition count and `autoAddPartitions` is disabled (the default), then the binder will fail to start. If a topic already exists with a smaller partition count and `autoAddPartitions` is enabled, new partitions will be added. If a topic already exists with a larger number of partitions than the maximum of (`minPartitionCount` and `partitionCount`), the existing partition count will be used.

3.4 Usage examples

In this section, we illustrate the use of the above properties for specific scenarios.

Example: Setting `autoCommitOffset` false and relying on manual acking.

This example illustrates how one may manually acknowledge offsets in a consumer application.

This example requires that `spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset` is set to false. Use the corresponding input channel name for your example.

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class ManuallyAcknowdledgingConsumer {

 public static void main(String[] args) {
     SpringApplication.run(ManuallyAcknowdledgingConsumer.class, args);
 }
```

```
 @StreamListener(Sink.INPUT)
 public void process(Message<?> message) {
     Acknowledgment acknowledgment = message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT,
 Acknowledgment.class);
     if (acknowledgment != null) {
         System.out.println("Acknowledgment provided");
         acknowledgment.acknowledge();
     }
 }
}
```

Example: security configuration

Apache Kafka 0.9 supports secure connections between client and brokers. To take advantage of this feature, follow the guidelines in the Apache Kafka Documentation as well as the Kafka 0.9 security guidelines from the Confluent documentation. Use the `spring.cloud.stream.kafka.binder.configuration` option to set security properties for all clients created by the binder.

For example, for setting `security.protocol` to SASL_SSL, set:

```
spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_SSL
```

All the other security properties can be set in a similar manner.

When using Kerberos, follow the instructions in the reference documentation for creating and referencing the JAAS configuration.

Spring Cloud Stream supports passing JAAS configuration information to the application using a JAAS configuration file and using Spring Boot properties.

Using JAAS configuration files

The JAAS, and (optionally) krb5 file locations can be set for Spring Cloud Stream applications by using system properties. Here is an example of launching a Spring Cloud Stream application with SASL and Kerberos using a JAAS configuration file:

```
java -Djava.security.auth.login.config=/path.to/kafka_client_jaas.conf -jar log.jar \
  --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper:2181 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT
```

Using Spring Boot properties

As an alternative to having a JAAS configuration file, Spring Cloud Stream provides a mechanism for setting up the JAAS configuration for Spring Cloud Stream applications using Spring Boot properties.

The following properties can be used for configuring the login context of the Kafka client.

spring.cloud.stream.kafka.binder.jaas.loginModule
: The login module name. Not necessary to be set in normal cases.

  Default: `com.sun.security.auth.module.Krb5LoginModule`.

spring.cloud.stream.kafka.binder.jaas.controlFlag
: The control flag of the login module.

Default: `required`.

spring.cloud.stream.kafka.binder.jaas.options
: Map with a key/value pair containing the login module options.

Default: Empty map.

Here is an example of launching a Spring Cloud Stream application with SASL and Kerberos using Spring Boot configuration properties:

```
java --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
  --spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper:2181 \
  --spring.cloud.stream.bindings.input.destination=stream.ticktock \
  --spring.cloud.stream.kafka.binder.autoCreateTopics=false \
  --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT \
  --spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=true \
  --spring.cloud.stream.kafka.binder.jaas.options.storeKey=true \
  --spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc/security/keytabs/kafka_client.keytab \
  --spring.cloud.stream.kafka.binder.jaas.options.principal=kafka-client-1@EXAMPLE.COM
```

This represents the equivalent of the following JAAS file:

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
};
```

If the topics required already exist on the broker, or will be created by an administrator, autocreation can be turned off and only client JAAS properties need to be sent. As an alternative to setting `spring.cloud.stream.kafka.binder.autoCreateTopics` you can simply remove the broker dependency from the application. See the section called “Excluding Kafka broker jar from the classpath of the binder based application” for details.

> **Note**
>
> Do not mix JAAS configuration files and Spring Boot properties in the same application. If the `-Djava.security.auth.login.config` system property is already present, Spring Cloud Stream will ignore the Spring Boot properties.

> **Note**
>
> Exercise caution when using the `autoCreateTopics` and `autoAddPartitions` if using Kerberos. Usually applications may use principals that do not have administrative rights in Kafka and Zookeeper, and relying on Spring Cloud Stream to create/modify topics may fail. In secure environments, we strongly recommend creating topics and managing ACLs administratively using Kafka tooling.

Using the binder with Apache Kafka 0.10

The binder also supports connecting to Kafka 0.10 brokers. In order to support this, when you create the project that contains your application, include `spring-cloud-starter-stream-kafka` as you normally would do for 0.9 based applications. Then add these dependencies at the top of the `<dependencies>` section in the pom.xml file to override the Apache Kafka, Spring Kafka, and Spring Integration Kafka with 0.10-compatible versions as in the following example:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>1.1.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-kafka</artifactId>
  <version>2.1.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>0.10.0.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

> **Note**
>
> The versions above are provided only for the sake of the example. For best results, we recommend using the most recent 0.10-compatible versions of the projects.

Excluding Kafka broker jar from the classpath of the binder based application

The Apache Kafka Binder uses the administrative utilities which are part of the Apache Kafka server library to create and reconfigure topics. If the inclusion of the Apache Kafka server library and its dependencies is not necessary at runtime because the application will rely on the topics being configured administratively, the Kafka binder allows for Apache Kafka server dependency to be excluded from the application.

If you use Kafka 10 dependencies as advised above, all you have to do is not to include the kafka broker dependency. If you use Kafka 0.9, then ensure that you exclude the kafka broker jar from the `spring-cloud-starter-stream-kafka` dependency as following.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka_2.11</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

If you exclude the Apache Kafka server dependency and the topic is not present on the server, then the Apache Kafka broker will create the topic if auto topic creation is enabled on the server. Please keep in mind that if you are relying on this, then the Kafka server will use the default number of partitions and replication factors. On the other hand, if auto topic creation is disabled on the server, then care must be taken before running the application to create the topic with the desired number of partitions.

If you want to have full control over how partitions are allocated, then leave the default settings as they are, i.e. do not exclude the kafka broker jar and ensure that `spring.cloud.stream.kafka.binder.autoCreateTopics` is set to `true`, which is the default.

Part II. Appendices

Appendix A. Building

A.1 Basic Compile and Test

To build the source you will need to install JDK 1.7.

The build uses the Maven wrapper so you don't have to install a specific version of Maven. To enable the tests, you should have Kafka server 0.9 or above running before building. See below for more information on running the servers.

The main build command is

```
$ ./mvnw clean install
```

You can also add '-DskipTests' if you like, to avoid running the tests.

> **Note**
>
> You can also install Maven (>=3.3.3) yourself and run the `mvn` command in place of `./mvnw` in the examples below. If you do that you also might need to add `-P spring` if your local Maven settings do not contain repository declarations for spring pre-release artifacts.

> **Note**
>
> Be aware that you might need to increase the amount of memory available to Maven by setting a `MAVEN_OPTS` environment variable with a value like `-Xmx512m -XX:MaxPermSize=128m`. We try to cover this in the `.mvn` configuration, so if you find you have to do it to make a build succeed, please raise a ticket to get the settings added to source control.

The projects that require middleware generally include a `docker-compose.yml`, so consider using Docker Compose to run the middeware servers in Docker containers.

A.2 Documentation

There is a "full" profile that will generate documentation.

A.3 Working with the code

If you don't have an IDE preference we would recommend that you use Spring Tools Suite or Eclipse when working with the code. We use the m2eclipe eclipse plugin for maven support. Other IDEs and tools should also work without issue.

Importing into eclipse with m2eclipse

We recommend the m2eclipe eclipse plugin when working with eclipse. If you don't already have m2eclipse installed it is available from the "eclipse marketplace".

Unfortunately m2e does not yet support Maven 3.3, so once the projects are imported into Eclipse you will also need to tell m2eclipse to use the `.settings.xml` file for the projects. If you do not do this you may see many different errors related to the POMs in the projects. Open your Eclipse preferences,

expand the Maven preferences, and select User Settings. In the User Settings field click Browse and navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.

> **Note**
>
> Alternatively you can copy the repository settings from `.settings.xml` into your own `~/.m2/settings.xml`.

Importing into eclipse without m2eclipse

If you prefer not to use m2eclipse you can generate eclipse project metadata using the following command:

```
$ ./mvnw eclipse:eclipse
```

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu. [[contributing] == Contributing

Spring Cloud is released under the non-restrictive Apache 2.0 license, and follows a very standard Github development process, using Github tracker for issues and merging pull requests into master. If you want to contribute even something trivial please do not hesitate, but follow the guidelines below.

A.4 Sign the Contributor License Agreement

Before we accept a non-trivial patch or pull request we will need you to sign the contributor's agreement. Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. Active contributors might be asked to join the core team, and given the ability to merge pull requests.

A.5 Code Conventions and Housekeeping

None of these is essential for a pull request, but they will all help. They can also be added after the original pull request but before a merge.

- Use the Spring Framework code format conventions. If you use Eclipse you can import formatter settings using the `eclipse-code-formatter.xml` file from the Spring Cloud Build project. If using IntelliJ, you can use the Eclipse Code Formatter Plugin to import the same file.
- Make sure all new `.java` files to have a simple Javadoc class comment with at least an `@author` tag identifying you, and preferably at least a paragraph on what the class is for.
- Add the ASF license header comment to all new `.java` files (copy from existing files in the project)
- Add yourself as an `@author` to the .java files that you modify substantially (more than cosmetic changes).
- Add some Javadocs and, if you change the namespace, some XSD doc elements.
- A few unit tests would help a lot as well — someone has to do it.
- If no-one else is using your branch, please rebase it against the current master (or other target branch in the main project).

- When writing a commit message please follow these conventions, if you are fixing an existing issue please add `Fixes gh-XXXX` at the end of the commit message (where XXXX is the issue number).