# Spring Cloud Stream Reference Guide

Elmhurst.RC3

Sabby Anandan, Marius Bogoevici, Eric Bottard, Mark Fisher, Ilayaperumal Gopinathan, Gunnar Hillert, Mark Pollack, Patrick Peralta, Glenn Renfro, Thomas Risberg, Dave Syer, David Turanski, Janne Valkealahti, Benjamin Klein, Soby Chacko, Vinicius Carvalho, Gary Russell, Oleg Zhurakousky

Copyright © 2013-2017 Pivotal 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

I. Spring Cloud Stream Core ..... 1
1. Quick Start ..... 2
1.1. Step One - Create sample Application using Spring Initilaizer ..... 2
1.2. Step Two - Import project into the IDE ..... 2
1.3. Step Three - Add message handler, build and run ..... 2
2. What's New in 2.0? ..... 5
2.1. New Features and Components ..... 5
Polling Consumer ..... 5
Micrometer support ..... 5
New Actuator Binding controls ..... 5
Configurable RetryTemplate ..... 5
2.2. Notable changes and enhancements ..... 5
Both Actuator and Web dependencies are now optional ..... 5
Content-type negotiation improvenents ..... 6
2.3. Notable Deprecations ..... 6
Java serialization (Java native and Kryo) ..... 6
Deprecated classes and methods ..... 6
3. Introducing Spring Cloud Stream ..... 8
4. Main Concepts ..... 10
4.1. Application Model ..... 10
Fat JAR ..... 10
4.2. The Binder Abstraction ..... 10
4.3. Persistent Publish-Subscribe Support ..... 11
4.4. Consumer Groups ..... 12
4.5. Consumer Types ..... 12
Durability ..... 13
4.6. Partitioning Support ..... 13
5. Programming Model ..... 15
5.1. Declaring and Binding Producers and Consumers ..... 15
Triggering Binding Via `@EnableBinding` ..... 15
`@Input` and `@Output` ..... 15
Customizing Channel Names ..... 16
`Source`, `Sink`, and `Processor` ..... 16
Accessing Bound Channels ..... 17
Injecting the Bound Interfaces ..... 17
Injecting Channels Directly ..... 17
Producing and Consuming Messages ..... 18
Native Spring Integration Support ..... 18
Spring Integration Error Channel Support ..... 19
Message Channel Binders and Error Channels ..... 19
Using @StreamListener for Automatic Content Type Handling ..... 19
Using @StreamListener for dispatching messages to multiple methods ..... 20
Using Polled Consumers ..... 21
Reactive Programming Support ..... 22
Reactor-based handlers ..... 23
Reactive Sources ..... 24
Aggregation ..... 25

Configuring aggregate application ... 27
Configuring binding service properties for non self contained aggregate application ... 27
6. Binders ... 29
6.1. Producers and Consumers ... 29
6.2. Binder SPI ... 29
6.3. Binder Detection ... 30
Classpath Detection ... 30
6.4. Multiple Binders on the Classpath ... 30
6.5. Connecting to Multiple Systems ... 31
6.6. Binding visualization and control ... 31
6.7. Binder configuration properties ... 32
7. Configuration Options ... 34
7.1. Spring Cloud Stream Properties ... 34
7.2. Binding Properties ... 35
Properties for Use of Spring Cloud Stream ... 35
Consumer properties ... 35
Producer Properties ... 36
7.3. Using dynamically bound destinations ... 38
8. Content Type negotiation ... 41
8.1. Introduction ... 41
8.2. Mechanics ... 41
Content type vs. argument type ... 42
Message Converters ... 43
8.3. Provided MessageConverters ... 43
8.4. User defined Message Converters ... 44
9. Schema evolution support ... 46
9.1. Apache Avro Message Converters ... 46
9.2. Converters with schema support ... 46
9.3. Schema Registry Support ... 47
9.4. Schema Registry Server ... 47
Schema Registry Server API ... 47
9.5. Schema Registry Client ... 49
Using Confluent’s Schema Registry ... 50
Schema Registry Client properties ... 50
9.6. Avro Schema Registry Client Message Converters ... 50
Avro Schema Registry Message Converter properties ... 51
9.7. Schema Registration and Resolution ... 51
Schema Registration Process (Serialization) ... 52
Schema Resolution Process (Deserialization) ... 52
10. Inter-Application Communication ... 53
10.1. Connecting Multiple Application Instances ... 53
10.2. Instance Index and Instance Count ... 53
10.3. Partitioning ... 53
Configuring Output Bindings for Partitioning ... 53
Configuring Input Bindings for Partitioning ... 55
11. Testing ... 56
11.1. Disabling the test binder autoconfiguration ... 57
12. Health Indicator ... 58
13. Metrics Emitter ... 59

14. Samples ........ 62
14.1. Deploying Stream applications on CloudFoundry ........ 62
II. Binder Implementations ........ 63
15. Apache Kafka Binder ........ 64
15.1. Usage ........ 64
15.2. Apache Kafka Binder Overview ........ 64
15.3. Configuration Options ........ 64
Kafka Binder Properties ........ 64
Kafka Consumer Properties ........ 66
Kafka Producer Properties ........ 69
Usage examples ........ 70
Example: Setting `autoCommitOffset` false and relying on manual acking. ........ 70
Example: security configuration ........ 70
Example: Pausing and Resuming the Consumer ........ 72
15.4. Error Channels ........ 73
15.5. Kafka Metrics ........ 73
15.6. Dead-Letter Topic Processing ........ 73
15.7. Partitioning with the Kafka Binder ........ 75
16. Apache Kafka Streams Binder ........ 77
16.1. Usage ........ 77
16.2. Kafka Streams Binder Overview ........ 77
Streams DSL ........ 77
16.3. Configuration Options ........ 78
Kafka Streams Properties ........ 78
TimeWindow properties: ........ 79
16.4. Multiple Input Bindings ........ 80
Multiple Input Bindings as a Sink ........ 80
Multiple Input Bindings as a Processor ........ 80
16.5. Multiple Output Bindings (aka Branching) ........ 81
16.6. Message Conversion ........ 82
Outbound serialization ........ 82
Inbound Deserialization ........ 83
16.7. Error Handling ........ 84
Handling Deserialization Exceptions ........ 84
Handling Non-Deserialization Exceptions ........ 85
16.8. Interactive Queries ........ 86
17. RabbitMQ Binder ........ 87
17.1. Usage ........ 87
17.2. RabbitMQ Binder Overview ........ 87
17.3. Configuration Options ........ 88
RabbitMQ Binder Properties ........ 88
RabbitMQ Consumer Properties ........ 89
Rabbit Producer Properties ........ 93
17.4. Retry With the RabbitMQ Binder ........ 97
Overview ........ 97
Putting it All Together ........ 98
17.5. Error Channels ........ 99
17.6. Dead-Letter Queue Processing ........ 99
Non-Partitioned Destinations ........ 100

Partitioned Destinations .................... 101
republishToDlq=false .................... 101
republishToDlq=true .................... 102
17.7. Partitioning with the RabbitMQ Binder .................... 103
III. Appendices .................... 106
A. Building .................... 107
A.1. Basic Compile and Test .................... 107
A.2. Documentation .................... 107
A.3. Working with the code .................... 107
Importing into eclipse with m2eclipse .................... 107
Importing into eclipse without m2eclipse .................... 108
A.4. Sign the Contributor License Agreement .................... 108
A.5. Code Conventions and Housekeeping .................... 108

# Part I. Spring Cloud Stream Core

# 1. Quick Start

You can try Spring Cloud Stream in less then 5 min even before you jump into any details and the following *three-step guide* will help.

We'll create a simple Spring Cloud Stream application which receives messages coming from the messaging middleware of your choice (more on this later) and logs received messages to the console. We'll call it *LoggingConsumer*. While not very practical it will certainly provide a good introduction to some of the main concepts and abstractions, making it easier to digest the rest of this user guide.

So let's get started. . .

## 1.1 Step One - Create sample Application using Spring Initilaizer

Visit the [Spring Initializr](). This is where we'll generate our *LoggingConsumer* application.

In the *Dependencies* start typing 'stream' and *Cloud Stream* option should pop up. Select it. Now start typing either 'kafka' or 'rabbit'. Basically this is where you are choosing what messaging midleware this application will be bound to. Choose the one you have already installed and/or feel more comfortable with installing/running. Also, as you can see from the Initilaizer screen there are few other options you can choose. For example, you can choose Gradle as your build tool instead of the default Maven. With the *Dependencies* selected the only other thing you have to identify is the application name - *logging-consumer*. Your configuration screeen should now contain the following:

```
Dependencies: Cloud Stream, RabbitMQ (or Kafka)
Group: com.example - default
Artifact: logging-consumer
Spring Boot Version: 2.0.0 (or above) - default
```

Click on *Generate Project* button. This will donwload the zipped version of the generated project to your hard drive. Unzip it and you're ready for Step Two.

## 1.2 Step Two - Import project into the IDE

Here you simply import the project into your IDE of choice. Please keep in mind that dependening on the IDE you may need to follow a specific import procedures. For example depending on how the project was generated (Maven or Gradle) you may need to follow specific import procedure (e.g., in Eclipse/STS: `File # Import # Maven # Existing Maven Project`).

Ones imported the project must have no errors of any kind and `src/main/java` should also contain `com.example.loggingconsumer.LoggingConsumerApplication`.

Technically at this point you can just run the application's main class since it's already a valid *Spring Boot* application, but it does not do anything, so let's add some code.

## 1.3 Step Three - Add message handler, build and run

Modify the `com.example.loggingconsumer.LoggingConsumerApplication` to look as follows:

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

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

 @StreamListener(Sink.INPUT)
 public void handle(Person person) {
  System.out.println("Received: " + person);
 }

 public static class Person {
  private String name;
  public String getName() {
   return name;
  }
  public void setName(String name) {
   this.name = name;
  }
  public String toString() {
   return this.name;
  }
 }
}
```

As you can see from the above:

- We've enabled `Sink` binding (input-no-output) via `@EnableBinding(Sink.class)`. This will signal to the framework to initiate binding to the messaging middleware where it will auto-create the destination (i.e., queue, topic) which will be bound to `Sink.INPUT` channel.
- We've added handler method to receive incoming Message as type `Person`. What this means is that here youcan already observe one of the core features of the framework where it will attempt to automatically convert incoming message's payload to type `Person`.

This is it, we now have a fully functional Spring Cloud Stream application that does something. From here for simplicity we'll assume RabbitMQ was selected in *step one*. Assuming you have RabbitMQ installed and running, start the application by simply running its `main` method.

You should see following output:

```
--- [ main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound:
 input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg, bound to: input
--- [ main] o.s.a.r.c.CachingConnectionFactory       : Attempting to connect to: [localhost:5672]
--- [ main] o.s.a.r.c.CachingConnectionFactory       : Created new connection:
 rabbitConnectionFactory#2a3a299:0/SimpleConnection@66c83fc8. . .
. . .
--- [ main] o.s.i.a.i.AmqpInboundChannelAdapter      : started
 inbound.input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg
. . .
--- [ main] c.e.l.LoggingConsumerApplication         : Started LoggingConsumerApplication in 2.531
 seconds (JVM running for 2.897)
```

Go to RabbitMQ management console or any other RabbitMQ client and simply send message to `input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg` (NOTE: the `anonymous.CbMIwdkJSBO1ZoPDOtHtCg` part represents the group name and is generated and will be different in your environment. For something more predictable you can use explicit group name via `spring.cloud.stream.bindings.input.group=hello`).

The contents of the message should be JSON representation of `Person` class, so let's send this:

```
{"name":"Turd Ferguson"}
```

And in your console you should see:

```
Received: Turd Ferguson
```

You can also build/package your application into a boot jar (i.e., `./mvnw clean install`) and run the built JAR using `java -jar` command.

That is all!

# 2. What’s New in 2.0?

Spring Cloud Stream introduces quite a number of new features, enhancements and changes. The following sections outline most notable ones.

## 2.1 New Features and Components

### Polling Consumer

Introduction of *polled consumers*, where the application can control message processing rates. Please refer to the appropriate section for more details. You can also read this blog for more details spring.io/blog/2018/02/27/spring-cloud-stream-2-0-polled-consumers

### Micrometer support

Metrics has been switched to use Micrometer. `MeterRegistry` is also provided as a bean so custom application can autowire it to capture custom metrics. Please refer to the appropriate section for more details

### New Actuator Binding controls

There are now new new Actuator binding controls to both visualize as well as control Bindings lifecycle. For more details please visit Section 6.6, “Binding visualization and control”

### Configurable RetryTemplate

Aside from providing properties to configure `RetryTemplate` we now allow you to provide your own effectively overriding the one provided by the framework. Simply configure it as a `@Bean` in your application.

## 2.2 Notable changes and enhancements

### Both Actuator and Web dependencies are now optional

This helps to slim down the footprint of the deployed application in the event neither of the functionality is required. It also allows one to swicth between the reactive and conventional web paradigms by adding one of the following dependencies manually:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

or

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Actuator dependency can be added as follows:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

### Content-type negotiation improvenents

One of the core themes for 2.0 is improvements (both consistency and performance) around content-type negotiation and message conversion. The following summary outlines notable changes and improvements. Please refer to the appropriate section for more details as well as this blog spring.io/blog/2018/02/26/spring-cloud-stream-2-0-content-type-negotiation-and-transformation.

- All message conversion is now handled **only** by `MessageConverters`.
- Introduction of `@StreamMessageConverter` annotation to provide custom `MessageConverters`.
- Introduction of the default *Content Type* as `application/json` which needs to be taken into consideration when migrating 1.3 application and/or operating in the mixed mode (i.e., 1.3 producer → 2.0 consumer).
- Messages with textual payloads and *contentType* `text/…` or `…/json` are no longer converted to `Message<String>` for cases where argument type of the provided `MessageHandler` can not be determnied (i.e., `public void handle(Message<?> message)` or `public void handle(Object payload)`). Further more, a strong argument type may not be enough to properly convert messages, so `contentType` header is may be used as supplement by some `MessageConverters`.

## 2.3 Notable Deprecations

### Java serialization (Java native and Kryo)

- `JavaSerializationMessageConverter` and `KryoMessageConverter`. While these two converters remain for now, they will be moved out of the core packages and support in the future. The main reason for this deprecation is to signal the issue *type-based language-specific* serialization couuld cause in the distributed environments, where Producers and Consumers may not only depend on different JVM versions or have different versions of supporting libraries (i.e., Kryo), but to also draw the attention to the fact that Consumers and Producers may and in a lot of cases are non-Java based.

### Deprecated classes and methods

Following is a quick summary of notable deprecations. See corresponding javadocs fort more details.

- `SharedChannelRegistry` in favor of `SharedBindingTargetRegistry`.
- `Bindings` - beans qualified by it are already uniquely identified by their type. For example, provided `Source`, `Processor` or custom bindings:

```
public interface Foo {
 String OUTPUT = "fooOutput";

 @Output(Foo.OUTPUT)
 MessageChannel output();
}
```

- `HeaderMode.raw`. Use `none`, `headers` or `embeddedHeaders`
- `ProducerProperties.partitionKeyExtractorClass` in favor of `partitionKeyExtractorName` and `ProducerProperties.partitionSelectorClass` in favor of `partitionSelectorName`. This is to ensure that both components are Spring configured/managed and referenced in Spring-friendly way.

- `BinderAwareRouterBeanPostProcessor` - while the component exists it is no longer a Bean Post Processor and will be renamed in the future.
- `BinderProperties.setEnvironment(Properties environment)` in favor of `BinderProperties.setEnvironment(Map<String, Object> environment)`.

This section goes into more detail about how you can work with Spring Cloud Stream. It covers topics such as creating and running stream applications.

# 3. Introducing Spring Cloud Stream

Spring Cloud Stream is a framework for building message-driven microservice applications. Spring Cloud Stream builds upon Spring Boot to create standalone, production-grade Spring applications, and uses Spring Integration to provide connectivity to message brokers. It provides opinionated configuration of middleware from several vendors, introducing the concepts of persistent publish-subscribe semantics, consumer groups, and partitions.

You can add the `@EnableBinding` annotation to your application to get immediate connectivity to a message broker, and you can add `@StreamListener` to a method to cause it to receive events for stream processing. The following is a simple sink application which receives external messages.

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

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

  @StreamListener(Sink.INPUT)
  public void processVote(Vote vote) {
      votingService.recordVote(vote);
  }
}
```

The `@EnableBinding` annotation takes one or more interfaces as parameters (in this case, the parameter is a single `Sink` interface). An interface declares input and/or output channels. Spring Cloud Stream provides the interfaces `Source`, `Sink`, and `Processor`; you can also define your own interfaces.

The following is the definition of the `Sink` interface:

```
public interface Sink {
  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

The `@Input` annotation identifies an *input channel*, through which received messages enter the application; the `@Output` annotation identifies an *output channel*, through which published messages leave the application. The `@Input` and `@Output` annotations can take a channel name as a parameter; if a name is not provided, the name of the annotated method will be used.

Spring Cloud Stream will create an implementation of the interface for you. You can use this in the application by autowiring it, as in the following example of a test case.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = VoteRecordingSinkApplication.class)
@WebAppConfiguration
@DirtiesContext
public class StreamApplicationTests {

  @Autowired
  private Sink sink;

  @Test
  public void contextLoads() {
    assertNotNull(this.sink.input());
  }
```

```
}
```

# 4. Main Concepts

Spring Cloud Stream provides a number of abstractions and primitives that simplify the writing of message-driven microservice applications. This section gives an overview of the following:

- Spring Cloud Stream's application model
- The Binder abstraction
- Persistent publish-subscribe support
- Consumer group support
- Partitioning support
- A pluggable Binder API

## 4.1 Application Model

A Spring Cloud Stream application consists of a middleware-neutral core. The application communicates with the outside world through input and output *channels* injected into it by Spring Cloud Stream. Channels are connected to external brokers through middleware-specific Binder implementations.

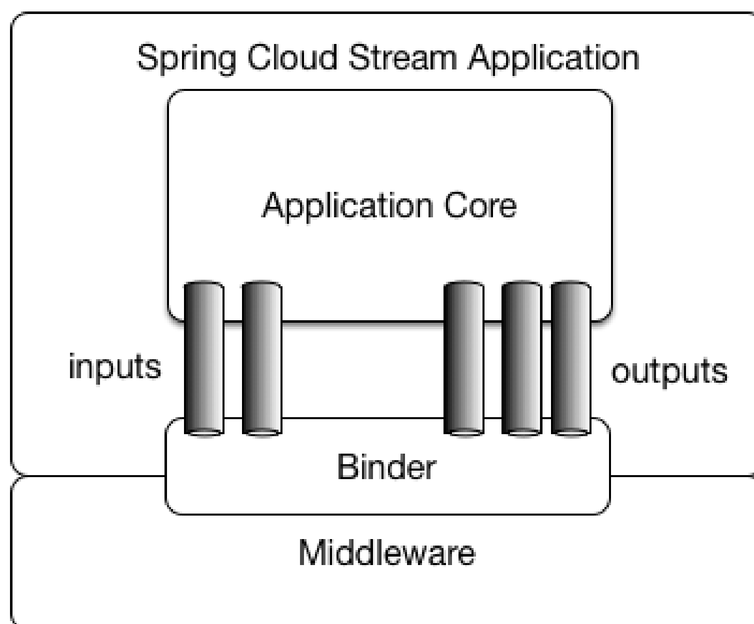


*Figure 4.1. Spring Cloud Stream Application*

### Fat JAR

Spring Cloud Stream applications can be run in standalone mode from your IDE for testing. To run a Spring Cloud Stream application in production, you can create an executable (or "fat") JAR by using the standard Spring Boot tooling provided for Maven or Gradle.

## 4.2 The Binder Abstraction

Spring Cloud Stream provides Binder implementations for Kafka and Rabbit MQ. Spring Cloud Stream also includes a TestSupportBinder, which leaves a channel unmodified so that tests can interact with

channels directly and reliably assert on what is received. You can use the extensible API to write your own Binder.

Spring Cloud Stream uses Spring Boot for configuration, and the Binder abstraction makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the destinations (e.g., the Kafka topics or RabbitMQ exchanges) to which channels connect. Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and `application.yml` or `application.properties` files). In the sink example from the Chapter 3, *Introducing Spring Cloud Stream* section, setting the application property `spring.cloud.stream.bindings.input.destination` to `raw-sensor-data` will cause it to read from the `raw-sensor-data` Kafka topic, or from a queue bound to the `raw-sensor-data` RabbitMQ exchange.

Spring Cloud Stream automatically detects and uses a binder found on the classpath. You can easily use different types of middleware with the same code: just include a different binder at build time. For more complex use cases, you can also package multiple binders with your application and have it choose the binder, and even whether to use different binders for different channels, at runtime.

## 4.3 Persistent Publish-Subscribe Support

Communication between applications follows a publish-subscribe model, where data is broadcast through shared topics. This can be seen in the following figure, which shows a typical deployment for a set of interacting Spring Cloud Stream applications.

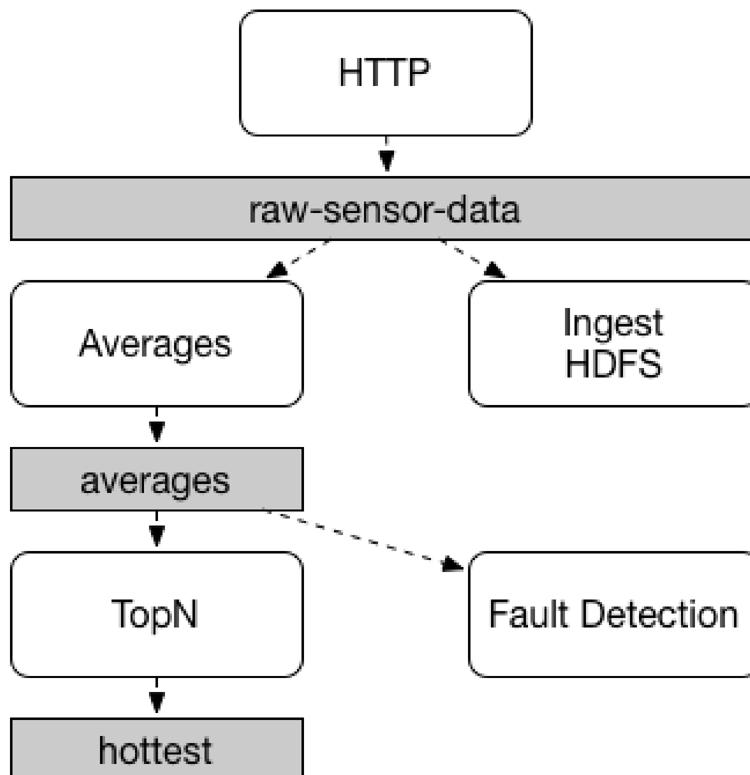


*Figure 4.2. Spring Cloud Stream Publish-Subscribe*

Data reported by sensors to an HTTP endpoint is sent to a common destination named `raw-sensor-data`. From the destination, it is independently processed by a microservice application that computes

time-windowed averages and by another microservice application that ingests the raw data into HDFS. In order to process the data, both applications declare the topic as their input at runtime.

The publish-subscribe communication model reduces the complexity of both the producer and the consumer, and allows new applications to be added to the topology without disruption of the existing flow. For example, downstream from the average-calculating application, you can add an application that calculates the highest temperature values for display and monitoring. You can then add another application that interprets the same flow of averages for fault detection. Doing all communication through shared topics rather than point-to-point queues reduces coupling between microservices.

While the concept of publish-subscribe messaging is not new, Spring Cloud Stream takes the extra step of making it an opinionated choice for its application model. By using native middleware support, Spring Cloud Stream also simplifies use of the publish-subscribe model across different platforms.

## 4.4 Consumer Groups

While the publish-subscribe model makes it easy to connect applications through shared topics, the ability to scale up by creating multiple instances of a given application is equally important. When doing this, different instances of an application are placed in a competing consumer relationship, where only one of the instances is expected to handle a given message.

Spring Cloud Stream models this behavior through the concept of a *consumer group*. (Spring Cloud Stream consumer groups are similar to and inspired by Kafka consumer groups.) Each consumer binding can use the `spring.cloud.stream.bindings.<channelName>.group` property to specify a group name. For the consumers shown in the following figure, this property would be set as `spring.cloud.stream.bindings.<channelName>.group=hdfsWrite` or `spring.cloud.stream.bindings.<channelName>.group=average`.

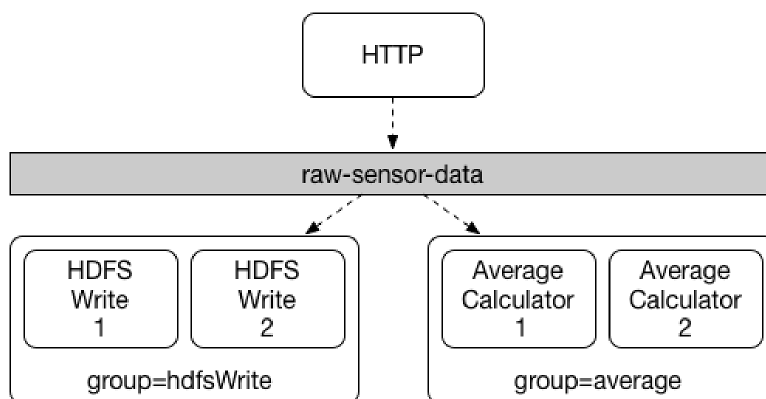


*Figure 4.3. Spring Cloud Stream Consumer Groups*

All groups which subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups.

## 4.5 Consumer Types

Two types of consumer are supported:

- Message-driven (sometimes referred to as Asynchronous)

- Polled (sometimes referred to as Synchronous)

Prior to *version 2.0*, only asynchronous consumers were supported, where a message is delivered as soon as it is available (and there is a thread available to process it).

You might want to use a synchronous consumer when you wish to control the rate at which messages are processed.

### Durability

Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are *durable*. That is, a binder implementation ensures that group subscriptions are persistent, and ones at least one subscription for a group has been created, the group will receive messages, even if they are sent while all applications in the group are stopped.

**Note**

Anonymous subscriptions are non-durable by nature. For some binder implementations (e.g., RabbitMQ), it is possible to have non-durable group subscriptions.

In general, it is preferable to always specify a consumer group when binding an application to a given destination. When scaling up a Spring Cloud Stream application, you must specify a consumer group for each of its input bindings. This prevents the application's instances from receiving duplicate messages (unless that behavior is desired, which is unusual).

## 4.6 Partitioning Support

Spring Cloud Stream provides support for *partitioning* data between multiple instances of a given application. In a partitioned scenario, the physical communication medium (e.g., the broker topic) is viewed as being structured into multiple partitions. One or more producer application instances send data to multiple consumer application instances and ensure that data identified by common characteristics are processed by the same consumer instance.

Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion. Partitioning can thus be used whether the broker itself is naturally partitioned (e.g., Kafka) or not (e.g., RabbitMQ).

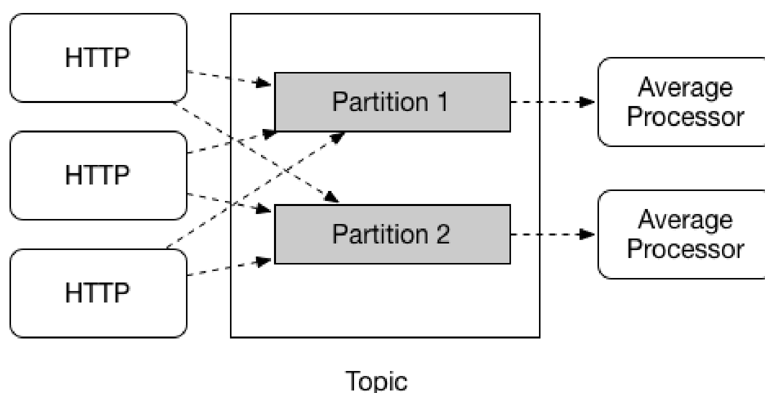


*Figure 4.4. Spring Cloud Stream Partitioning*

Partitioning is a critical concept in stateful processing, where it is critical, for either performance or consistency reasons, to ensure that all related data is processed together. For example, in the time-

windowed average calculation example, it is important that all measurements from any given sensor are processed by the same application instance.

**Note**

To set up a partitioned processing scenario, you must configure both the data-producing and the data-consuming ends.

# 5. Programming Model

This section describes Spring Cloud Stream's programming model. Spring Cloud Stream provides a number of predefined annotations for declaring bound input and output channels as well as how to listen to channels.

## 5.1 Declaring and Binding Producers and Consumers

### Triggering Binding Via `@EnableBinding`

You can turn a Spring application into a Spring Cloud Stream application by applying the `@EnableBinding` annotation to one of the application's configuration classes. The `@EnableBinding` annotation itself is meta-annotated with `@Configuration` and triggers the configuration of Spring Cloud Stream infrastructure:

```
...
@Import(...)
@Configuration
@EnableIntegration
public @interface EnableBinding {
    ...
    Class<?>[] value() default {};
}
```

The `@EnableBinding` annotation can take as parameters one or more interface classes that contain methods which represent bindable components (typically message channels).

**Note**

The `@EnableBinding` annotation is only required on your `Configuration` classes, you can provide as many binding interfaces as you need, for instance: `@EnableBinding(value={Orders.class, Payment.class}`. Where both `Order` and `Payment` interfaces would declare `@Input` and `@Output` channels.

### `@Input` and `@Output`

A Spring Cloud Stream application can have an arbitrary number of input and output channels defined in an interface as `@Input` and `@Output` methods:

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

Using this interface as a parameter to `@EnableBinding` will trigger the creation of three bound channels named `orders`, `hotDrinks`, and `coldDrinks`, respectively.

```
@EnableBinding(Barista.class)
public class CafeConfiguration {

   ...
}
```

**Note**

In Spring Cloud Stream, the bindable `MessageChannel` components are the Spring Messaging `MessageChannel` (for outbound) and its extension `SubscribableChannel` (for inbound). Using the same mechanism, other bindable components can be supported. `KStream` support in Spring Cloud Stream Kafka binder is one such example where KStream is used as inbound/outbound `bindable` components. Also, as discussed below, a `PollableMessageSource` can be bound to an inbound destination. In this documentation, we will continue to refer to MessageChannels as the `bindable` components.

Starting with *version 2.0*, you can now bind a pollable consumer as follows:

```
public interface PolledBarista {

    @Input
    PollableMessageSource orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

In this case, an implementation of `PollableMessageSource` is bound to the `orders` "channel".

#### Customizing Channel Names

Using the `@Input` and `@Output` annotations, you can specify a customized channel name for the channel, as shown in the following example:

```
public interface Barista {
    ...
    @Input("inboundOrders")
    SubscribableChannel orders();
}
```

In this example, the created bound channel will be named `inboundOrders`.

#### `Source`, `Sink`, and `Processor`

For easy addressing of the most common use cases, which involve either an input channel, an output channel, or both, Spring Cloud Stream provides three predefined interfaces out of the box.

`Source` can be used for an application which has a single outbound channel.

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();

}
```

`Sink` can be used for an application which has a single inbound channel.

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
```

```
    SubscribableChannel input();

}
```

`Processor` can be used for an application which has both an inbound channel and an outbound channel.

```
public interface Processor extends Source, Sink {
}
```

Spring Cloud Stream provides no special handling for any of these interfaces; they are only provided out of the box.

## Accessing Bound Channels

### Injecting the Bound Interfaces

For each bound interface, Spring Cloud Stream will generate a bean that implements the interface. Invoking a `@Input`-annotated or `@Output`-annotated method of one of these beans will return the relevant bound channel.

The bean in the following example sends a message on the output channel when its `hello` method is invoked. It invokes `output()` on the injected `Source` bean to retrieve the target channel.

```
@Component
public class SendingBean {

    private Source source;

    @Autowired
    public SendingBean(Source source) {
        this.source = source;
    }

    public void sayHello(String name) {
         source.output().send(MessageBuilder.withPayload(name).build());
    }
}
```

### Injecting Channels Directly

Bound channels can be also injected directly:

```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {
         output.send(MessageBuilder.withPayload(name).build());
    }
}
```

If the name of the channel is customized on the declaring annotation, that name should be used instead of the method name. Given the following declaration:

```
public interface CustomSource {
    ...
```

```
    @Output("customOutput")
    MessageChannel output();
}
```

The channel will be injected as shown in the following example:

```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(@Qualifier("customOutput") MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {
         this.output.send(MessageBuilder.withPayload(name).build());
    }
}
```

## Producing and Consuming Messages

You can write a Spring Cloud Stream application using either Spring Integration annotations or Spring Cloud Stream's `@StreamListener` annotation. The `@StreamListener` annotation is modeled after other Spring Messaging annotations (such as `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.) but adds content type management and type coercion features.

### Native Spring Integration Support

Because Spring Cloud Stream is based on Spring Integration, Stream completely inherits Integration's foundation and infrastructure as well as the component itself. For example, you can attach the output channel of a `Source` to a `MessageSource`:

```
@EnableBinding(Source.class)
public class TimerSource {

  @Value("${format}")
  private String format;

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "${fixedDelay}",
 maxMessagesPerPoll = "1"))
  public MessageSource<String> timerMessageSource() {
    return () -> new GenericMessage<>(new SimpleDateFormat(format).format(new Date()));
  }
}
```

Or you can use a processor's channels in a transformer:

```
@EnableBinding(Processor.class)
public class TransformProcessor {
  @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
  public Object transform(String message) {
    return message.toUpperCase();
  }
}
```

**Note**

It's important to understand that when you consume from the same binding using `@StreamListener` a pubsub model is used, where each method annotated with `@StreamListener` receives it's own copy of the message, each one has its own consumer

group. However, if you share a bindable channel as an input for `@Aggregator`, `@Transformer` or `@ServiceActivator`, those will consume in a competing model, no individual consumer group is created for each subscription.

#### Spring Integration Error Channel Support

Spring Cloud Stream supports publishing error messages received by the Spring Integration global error channel. Error messages sent to the `errorChannel` can be published to a specific destination at the broker by configuring a binding for the outbound target named `error`. For example, to publish error messages to a broker destination named "myErrors", provide the following property: `spring.cloud.stream.bindings.error.destination=myErrors`.

#### Message Channel Binders and Error Channels

Starting with *version 1.3*, some `MessageChannel` - based binders publish errors to a discrete error channel for each destination. In addition, these error channels are bridged to the global Spring Integration `errorChannel` mentioned above. You can therefore consume errors for specific destinations and/or for all destinations, using a standard Spring Integration flow (`IntegrationFlow`, `@ServiceActivator`, etc.).

On the consumer side, the listener thread catches any exceptions and forwards an `ErrorMessage` to the destination's error channel. The payload of the message is a `MessagingException` with the normal `failedMessage` and `cause` properties. Usually, the raw data received from the broker is included in a header. For binders that support (and are configured with) a dead letter destination; a `MessagePublishingErrorHandler` is subscribed to the channel, and the raw data is forwarded to the dead letter destination.

On the producer side; for binders that support some kind of async result after publishing messages (e.g. RabbitMQ, Kafka), you can enable an error channel by setting the …`producer.errorChannelEnabled` to `true`. The payload of the `ErrorMessage` depends on the binder implementation but will be a `MessagingException` with the normal `failedMessage` property, as well as additional properties about the failure. Refer to the binder documentation for complete details.

#### Using @StreamListener for Automatic Content Type Handling

Complementary to its Spring Integration support, Spring Cloud Stream provides its own `@StreamListener` annotation, modeled after other Spring Messaging annotations (e.g. `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.). The `@StreamListener` annotation provides a simpler model for handling inbound messages, especially when dealing with use cases that involve content type management and type coercion.

Spring Cloud Stream provides an extensible `MessageConverter` mechanism for handling data conversion by bound channels and for, in this case, dispatching to methods annotated with `@StreamListener`. The following is an example of an application which processes external `Vote` events:

```
@EnableBinding(Sink.class)
public class VoteHandler {

  @Autowired
  VotingService votingService;

  @StreamListener(Sink.INPUT)
  public void handle(Vote vote) {
    votingService.record(vote);
  }
}
```

The distinction between `@StreamListener` and a Spring Integration `@ServiceActivator` is seen when considering an inbound `Message` that has a `String` payload and a `contentType` header of `application/json`. In the case of `@StreamListener`, the `MessageConverter` mechanism will use the `contentType` header to parse the `String` payload into a `Vote` object.

As with other Spring Messaging methods, method arguments can be annotated with `@Payload`, `@Headers` and `@Header`.

**Note**

For methods which return data, you must use the `@SendTo` annotation to specify the output binding destination for data returned by the method:

```
@EnableBinding(Processor.class)
public class TransformProcessor {

  @Autowired
  VotingService votingService;

  @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
  public VoteResult handle(Vote vote) {
    return votingService.record(vote);
  }
}
```

### Using @StreamListener for dispatching messages to multiple methods

Since version 1.2, Spring Cloud Stream supports dispatching messages to multiple `@StreamListener` methods registered on an input channel, based on a condition.

In order to be eligible to support conditional dispatching, a method must satisfy the follow conditions:

- it must not return a value
- it must be an individual message handling method (reactive API methods are not supported)

The condition is specified via a SpEL expression in the `condition` attribute of the annotation and is evaluated for each message. All the handlers that match the condition will be invoked in the same thread and no assumption must be made about the order in which the invocations take place.

An example of using `@StreamListener` with dispatching conditions can be seen below. In this example, all the messages bearing a header `type` with the value `foo` will be dispatched to the `receiveFoo` method, and all the messages bearing a header `type` with the value `bar` will be dispatched to the `receiveBar` method.

```
@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class TestPojoWithAnnotatedArguments {

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='foo'")
    public void receiveFoo(@Payload FooPojo fooPojo) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bar'")
    public void receiveBar(@Payload BarPojo barPojo) {
       // handle the message
    }
}
```

**Note**

Dispatching via `@StreamListener` conditions is only supported for handlers of individual messages, and not for reactive programming support (described below).

### Using Polled Consumers

When using polled consumers, you poll the `PollableMessageSource` on demand. For example, given…

```
public interface PolledConsumer {

    @Input
    PollableMessageSource destIn();

    @Output
    MessageChannel destOut();

}
```

…you might use that consumer as follows:

```
@Bean
public ApplicationRunner poller(PollableMessageSource destIn, MessageChannel destOut) {
    return args -> {
        while (someCondition()) {
            try {
                if (!destIn.poll(m -> {
                    String newPayload = ((String) m.getPayload()).toUpperCase();
                    destOut.send(new GenericMessage<>(newPayload));
                })) {
                    Thread.sleep(1000);
                }
            }
            catch (Exception e) {
                // handle failure (throw an exception to reject the message);
            }
        }
    };
}
```

The `PollableMessageSource.poll()` method takes a `MessageHandler` argument (often a lambda expression as shown here). It returns `true` if the message was received and successfully processed.

As with message-driven consumers, if the `MessageHandler` throws an exception, messages are published to error channels as discussed in the section called “Message Channel Binders and Error Channels”.

Normally, the `poll()` method will acknowledge the message when the `MessageHandler` exits. If the method exits abnormally, the message is rejected (not requeued). You can override that behavior, by taking responsibility for the acknowledgment, as follows:

```
@Bean
public ApplicationRunner poller(PollableMessageSource dest1In, MessageChannel dest2Out) {
    return args -> {
        while (someCondition()) {
            if (!dest1In.poll(m -> {
                StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).noAutoAck();
                // e.g. hand off to another thread which can perform the ack
                // or acknowledge(Status.REQUEUE)

            })) {
                Thread.sleep(1000);
```

```
            }
        }
    };
}
```

**Important**

You must ack (or nack) the message at some point, to avoid resource leaks.

**Important**

Some messaging systems (such as Apache Kafka) maintain a simple offset in a log, if a delivery fails and is requeued with `StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUE` any later successfully ack'd messages will be redelivered.

There is also an overloaded `poll` method:

```
poll(MessageHandler handler, ParameterizedTypeReference<?> type)
```

The `type` is a conversion hint allowing the incoming message payload to be converted:

```
boolean result = pollableSource.poll(received -> {
   Map<String, Foo> payload = (Map<String, Foo>) received.getPayload();
            ...

  }, new ParameterizedTypeReference<Map<String, Foo>>() {});
```

## Reactive Programming Support

Spring Cloud Stream also supports the use of reactive APIs where incoming and outgoing data is handled as continuous data flows. Support for reactive APIs is available via the `spring-cloud-stream-reactive`, which needs to be added explicitly to your project.

The programming model with reactive APIs is declarative, where instead of specifying how each individual message should be handled, you can use operators that describe functional transformations from inbound to outbound data flows.

Spring Cloud Stream supports the following reactive APIs:

- Reactor

In the future, it is intended to support a more generic model based on Reactive Streams.

The reactive programming model is also using the `@StreamListener` annotation for setting up reactive handlers. The differences are that:

- the `@StreamListener` annotation must not specify an input or output, as they are provided as arguments and return values from the method;
- the arguments of the method must be annotated with `@Input` and `@Output` indicating which input or output will the incoming and respectively outgoing data flows connect to;
- the return value of the method, if any, will be annotated with `@Output`, indicating the input where data shall be sent.

**Note**

Reactive programming support requires Java 1.8.

**Note**

As of Spring Cloud Stream 1.1.1 and later (starting with release train Brooklyn.SR2), reactive programming support requires the use of Reactor 3.0.4.RELEASE and higher. Earlier Reactor versions (including 3.0.1.RELEASE, 3.0.2.RELEASE and 3.0.3.RELEASE) are not supported. `spring-cloud-stream-reactive` will transitively retrieve the proper version, but it is possible for the project structure to manage the version of the `io.projectreactor:reactor-core` to an earlier release, especially when using Maven. This is the case for projects generated via Spring Initializr with Spring Boot 1.x, which will override the Reactor version to `2.0.8.RELEASE`. In such cases you must ensure that the proper version of the artifact is released. This can be simply achieved by adding a direct dependency on `io.projectreactor:reactor-core` with a version of `3.0.4.RELEASE` or later to your project.

**Note**

The use of term `reactive` is currently referring to the reactive APIs being used and not to the execution model being reactive (i.e. the bound endpoints are still using a 'push' rather than 'pull' model). While some backpressure support is provided by the use of Reactor, we do intend on the long run to support entirely reactive pipelines by the use of native reactive clients for the connected middleware.

#### Reactor-based handlers

A Reactor based handler can have the following argument types:

- For arguments annotated with `@Input`, it supports the Reactor type `Flux`. The parameterization of the inbound Flux follows the same rules as in the case of individual message handling: it can be the entire `Message`, a POJO which can be the `Message` payload, or a POJO which is the result of a transformation based on the `Message` content-type header. Multiple inputs are provided;
- For arguments annotated with `Output`, it supports the type `FluxSender` which connects a `Flux` produced by the method with an output. Generally speaking, specifying outputs as arguments is only recommended when the method can have multiple outputs;

A Reactor based handler supports a return type of `Flux`, case in which it must be annotated with `@Output`. We recommend using the return value of the method when a single output flux is available.

Here is an example of a simple Reactor-based Processor.

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Flux<String> receive(@Input(Processor.INPUT) Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```

The same processor using output arguments looks like this:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {
```

```
  @StreamListener
  public void receive(@Input(Processor.INPUT) Flux<String> input,
     @Output(Processor.OUTPUT) FluxSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```

### Reactive Sources

Spring Cloud Stream reactive support also provides the ability for creating reactive sources through the StreamEmitter annotation. Using StreamEmitter annotation, a regular source may be converted to a reactive one. StreamEmitter is a method level annotation that marks a method to be an emitter to outputs declared via EnableBinding. It is not allowed to use the Input annotation along with StreamEmitter, as the methods marked with this annotation are not listening from any input, rather generating to an output. Following the same programming model used in StreamListener, StreamEmitter also allows flexible ways of using the Output annotation depending on whether the method has any arguments, return type etc.

Here are some examples of using StreamEmitter in various styles.

The following example will emit the "Hello World" message every millisecond and publish to a Flux. In this case, the resulting messages in Flux will be sent to the output channel of the Source.

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  public Flux<String> emit() {
    return Flux.intervalMillis(1)
            .map(l -> "Hello World");
  }
}
```

Following is another flavor of the same sample as above. Instead of returning a Flux, this method uses a FluxSender to programmatically send Flux from a source.

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  public void emit(FluxSender output) {
    output.send(Flux.intervalMillis(1)
            .map(l -> "Hello World"));
  }
}
```

Following is exactly same as the above snippet in functionality and style. However, instead of using an explicit Output annotation at the method level, it is used as the method parameter level.

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  public void emit(@Output(Source.OUTPUT) FluxSender output) {
    output.send(Flux.intervalMillis(1)
            .map(l -> "Hello World"));
  }
}
```

Here is yet another flavor of writing reacting sources using the Reactive Streams Publisher API and the support for it in the Spring Integration Java DSL. The Publisher is still using Reactor Flux under the hood, but from an application perspective, that is transparent to the user and only needs Reactive Streams and Java DSL for Spring Integration.

```
@EnableBinding(Source.class)
@EnableAutoConfiguration
public static class HelloWorldEmitter {

  @StreamEmitter
  @Output(Source.OUTPUT)
  @Bean
  public Publisher<Message<String>> emit() {
    return IntegrationFlows.from(() ->
                new GenericMessage<>("Hello World"),
          e -> e.poller(p -> p.fixedDelay(1)))
          .toReactivePublisher();
  }
}
```

## Aggregation

Spring Cloud Stream provides support for aggregating multiple applications together, connecting their input and output channels directly and avoiding the additional cost of exchanging messages via a broker. As of version 1.0 of Spring Cloud Stream, aggregation is supported only for the following types of applications:

- *sources* - applications with a single output channel named `output`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Source`
- *sinks* - applications with a single input channel named `input`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Sink`
- *processors* - applications with a single input channel named `input` and a single output channel named `output`, typically having a single binding of the type `org.springframework.cloud.stream.messaging.Processor`.

They can be aggregated together by creating a sequence of interconnected applications, in which the output channel of an element in the sequence is connected to the input channel of the next element, if it exists. A sequence can start with either a *source* or a *processor*, it can contain an arbitrary number of *processors* and must end with either a *processor* or a *sink*.

Depending on the nature of the starting and ending element, the sequence may have one or more bindable channels, as follows:

- if the sequence starts with a source and ends with a sink, all communication between the applications is direct and no channels will be bound
- if the sequence starts with a processor, then its input channel will become the `input` channel of the aggregate and will be bound accordingly
- if the sequence ends with a processor, then its output channel will become the `output` channel of the aggregate and will be bound accordingly

Aggregation is performed using the `AggregateApplicationBuilder` utility class, as in the following example. Let's consider a project in which we have source, processor and a sink, which may be defined in the project, or may be contained in one of the project's dependencies.

**Note**

Each component (source, sink or processor) in an aggregate application must be provided in a separate package if the configuration classes use `@SpringBootApplication`. This is required to avoid cross-talk between applications, due to the classpath scanning performed by `@SpringBootApplication` on the configuration classes inside the same package. In the example below, it can be seen that the Source, Processor and Sink application classes are grouped in separate packages. A possible alternative is to provide the source, sink or processor configuration in a separate `@Configuration` class, avoid the use of `@SpringBootApplication`/`@ComponentScan` and use those for aggregation.

```
package com.app.mysink;

// Imports omitted

@SpringBootApplication
@EnableBinding(Sink.class)
public class SinkApplication {

    private static Logger logger = LoggerFactory.getLogger(SinkApplication.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void loggerSink(Object payload) {
        logger.info("Received: " + payload);
    }
}
```

```
package com.app.myprocessor;

// Imports omitted

@SpringBootApplication
@EnableBinding(Processor.class)
public class ProcessorApplication {

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public String loggerSink(String payload) {
        return payload.toUpperCase();
    }
}
```

```
package com.app.mysource;

// Imports omitted

@SpringBootApplication
@EnableBinding(Source.class)
public class SourceApplication {

    @InboundChannelAdapter(value = Source.OUTPUT)
    public String timerMessageSource() {
        return new SimpleDateFormat().format(new Date());
    }
}
```

Each configuration can be used for running a separate component, but in this case they can be aggregated together as follows:

```
package com.app;

// Imports omitted

@SpringBootApplication
public class SampleAggregateApplication {
```

```
    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).args("--fixedDelay=5000")
            .via(ProcessorApplication.class)
            .to(SinkApplication.class).args("--debug=true").run(args);
    }
}
```

The starting component of the sequence is provided as argument to the `from()` method. The ending component of the sequence is provided as argument to the `to()` method. Intermediate processors are provided as argument to the `via()` method. Multiple processors of the same type can be chained together (e.g. for pipelining transformations with different configurations). For each component, the builder can provide runtime arguments for Spring Boot configuration.

### Configuring aggregate application

Spring Cloud Stream supports passing properties for the individual applications inside the aggregate application using 'namespace' as prefix.

The namespace can be set for applications as follows:

```
@SpringBootApplication
public class SampleAggregateApplication {

    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).namespace("source").args("--fixedDelay=5000")
            .via(ProcessorApplication.class).namespace("processor1")
            .to(SinkApplication.class).namespace("sink").args("--debug=true").run(args);
    }
}
```

Ones the 'namespace' is set for the individual applications, the application properties with the `namespace` as prefix can be passed to the aggregate application using any supported property source (commandline, environment properties etc.).

For instance, to override the default `fixedDelay` and `debug` properties of 'source' and 'sink' applications:

```
java -jar target/MyAggregateApplication-0.0.1-SNAPSHOT.jar --source.fixedDelay=10000 --sink.debug=false
```

### Configuring binding service properties for non self contained aggregate application

The non self-contained aggregate application is bound to external broker via either or both the inbound/ outbound components (typically, message channels) of the aggregate application while the applications inside the aggregate application are directly bound. For example: a source application’s output and a processor application’s input are directly bound while the processor’s output channel is bound to an external destination at the broker. When passing the binding service properties for non-self contained aggregate application, it is required to pass the binding service properties to the aggregate application instead of setting them as 'args' to individual child application. For instance,

```
@SpringBootApplication
public class SampleAggregateApplication {

    public static void main(String[] args) {
        new AggregateApplicationBuilder()
            .from(SourceApplication.class).namespace("source").args("--fixedDelay=5000")
            .via(ProcessorApplication.class).namespace("processor1").args("--debug=true").run(args);
    }
}
```

The binding properties like `--spring.cloud.stream.bindings.output.destination=processor-output` need to be specified as one of the external configuration properties (cmdline arg etc.).

# 6. Binders

Spring Cloud Stream provides a Binder abstraction for use in connecting to physical destinations at the external middleware. This section provides information about the main concepts behind the Binder SPI, its main components, and implementation-specific details.

## 6.1 Producers and Consumers

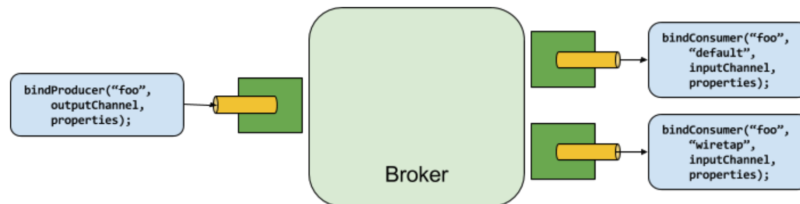


*Figure 6.1. Producers and Consumers*

A *producer* is any component that sends messages to a channel. The channel can be bound to an external message broker via a Binder implementation for that broker. When invoking the `bindProducer()` method, the first parameter is the name of the destination within the broker, the second parameter is the local channel instance to which the producer will send messages, and the third parameter contains properties (such as a partition key expression) to be used within the adapter that is created for that channel.

A *consumer* is any component that receives messages from a channel. As with a producer, the consumer's channel can be bound to an external message broker. When invoking the `bindConsumer()` method, the first parameter is the destination name, and a second parameter provides the name of a logical group of consumers. Each group that is represented by consumer bindings for a given destination receives a copy of each message that a producer sends to that destination (i.e., publish-subscribe semantics). If there are multiple consumer instances bound using the same group name, then messages will be load-balanced across those consumer instances so that each message sent by a producer is consumed by only a single consumer instance within each group (i.e., queueing semantics).

## 6.2 Binder SPI

The Binder SPI consists of a number of interfaces, out-of-the box utility classes and discovery strategies that provide a pluggable mechanism for connecting to external middleware.

The key point of the SPI is the `Binder` interface which is a strategy for connecting inputs and outputs to external middleware.

```
public interface Binder<T, C extends ConsumerProperties, P extends ProducerProperties> {
    Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C consumerProperties);

    Binding<T> bindProducer(String name, T outboundBindTarget, P producerProperties);
}
```

The interface is parameterized, offering a number of extension points:

- input and output bind targets - as of version 1.0, only `MessageChannel` is supported, but this is intended to be used as an extension point in the future;
- extended consumer and producer properties - allowing specific Binder implementations to add supplemental properties which can be supported in a type-safe manner.

A typical binder implementation consists of the following

- a class that implements the `Binder` interface;
- a Spring `@Configuration` class that creates a bean of the type above along with the middleware connection infrastructure;
- a `META-INF/spring.binders` file found on the classpath containing one or more binder definitions, e.g.

```
kafka:\
org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```

## 6.3 Binder Detection

Spring Cloud Stream relies on implementations of the Binder SPI to perform the task of connecting channels to message brokers. Each Binder implementation typically connects to one type of messaging system.

### Classpath Detection

By default, Spring Cloud Stream relies on Spring Boot's auto-configuration to configure the binding process. If a single Binder implementation is found on the classpath, Spring Cloud Stream will use it automatically. For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can simply add the following dependency:

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

For the specific maven coordinates of other binder dependencies, please refer to the documentation of that binder implementation.

## 6.4 Multiple Binders on the Classpath

When multiple binders are present on the classpath, the application must indicate which binder is to be used for each channel binding. Each binder configuration contains a `META-INF/spring.binders`, which is a simple properties file:

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

Similar files exist for the other provided binder implementations (e.g., Kafka), and custom binder implementations are expected to provide them, as well. The key represents an identifying name for the binder implementation, whereas the value is a comma-separated list of configuration classes that each contain one and only one bean definition of type `org.springframework.cloud.stream.binder.Binder`.

Binder selection can either be performed globally, using the `spring.cloud.stream.defaultBinder` property (e.g., `spring.cloud.stream.defaultBinder=rabbit`) or individually, by configuring the binder on each channel binding. For instance, a processor application (that has channels with the names `input` and `output` for read/write respectively) which reads from Kafka and writes to RabbitMQ can specify the following configuration:

```
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

## 6.5 Connecting to Multiple Systems

By default, binders share the application's Spring Boot auto-configuration, so that one instance of each binder found on the classpath will be created. If your application should connect to more than one broker of the same type, you can specify multiple binder configurations, each with different environment settings.

**Note**

Turning on explicit binder configuration will disable the default binder configuration process altogether. If you do this, all binders in use must be included in the configuration. Frameworks that intend to use Spring Cloud Stream transparently may create binder configurations that can be referenced by name, but will not affect the default binder configuration. In order to do so, a binder configuration may have its `defaultCandidate` flag set to false, e.g. `spring.cloud.stream.binders.<configurationName>.defaultCandidate=false`. This denotes a configuration that will exist independently of the default binder configuration process.

For example, this is the typical configuration for a processor application which connects to two RabbitMQ broker instances:

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: foo
          binder: rabbit1
        output:
          destination: bar
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```

## 6.6 Binding visualization and control

Since version 2.0 Spring Cloud Stream supports visualization and control of the Bindings via Actuator endpoints.

**Note**

Given that starting with version 2.0 *actuator* and *web* are optional, one must first add one of the web dependencies as well as the actuator dependency manually.

```
<dependency>
     <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

or

```
<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Actuator dependency can be added as follows:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

You must also enable `bindings` actuator endpoints with the following property `--management.endpoints.web.exposure.include=bindings`.

Once the above prerequisites are satisfied you should see the following in the logs when application is started:

```
: Mapped "{[/actuator/bindings/{name}],methods=[POST]. . .
: Mapped "{[/actuator/bindings],methods=[GET]. . .
: Mapped "{[/actuator/bindings/{name}],methods=[GET]. . .
```

To visualize current bindings simply access the following URL:

```
http://<host>:<port>/actuator/bindings
```

or

```
http://<host>:<port>/actuator/bindings/myBindingName
```

…if you want to visualize a single binding named 'myBindingName'

You can also *stop, start, pause* and *resume* individual binding by posting to the same URL while providing `state` argument as JSON.

For example,

```
curl -d '{"state":"STOPPED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/
bindings/myBindingName
curl -d '{"state":"STARTED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/
bindings/myBindingName
curl -d '{"state":"PAUSED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/
bindings/myBindingName
curl -d '{"state":"RESUMED"}' -H "Content-Type: application/json" -X POST http://<host>:<port>/actuator/
bindings/myBindingName
```

**Note**

*PAUSED* and *RESUMED* are only effective if corresponding binder and its underlyig technology supports it, otherwise you'll see the warning message in the logs. Currently only Kafka binder supports *PAUSED* and *RESUMED* state.

## 6.7 Binder configuration properties

The following properties are available when creating custom binder configurations. They must be prefixed with `spring.cloud.stream.binders.<configurationName>`.

type
: The binder type. It typically references one of the binders found on the classpath, in particular a key in a `META-INF/spring.binders` file.

  By default, it has the same value as the configuration name.

inheritEnvironment
: Whether the configuration will inherit the environment of the application itself.

  Default `true`.

environment
: Root for a set of properties that can be used to customize the environment of the binder. When this is configured, the context in which the binder is being created is not a child of the application context. This allows for complete separation between the binder components and the application components.

  Default `empty`.

defaultCandidate
: Whether the binder configuration is a candidate for being considered a default binder, or can be used only when explicitly referenced. This allows adding binder configurations without interfering with the default processing.

  Default `true`.

# 7. Configuration Options

Spring Cloud Stream supports general configuration options as well as configuration for bindings and binders. Some binders allow additional binding properties to support middleware-specific features.

Configuration options can be provided to Spring Cloud Stream applications via any mechanism supported by Spring Boot. This includes application arguments, environment variables, and YAML or .properties files.

## 7.1 Spring Cloud Stream Properties

spring.cloud.stream.instanceCount
: The number of deployed instances of an application. Must be set for partitioning on the producer side, and on the consumer side if using RabbitMQ and with Kafka if `autoRebalanceEnabled=false`.

  Default: `1`.

spring.cloud.stream.instanceIndex
: The instance index of the application: a number from `0` to `instanceCount`-1. Used for partitioning with RabbitMQ and with Kafka if `autoRebalanceEnabled=false`. Automatically set in Cloud Foundry to match the application's instance index.

spring.cloud.stream.dynamicDestinations
: A list of destinations that can be bound dynamically (for example, in a dynamic routing scenario). If set, only listed destinations can be bound.

  Default: empty (allowing any destination to be bound).

spring.cloud.stream.defaultBinder
: The default binder to use, if multiple binders are configured. See Multiple Binders on the Classpath.

  Default: empty.

spring.cloud.stream.overrideCloudConnectors
: This property is only applicable when the `cloud` profile is active and Spring Cloud Connectors are provided with the application. If the property is false (the default), the binder will detect a suitable bound service (e.g. a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and will use it for creating connections (usually via Spring Cloud Connectors). When set to true, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (e.g. relying on the `spring.rabbitmq.*` properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment when connecting to multiple systems.

  Default: false.

spring.cloud.stream.bindingRetryInterval
: The interval (seconds) between retrying binding creation when, for example, the binder doesn't support late binding and the broker is down (e.g. Apache Kafka). Set to zero to treat such conditions as fatal, preventing the application from starting.

  Default: 30

## 7.2 Binding Properties

Binding properties are supplied using the format `spring.cloud.stream.bindings.<channelName>.<property>=<value>`. The `<channelName>` represents the name of the channel being configured (e.g., `output` for a `Source`).

To avoid repetition, Spring Cloud Stream supports setting values for all channels, in the format `spring.cloud.stream.default.<property>=<value>`.

In what follows, we indicate where we have omitted the `spring.cloud.stream.bindings.<channelName>.` prefix and focus just on the property name, with the understanding that the prefix will be included at runtime.

### Properties for Use of Spring Cloud Stream

The following binding properties are available for both input and output bindings and must be prefixed with `spring.cloud.stream.bindings.<channelName>.`, e.g. `spring.cloud.stream.bindings.input.destination=ticktock`.

Default values can be set by using the prefix `spring.cloud.stream.default`, e.g. `spring.cloud.stream.default.contentType=application/json`.

destination
: The target destination of a channel on the bound middleware (e.g., the RabbitMQ exchange or Kafka topic). If the channel is bound as a consumer, it could be bound to multiple destinations and the destination names can be specified as comma separated String values. If not set, the channel name is used instead. The default value of this property cannot be overridden.

group
: The consumer group of the channel. Applies only to inbound bindings. See Consumer Groups.

  Default: null (indicating an anonymous consumer).

contentType
: The content type of the channel.

  Default: null (so that no type coercion is performed).

binder
: The binder used by this binding. See Section 6.4, “Multiple Binders on the Classpath” for details.

  Default: null (the default binder will be used, if one exists).

### Consumer properties

The following binding properties are available for input bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.consumer.`, e.g. `spring.cloud.stream.bindings.input.consumer.concurrency=3`.

Default values can be set by using the prefix `spring.cloud.stream.default.consumer`, e.g. `spring.cloud.stream.default.consumer.headerMode=none`.

concurrency
: The concurrency of the inbound consumer.

  Default: `1`.

partitioned
: Whether the consumer receives data from a partitioned producer.

  Default: `false`.

headerMode
: When set to `none`, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when consuming data from non-Spring Cloud Stream applications when native headers are not supported. When set to `headers`, uses the middleware's native header mechanism. When set to `embeddedHeaders`, embeds headers into the message payload.

  Default: depends on binder implementation.

maxAttempts
: If processing fails, the number of attempts to process the message (including the first). Set to 1 to disable retry.

  Default: `3`.

backOffInitialInterval
: The backoff initial interval on retry.

  Default: `1000`.

backOffMaxInterval
: The maximum backoff interval.

  Default: `10000`.

backOffMultiplier
: The backoff multiplier.

  Default: `2.0`.

instanceIndex
: When set to a value greater than equal to zero, allows customizing the instance index of this consumer (if different from `spring.cloud.stream.instanceIndex`). When set to a negative value, it will default to `spring.cloud.stream.instanceIndex`. See that property for more information.

  Default: `-1`.

instanceCount
: When set to a value greater than equal to zero, allows customizing the instance count of this consumer (if different from `spring.cloud.stream.instanceCount`). When set to a negative value, it will default to `spring.cloud.stream.instanceCount`. See that property for more information.

  Default: `-1`.

## Producer Properties

The following binding properties are available for output bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.producer.`, e.g. `spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id`.

Default values can be set by using the prefix `spring.cloud.stream.default.producer`, e.g. `spring.cloud.stream.default.producer.partitionKeyExpression=payload.id`.

partitionKeyExpression
: A SpEL expression that determines how to partition outbound data. If set, or if `partitionKeyExtractorClass` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Section 4.6, “Partitioning Support”.

  Default: null.

partitionKeyExtractorClass
: A `PartitionKeyExtractorStrategy` implementation. If set, or if `partitionKeyExpression` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Section 4.6, “Partitioning Support”.

  Default: null.

partitionSelectorClass
: A `PartitionSelectorStrategy` implementation. Mutually exclusive with `partitionSelectorExpression`. If neither is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

  Default: null.

partitionSelectorExpression
: A SpEL expression for customizing partition selection. Mutually exclusive with `partitionSelectorClass`. If neither is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

  Default: null.

partitionCount
: The number of target partitions for the data, if partitioning is enabled. Must be set to a value greater than 1 if the producer is partitioned. On Kafka, interpreted as a hint; the larger of this and the partition count of the target topic is used instead.

  Default: `1`.

requiredGroups
: A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (e.g., by pre-creating durable queues in RabbitMQ).

headerMode
: When set to `none`, disables header embedding on output. Effective only for messaging middleware that does not support message headers natively and requires header embedding. This option is useful when producing data for non-Spring Cloud Stream applications when native headers are not supported. When set to `headers`, uses the middleware’s native header mechanism. When set to `embeddedHeaders`, embeds headers into the message payload.

  Default: Depends on binder implementation.

useNativeEncoding

When set to `true`, the outbound message is serialized directly by client library, which must be configured correspondingly (e.g. setting an appropriate Kafka producer value serializer). When this configuration is being used, the outbound message marshalling is not based on the `contentType` of the binding. When native encoding is used, it is the responsibility of the consumer to use appropriate decoder (ex: Kafka consumer value de-serializer) to deserialize the inbound message. Also, when native encoding/decoding is used the `headerMode=embeddedHeaders` property is ignored and headers will not be embedded into the message.

Default: `false`.

errorChannelEnabled

When set to `true`, if the binder supports async send results; send failures will be sent to an error channel for the destination. See the section called “Message Channel Binders and Error Channels” for more information.

Default: `false`.

## 7.3 Using dynamically bound destinations

Besides the channels defined via `@EnableBinding`, Spring Cloud Stream allows applications to send messages to dynamically bound destinations. This is useful, for example, when the target destination needs to be determined at runtime. Applications can do so by using the `BinderAwareChannelResolver` bean, registered automatically by the `@EnableBinding` annotation.

The property 'spring.cloud.stream.dynamicDestinations' can be used for restricting the dynamic destination names to a set known beforehand (whitelisting). If the property is not set, any destination can be bound dynamically.

The `BinderAwareChannelResolver` can be used directly as in the following example, in which a REST controller uses a path variable to decide the target channel.

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;

    @RequestMapping(path = "/{target}", method = POST, consumes = "*/*")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @PathVariable("target") target,
           @RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
        sendMessage(body, target, contentType);
    }

    private void sendMessage(String body, String target, Object contentType) {
        resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE,
 contentType))));
    }
}
```

After starting the application on the default port 8080, when sending the following data:

```
curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders
```

The destinations 'customers' and 'orders' are created in the broker (for example: exchange in case of Rabbit or topic in case of Kafka) with the names 'customers' and 'orders', and the data is published to the appropriate destinations.

The `BinderAwareChannelResolver` is a general purpose Spring Integration `DestinationResolver` and can be injected in other components. For example, in a router using a SpEL expression based on the `target` field of an incoming JSON message.

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

    @Autowired
    private BinderAwareChannelResolver resolver;


    @RequestMapping(path = "/", method = POST, consumes = "application/json")
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void handleRequest(@RequestBody String body, @RequestHeader(HttpHeaders.CONTENT_TYPE) Object
 contentType) {
        sendMessage(body, contentType);
    }

    private void sendMessage(Object body, Object contentType) {
        routerChannel().send(MessageBuilder.createMessage(body,
                new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE,
 contentType))));
    }

    @Bean(name = "routerChannel")
    public MessageChannel routerChannel() {
        return new DirectChannel();
    }

    @Bean
    @ServiceActivator(inputChannel = "routerChannel")
    public ExpressionEvaluatingRouter router() {
        ExpressionEvaluatingRouter router =
            new ExpressionEvaluatingRouter(new
 SpelExpressionParser().parseExpression("payload.target"));
        router.setDefaultOutputChannelName("default-output");
        router.setChannelResolver(resolver);
        return router;
    }
}
```

The Router Sink Application uses this technique to create the destinations on-demand.

If the channel names are known in advance, you can configure the producer properties as with any other destination. Alternatively, if you register a `NewBindingCallback<>` bean, it will be invoked just before the binding is created. The callback takes the generic type of the extended producer properties used by the binder; it has one method:

```
void configure(String channelName, MessageChannel channel, ProducerProperties producerProperties,
        T extendedProducerProperties);
```

The following is an example using the RabbitMQ binder:

```
@Bean
public NewBindingCallback<RabbitProducerProperties> dynamicConfigurer() {
    return (name, channel, props, extended) -> {
        props.setRequiredGroups("bindThisQueue");
        extended.setQueueNameGroupOnly(true);
        extended.setAutoBindDlq(true);
        extended.setDeadLetterQueueName("myDLQ");
```

```
    };
}
```

**Note**

If you need to support dynamic destinations with multiple binder types, use `Object` for the generic type and cast the `extended` argument as needed.

# 8. Content Type negotiation

## 8.1 Introduction

Data transformation is one of the core features of any message-driven microservice architecture. Given that in Spring Cloud Stream, such data is represented as a Spring `Message`, such message may have to be transformed to a desired shape/size before reaching its destination. This is required for two reasons:

*1. To convert the contents of the incoming message to match the signature of the application-provided handler.*

*2. To convert the contents of the outgoing message to the wire format.*

The wire format is typically `byte[]` (i.e., Kafka and Rabbit binders), but is governed by the binder implementation.

In Spring Cloud Stream, message transformation is accomplished with a `org.springframework.messaging.converter.MessageConverter`.

**Note**

As a supplement to the details to follow you may also want to read the following blog

## 8.2 Mechanics

To better understand the mechanics and the necessity behind content-type negotiation let's look at the very simple use case using the following message handler as an example. Also let's assume that this is the only handler in the application (no internal pipeline) for simplicity.

```
@StreamListener(Processor.INPUT)
@SendTo(Processor.OUTPUT)
public String handle(Person person) {..}
```

The above handler expects `Person` type as an argument and will produce `String` type as an output. In order for the framework to succeed in passing the incoming `Message` as an argument to this handler it has to somehow transform the payload of the `Message` from the wire format to `Person` type. In other words the framework must locate and apply the appropriate `MessageConverter`. To accomplish that the framework needs some instructions from the user. One of these instructions is already provided by the signature of the handler method itself (`Person` type), so in theory, that should and in some cases is enough, but for the majority of the use cases in order to select the appropriate `MessageConverter` the framework needs an additional piece of information. That missing piece is `contentType`.

Spring Cloud Stream provides three simple mechanisms to define `contentType` and they all come with precedence order:

*1. **HEADER** - the `contentType` can be communicated through the Message itself. By simply providing `contentType` header you are declaring the content type to use to locate and apply the appropriate MessageConverter.*

*2. **BINDING** - the `contentType` can be set per destination binding via `spring.cloud.stream.bindings.input.content-type` property. NOTE: the segment `input` in the property name corresponds to the actual name of the destination which is "input" in our case. This*

*approach allows one to declare per-binding the content type to use to locate and apply the appropriate MessageConverter.*

*3.* ***DEFAULT*** *- in the event `contentType` is not present in the Message header and/or binding, the default `application/json` content type will be used to locate and apply the appropriate MessageConverter.*

As mentioned, the above also demonstrates the order of precedence in the event there is a tie. For example, header provided content type takes precedence over any other content type. The same applies for content type set per binding which essentially allows one to override the default content type. But it also provides a sensible default which was determined from the community feedback.

Another reason for making `application/json` the default stems from the interoperability requirements driven by distributed microservices architectures where producer and consumer not only run in different JVMs, but can also run on different non-JVM platforms.

Once the non-void handler method returns and unless the return value is already a `Message`, the new `Message` is constructed with return vlaue as the payload while inheriting headers from the input `Message` less the ones defined/filtered by `SpringIntegrationProperties.messageHandlerNotPropagatedHeaders`. By default, there is only one header set there - `contentType`. This means that the new `Message` will not have `contentType` header set, thus ensuring that the `contentType` can evolve. You can always opt out to returning a `Message` from the handler method where you can inject any header you wish.

If there is an internal pipeline the `Message` is sent to the next handler going through the same process of conversion, or if there is no internal pipeline or you've reached the end of it the `Message` is sent back to the output destination.

## Content type vs. argument type

As it was mentioned, for the framework to select the appropriate MessageConverter it requires *argument type* and optionally *content type* information. The logic for selecting the appropriate `MessageConverter` resides with the argument resolvers (`HandlerMethodArgumentResolvers`), right before the invocation of the user defined handler method (that is when the actual argument type is known to the framework). If argument type does NOT match the type of the current payload the framework delegates to the stack of the pre-configured `MessageConverters` to see if any one of them can convert the payload. As you can see the `Object fromMessage(Message<?> message, Class<?> targetClass);` operation of the MessageConverter takes `targetClass` as one of its arguments. The framework also ensures that the provided `Message` always contains `contentType` header in the event one was not there already (injects the default one or the one set per binding). That is the mechanism by which framework determines if message can be converted to a target type - `contentType` and argumenyt type. If no appropriate `MessageConverter` is found the exception is thrown at which time you can add custom `MessageConverter` (more on this later).

But what if the payload type matches the target type declared by the handler method? In this cases there is obviously nothing to convert and the payload will be passed unmodified. While this sounds pretty straight forward and logical, keep in mind handler methods that take `Message<?>` and/or `Object` as an argument. By doing so you are essentially forfeiting the conversion process by declaring the target type to be `Object` which is an `instanceof` everything in Java.

In other words:

**Note**

Do NOT expect Message to be converted into some type based on the `contentType` only. Remember that the `contentType` is complimentary to the target type. A hint if you wish which `MessageConverter` may or may not take into consideration.

### Message Converters

`MessageConverters` define two methods:

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);
```

It is important to understand the contract of these methods and their usage specifically in the context of Spring Cloud Stream.

The `fromMessage` method converts incoming `Message` to an argument type. The payload of the `Message` could be *any type* and it's up to the actual implementation of the `MessageConverter` to support multiple types. For example, some JSON converter may support the payload type as `byte[]` and `String` etc. This is important when application contains an internal pipeline (i.e., *input # handler1 # handler2 #. . . # output*) and the output of the upstream handler results in a `Message` which may not be in the initial wire format.

However. . .

The `toMessage` method has a more strict contract and must always convert `Message` to the wire format - `byte[]`.

So for all intents and purposes (and especially when implementing your own converter) you might as well look at them as:

```
Object fromMessage(Message<?> message, Class<?> targetClass);

Message<byte[]> toMessage(Object payload, @Nullable MessageHeaders headers);
```

## 8.3 Provided MessageConverters

As it was mentioned earlier the framework already provides a stack of `MessageConverters` to handle most common use cases. Below is the ordered list of provided `MessageConverters`.

**Note**

It is important to understand the importance of the order since the mechanism by which the framework locates the appropriate `MessageConverter` is by iterating through each and asking if it can convert using the first one that can convert.

1. `ApplicationJsonMessageMarshallingConverter` - *variation of the `org.springframework.messaging.converter.MappingJackson2MessageConverter`. Supports conversion of the payload of the `Message` from `String` or `byte[]`.*
2. `TupleJsonMessageConverter` - ***[DEPRECATED]*** *Supports conversion of the payload of the `Message` from `org.springframework.tuple.Tuple`.*
3. `ByteArrayMessageConverter` - *Supports conversion of the payload of the `Message` from `byte[]` to `byte[]` for cases when `contentType` is set to `application/octet-stream`. Essentially a pass through and exists primarily for backward compatibility.*

4. `ObjectStringMessageConverter` - *Supports conversion of any type to a `String`, when contentType is `text/plain`. Invokes Object's `toString()` method or if payload is `byte[]` then new `String(byte[])`.*

5. `JavaSerializationMessageConverter` - ***[DEPRECATED]** Supports conversion based on java serialization when `contentType` is `application/x-java-serialized-object`.*

6. `KryoMessageConverter` - ***[DEPRECATED]** Supports conversion based on kryo serialization when `contentType` is `application/x-java-object`.*

7. `JsonUnmarshallingConverter` - *Similar to the `ApplicationJsonMessageMarshallingConverter`. Supports conversion of any type when `contentType` is `application/x-java-object`. Expects the actual type information to be embedded in the `contentType` as an attribute (e.g., `application/x-java-object;type=foo.bar.Baz`).*

In the event no appropriate converter is found the framework will throw an exception at which point you should check your code and configfuration and ensure you didn't miss anything (i.e., provide `contentType` via binding or header). However, most likely you are dealing with some uncommon case (custom `contentType` perhaps) and the current stack of provided `MessageConverters` doesn't know how to convert. And if that's the case you can add custom `MessageConverter`.

## 8.4 User defined Message Converters

Spring Cloud Stream exposes a mechanism to define and register additional `MessageConverters`. All you need to do is implement `org.springframework.messaging.converter.MessageConverter`, confiure it as `@Bean` and annotate it with `@StreamMessageConverter` and it will be added to the existing stack of `MessageConverters`. The `@StreamMessageConverter` qualifier annotation is to avoid picking up other converters that may be present on the *Application Context*.

**Note**

It is important to undetrstand that custom `MessageConverters` are added to the head of the existing stack. This allows custom `MessageConverters` to take precedence over the existing ones, thus supporting not only addition, but the override of the existing ones.

Here is an example of creating a message converter bean to support new content type `application/bar`:

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

    ...

    @Bean
    @StreamConverter
    public MessageConverter customMessageConverter() {
        return new MyCustomMessageConverter();
    }
}
```

```
public class MyCustomMessageConverter extends AbstractMessageConverter {

    public MyCustomMessageConverter() {
        super(new MimeType("application", "bar"));
```

```
    }

    @Override
    protected boolean supports(Class<?> clazz) {
        return (Bar.class.equals(clazz));
    }

    @Override
    protected Object convertFromInternal(Message<?> message, Class<?> targetClass, Object
 conversionHint) {
        Object payload = message.getPayload();
        return (payload instanceof Bar ? payload : new Bar((byte[]) payload));
    }
}
```

Spring Cloud Stream also provides support for Avro-based converters and schema evolution. See the specific section for details.

# 9. Schema evolution support

Spring Cloud Stream provides support for schema-based message converters through its `spring-cloud-stream-schema` module. Currently, the only serialization format supported out of the box for schema-based message converters is Apache Avro, with more formats to be added in future versions.

## 9.1 Apache Avro Message Converters

The `spring-cloud-stream-schema` module contains two types of message converters that can be used for Apache Avro serialization:

- converters using the class information of the serialized/deserialized objects, or a schema with a location known at startup;
- converters using a schema registry - they locate the schemas at runtime, as well as dynamically registering new schemas as domain objects evolve.

## 9.2 Converters with schema support

The `AvroSchemaMessageConverter` supports serializing and deserializing messages either using a predefined schema or by using the schema information available in the class (either reflectively, or contained in the `SpecificRecord`). If the target type of the conversion is a `GenericRecord`, then a schema must be set.

For using it, you can simply add it to the application context, optionally specifying one ore more `MimeTypes` to associate it with. The default `MimeType` is `application/avro`.

Here is an example of configuring it in a sink application registering the Apache Avro `MessageConverter`, without a predefined schema:

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      return new AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes"));
  }
}
```

Conversely, here is an application that registers a converter with a predefined schema, to be found on the classpath:

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      AvroSchemaMessageConverter converter = new AvroSchemaMessageConverter(MimeType.valueOf("avro/
bytes"));
      converter.setSchemaLocation(new ClassPathResource("schemas/User.avro"));
      return converter;
  }
}
```

In order to understand the schema registry client converter, we will describe the schema registry support first.

## 9.3 Schema Registry Support

Most serialization models, especially the ones that aim for portability across different platforms and languages, rely on a schema that describes how the data is serialized in the binary payload. In order to serialize the data and then to interpret it, both the sending and receiving sides must have access to a schema that describes the binary format. In certain cases, the schema can be inferred from the payload type on serialization, or from the target type on deserialization, but in a lot of cases applications benefit from having access to an explicit schema that describes the binary data format. A schema registry allows you to store schema information in a textual format (typically JSON) and makes that information accessible to various applications that need it to receive and send data in binary format. A schema is referenceable as a tuple consisting of:

- a *subject* that is the logical name of the schema;
- the schema *version*;
- the schema *format* which describes the binary format of the data.

## 9.4 Schema Registry Server

Spring Cloud Stream provides a schema registry server implementation. In order to use it, you can simply add the `spring-cloud-stream-schema-server` artifact to your project and use the `@EnableSchemaRegistryServer` annotation, adding the schema registry server REST controller to your application. This annotation is intended to be used with Spring Boot web applications, and the listening port of the server is controlled by the `server.port` setting. The `spring.cloud.stream.schema.server.path` setting can be used to control the root path of the schema server (especially when it is embedded in other applications). The `spring.cloud.stream.schema.server.allowSchemaDeletion` boolean setting enables the deletion of schema. By default this is disabled.

The schema registry server uses a relational database to store the schemas. By default, it uses an embedded database. You can customize the schema storage using the Spring Boot SQL database and JDBC configuration options.

A Spring Boot application enabling the schema registry looks as follows:

```
@SpringBootApplication
@EnableSchemaRegistryServer
public class SchemaRegistryServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SchemaRegistryServerApplication.class, args);
    }
}
```

### Schema Registry Server API

The Schema Registry Server API consists of the following operations:

`POST /`

Register a new schema.

Accepts JSON payload with the following fields:

- `subject` the schema subject;
- `format` the schema format;
- `definition` the schema definition.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /{subject}/{format}/{version}`

Retrieve an existing schema by its subject, format and version.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /{subject}/{format}`

Retrieve a list of existing schema by its subject and format.

Response is a list of schemas with each schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`GET /schemas/{id}`

Retrieve an existing schema by its id.

Response is a schema object in JSON format, with the following fields:

- `id` the schema id;
- `subject` the schema subject;
- `format` the schema format;
- `version` the schema version;
- `definition` the schema definition.

`DELETE /{subject}/{format}/{version}`

Delete an existing schema by its subject, format and version.

`DELETE /schemas/{id}`

Delete an existing schema by its id.

`DELETE /{subject}`

Delete existing schemas by their subject.

**Note**

This note applies to users of Spring Cloud Stream 1.1.0.RELEASE only. Spring Cloud Stream 1.1.0.RELEASE used the table name `schema` for storing `Schema` objects, which is a keyword in a number of database implementations. To avoid any conflicts in the future, starting with 1.1.1.RELEASE we have opted for the name `SCHEMA_REPOSITORY` for the storage table. Any Spring Cloud Stream 1.1.0.RELEASE users that are upgrading are advised to migrate their existing schemas to the new table before upgrading.

## 9.5 Schema Registry Client

The client-side abstraction for interacting with schema registry servers is the `SchemaRegistryClient` interface, with the following structure:

```
public interface SchemaRegistryClient {

    SchemaRegistrationResponse register(String subject, String format, String schema);

    String fetch(SchemaReference schemaReference);

    String fetch(Integer id);

}
```

Spring Cloud Stream provides out of the box implementations for interacting with its own schema server, as well as for interacting with the Confluent Schema Registry.

A client for the Spring Cloud Stream schema registry can be configured using the `@EnableSchemaRegistryClient` as follows:

```
  @EnableBinding(Sink.class)
  @SpringBootApplication
  @EnableSchemaRegistryClient
  public static class AvroSinkApplication {
    ...
  }
```

**Note**

The default converter is optimized to cache not only the schemas from the remote server but also the `parse()` and `toString()` methods that are quite expensive. Because of this, it uses a `DefaultSchemaRegistryClient` that does not caches responses. If you intend to use the client directly on your code, you can request a bean that also caches responses to be created. To do that, just add the property `spring.cloud.stream.schemaRegistryClient.cached=true` to your application properties.

### Using Confluent's Schema Registry

The default configuration will create a `DefaultSchemaRegistryClient` bean. If you want to use the Confluent schema registry, you need to create a bean of type `ConfluentSchemaRegistryClient`, which will supersede the one configured by default by the framework.

```
@Bean
public SchemaRegistryClient
 schemaRegistryClient(@Value("${spring.cloud.stream.schemaRegistryClient.endpoint}") String endpoint){
  ConfluentSchemaRegistryClient client = new ConfluentSchemaRegistryClient();
  client.setEndpoint(endpoint);
  return client;
}
```

**Note**

The ConfluentSchemaRegistryClient is tested against Confluent platform version 3.2.2.

### Schema Registry Client properties

The Schema Registry Client supports the following properties:

spring.cloud.stream.schemaRegistryClient.endpoint
: The location of the schema-server. Use a full URL when setting this, including protocol (`http` or `https`) , port and context path.

Default
: [localhost:8990/](localhost:8990/)

spring.cloud.stream.schemaRegistryClient.cached
: Whether the client should cache schema server responses. Normally set to `false`, as the caching happens in the message converter. Clients using the schema registry client should set this to `true`.

Default
: `true`

## 9.6 Avro Schema Registry Client Message Converters

For Spring Boot applications that have a `SchemaRegistryClient` bean registered with the application context, Spring Cloud Stream will auto-configure an Apache Avro message converter that uses the schema registry client for schema management. This eases schema evolution, as applications that receive messages can get easy access to a writer schema that can be reconciled with their own reader schema.

For outbound messages, the `MessageConverter` will be activated if the content type of the channel is set to `application/*+avro`, e.g.:

```
spring.cloud.stream.bindings.output.contentType=application/*+avro
```

During the outbound conversion, the message converter will try to infer the schemas of the outbound messages based on their type and register them to a subject based on the payload type using the `SchemaRegistryClient`. If an identical schema is already found, then a reference to it will be retrieved. If not, the schema will be registered and a new version number will be provided. The message will be sent with a `contentType` header using the scheme `application/[prefix].[subject].v[version]+avro`, where `prefix` is configurable and `subject` is deduced from the payload type.

For example, a message of the type `User` may be sent as a binary payload with a content type of `application/vnd.user.v2+avro`, where `user` is the subject and `2` is the version number.

When receiving messages, the converter will infer the schema reference from the header of the incoming message and will try to retrieve it. The schema will be used as the writer schema in the deserialization process.

### Avro Schema Registry Message Converter properties

If you have enabled Avro based schema registry client by setting `spring.cloud.stream.bindings.output.contentType=application/*+avro` you can customize the behavior of the registration with the following properties.

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled
: Enable if you want the converter to use reflection to infer a Schema from a POJO.

Default
: `false`

spring.cloud.stream.schema.avro.readerSchema
: Avro compares schema versions by looking at a writer schema (origin payload) and a reader schema (your application payload), check Avro documentation for more information. If set, this overrides any lookups at the schema server and uses the local schema as the reader schema.

Default
: `null`

spring.cloud.stream.schema.avro.schemaLocations
: Register any `.avsc` files listed in this property with the Schema Server.

Default
: `empty`

spring.cloud.stream.schema.avro.prefix
: The prefix to be used on the Content-Type header.

Default
: `vnd`

## 9.7 Schema Registration and Resolution

To better understand how Spring Cloud Stream registers and resolves new schemas, as well as its use of Avro schema comparison features, we will provide two separate subsections below: one for the registration, and one for the resolution of schemas.

## Schema Registration Process (Serialization)

The first part of the registration process is extracting a schema from the payload that is being sent over a channel. Avro types such as `SpecificRecord` or `GenericRecord` already contain a schema, which can be retrieved immediately from the instance. In the case of POJOs a schema will be inferred if the property `spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled` is set to `true` (the default).

*Figure 9.1. Schema Writer Resolution Process*

Ones a schema is obtained, the converter will then load its metadata (version) from the remote server. First it queries a local cache, and if not found it then submits the data to the server that will reply with versioning information. The converter will always cache the results to avoid the overhead of querying the Schema Server for every new message that needs to be serialized.

*Figure 9.2. Schema Registration Process*

With the schema version information, the converter sets the `contentType` header of the message to carry the version information such as `application/vnd.user.v1+avro`

## Schema Resolution Process (Deserialization)

When reading messages that contain version information (i.e. a `contentType` header with a scheme like above), the converter will query the Schema server to fetch the **writer** schema of the message. Ones it has found the correct schema of the incoming message, it then retrieves the reader schema and using Avro's schema resolution support reads it into the reader definition (setting defaults and missing properties).

*Figure 9.3. Schema Reading Resolution Process*

**Note**

It's important to understand the difference between a writer schema (the application that wrote the message) and a reader schema (the receiving application). Please take a moment to read the Avro terminology and understand the process. Spring Cloud Stream will always fetch the writer schema to determine how to read a message. If you want to get Avro's schema evolution support working you need to make sure that a readerSchema was properly set for your application.

# 10. Inter-Application Communication

## 10.1 Connecting Multiple Application Instances

While Spring Cloud Stream makes it easy for individual Spring Boot applications to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-application pipelines, where microservice applications send data to each other. You can achieve this scenario by correlating the input and output destinations of adjacent applications.

Supposing that a design calls for the Time Source application to send data to the Log Sink application, you can use a common destination named `ticktock` for bindings within both applications.

Time Source (that has the channel name `output`) will set the following property:

```
spring.cloud.stream.bindings.output.destination=ticktock
```

Log Sink (that has the channel name `input`) will set the following property:

```
spring.cloud.stream.bindings.input.destination=ticktock
```

## 10.2 Instance Index and Instance Count

When scaling up Spring Cloud Stream applications, each instance can receive information about how many other instances of the same application exist and what its own instance index is. Spring Cloud Stream does this through the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties. For example, if there are three instances of a HDFS sink application, all three instances will have `spring.cloud.stream.instanceCount` set to `3`, and the individual applications will have `spring.cloud.stream.instanceIndex` set to `0`, `1`, and `2`, respectively.

When Spring Cloud Stream applications are deployed via Spring Cloud Data Flow, these properties are configured automatically; when Spring Cloud Stream applications are launched independently, these properties must be set correctly. By default, `spring.cloud.stream.instanceCount` is `1`, and `spring.cloud.stream.instanceIndex` is `0`.

In a scaled-up scenario, correct configuration of these two properties is important for addressing partitioning behavior (see below) in general, and the two properties are always required by certain binders (e.g., the Kafka binder) in order to ensure that data are split correctly across multiple consumer instances.

## 10.3 Partitioning

### Configuring Output Bindings for Partitioning

An output binding is configured to send partitioned data by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorName` (see next paragraph) properties, as well as its `partitionCount` property.

For example, the following is a valid and typical configuration:

```
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id
spring.cloud.stream.bindings.output.producer.partitionCount=5
```

Based on the above example configuration, data will be sent to the target partition using the following logic.

A partition key's value is calculated for each message sent to a partitioned output channel based on the `partitionKeyExpression`. The `partitionKeyExpression` is a SpEL expression which is evaluated against the outbound message for extracting the partitioning key.

If a SpEL expression is not sufficient for your needs, you can instead calculate the partition key value by providing implementation of `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` and configuring it as a bean (i.e., `@Bean`). In the event you have more then one bean of type `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` available in the Application Context you can further filter it by specifying its name via `partitionKeyExtractorName` property:

```
--spring.cloud.stream.bindings.output.producer.partitionKeyExtractorName=customPartitionKeyExtractor
--spring.cloud.stream.bindings.output.producer.partitionCount=5
. . .
@Bean
public CustomPartitionKeyExtractorClass customPartitionKeyExtractor() {
    return new CustomPartitionKeyExtractorClass();
}
```

**Note**

In previous versions of Spring Cloud Stream you could specify the implementation of `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` as `spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass` property. Since version 2.0 this property is deprecated and support for it will be removed in a future version.

Ones the message key is calculated, the partition selection process will determine the target partition as a value between `0` and `partitionCount - 1`. The default calculation, applicable in most scenarios, is based on the formula `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the 'key' (via the `partitionSelectorExpression` property) or by configuring an implementation of `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` as a bean (i.e., @Bean). And similarly to the `PartitionKeyExtractorStrategy` you can further filter it using `spring.cloud.stream.bindings.output.producer.partitionSelectorName` property in the event there are more then one bean of this type is available in the Application Context.

```
--spring.cloud.stream.bindings.output.producer.partitionSelectorName=customPartitionSelector
. . .
@Bean
public CustomPartitionSelectorClass customPartitionSelector() {
    return new CustomPartitionSelectorClass();
}
```

**Note**

In previous versions of Spring Cloud Stream you could specify the implementation of `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` as `spring.cloud.stream.bindings.output.producer.partitionSelectorClass` property. Since version 2.0 this property is deprecated and support for it will be removed in a future version.

#### Configuring Input Bindings for Partitioning

An input binding (with the channel name `input`) is configured to receive partitioned data by setting its `partitioned` property, as well as the `instanceIndex` and `instanceCount` properties on the application itself, as in the following example:

```
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceIndex=3
spring.cloud.stream.instanceCount=5
```

The `instanceCount` value represents the total number of application instances between which the data need to be partitioned, and the `instanceIndex` must be a unique value across the multiple instances, between `0` and `instanceCount - 1`. The instance index helps each application instance to identify the unique partition(s) from which it receives data. It is required by binders using technology that doesn’t support partitioning natively, for example, with RabbitMQ, there is a queue for each partition, with the queue name containing the instance index. With Kafka, if `autoRebalanceEnabled` is `true` (default), Kafka will take care of distributing partitions across instances and these properties are not required. If `autoRebalanceEnabled` is set to false, the `instanceCount` and `instanceIndex` are used by the binder to determine which partition(s) the instance will subscribe to (you must have at least as many partitions as there are instances). The binder will allocate the partitions instead of Kafka. This might be useful if you want messages for a particular partition to always go to the same instance. When a binder configuration that requires them, it is important to set both values correctly in order to ensure that all of the data is consumed and that the application instances receive mutually exclusive datasets.

While a scenario which using multiple instances for partitioned data processing may be complex to set up in a standalone case, Spring Cloud Dataflow can simplify the process significantly by populating both the input and output values correctly as well as relying on the runtime infrastructure to provide information about the instance index and instance count.

# 11. Testing

Spring Cloud Stream provides support for testing your microservice applications without connecting to a messaging system. You can do that by using the `TestSupportBinder` provided by the `spring-cloud-stream-test-support` library, which can be added as a test dependency to the application:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

**Note**

The `TestSupportBinder` uses the Spring Boot autoconfiguration mechanism to supersede the other binders found on the classpath. Therefore, when adding a binder as a dependency, make sure that the `test` scope is being used.

The `TestSupportBinder` allows users to interact with the bound channels and inspect what messages are sent and received by the application

For outbound message channels, the `TestSupportBinder` registers a single subscriber and retains the messages emitted by the application in a `MessageCollector`. They can be retrieved during tests and have assertions made against them.

The user can also send messages to inbound message channels, so that the consumer application can consume the messages. The following example shows how to test both input and output channels on a processor.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ExampleTest {

  @Autowired
  private Processor processor;

  @Autowired
  private MessageCollector messageCollector;

  @Test
  @SuppressWarnings("unchecked")
  public void testWiring() {
    Message<String> message = new GenericMessage<>("hello");
    processor.input().send(message);
    Message<String> received = (Message<String>) messageCollector.forChannel(processor.output()).poll();
    assertThat(received.getPayload(), equalTo("hello world"));
  }


  @SpringBootApplication
  @EnableBinding(Processor.class)
  public static class MyProcessor {

    @Autowired
    private Processor channels;

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public String transform(String in) {
      return in + " world";
    }
  }
}
```

In the example above, we are creating an application that has an input and an output channel, bound through the `Processor` interface. The bound interface is injected into the test so we can have access to both channels. We are sending a message on the input channel and we are using the `MessageCollector` provided by Spring Cloud Stream's test support to capture the message has been sent to the output channel as a result. Ones we have received the message, we can validate that the component functions correctly.

## 11.1 Disabling the test binder autoconfiguration

The intent behind the test binder superseding all the other binders on the classpath is to make it easy to test your applications without making changes to your production dependencies. In some cases (e.g. integration tests) it is useful to use the actual production binders instead, and that requires disabling the test binder autoconfiguration. In order to do so, you can exclude the `org.springframework.cloud.stream.test.binder.TestSupportBinderAutoConfiguration` class using one of the Spring Boot autoconfiguration exclusion mechanisms, as in the following example.

```
@SpringBootApplication(exclude = TestSupportBinderAutoConfiguration.class)
@EnableBinding(Processor.class)
public static class MyProcessor {

    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    public String transform(String in) {
        return in + " world";
    }
}
```

When autoconfiguration is disabled, the test binder is available on the classpath, and its `defaultCandidate` property is set to `false`, so that it does not interfere with the regular user configuration. It can be referenced under the name `test` e.g.:

```
spring.cloud.stream.defaultBinder=test
```

# 12. Health Indicator

Spring Cloud Stream provides a health indicator for binders. It is registered under the name of `binders` and can be enabled or disabled by setting the `management.health.binders.enabled` property.

# 13. Metrics Emitter

Spring Cloud Stream provides a module called `spring-cloud-stream-metrics` that can be used to emit any available metric from Spring Boot metrics endpoint to a named channel. This module allow operators to collect metrics from stream applications without relying on polling their endpoints.

The module is activated when you set the destination name for metrics binding, e.g. `spring.cloud.stream.bindings.applicationMetrics.destination=<DESTINATION_NAME>`. `applicationMetrics` can be configured in a similar fashion to any other producer binding. The default `contentType` setting of `applicationMetrics` is `application/json`.

The following properties can be used for customizing the emission of metrics:

spring.cloud.stream.metrics.key
: The name of the metric being emitted. Should be an unique value per application.

Default
: `${spring.application.name:${vcap.application.name:${spring.config.name:application}}}`

spring.cloud.stream.metrics.prefix
: Prefix string to be prepended to the metrics key.

  Default: ``

spring.cloud.stream.metrics.properties
: Just like the `includes` option, it allows white listing application properties that will be added to the metrics payload

  Default: null.

A detailed overview of the metrics export process can be found in the Spring Boot reference documentation. Spring Cloud Stream provides a metric exporter named `application` that can be configured via regular Spring Boot metrics configuration properties.

The exporter can be configured either by using the global Spring Boot configuration settings for exporters, or by using exporter-specific properties. For using the global configuration settings, the properties should be prefixed by `spring.metric.export` (e.g. `spring.metric.export.includes=integration**`). These configuration options will apply to all exporters (unless they have been configured differently). Alternatively, if it is intended to use configuration settings that are different from the other exporters (e.g. for restricting the number of metrics published), the Spring Cloud Stream provided metrics exporter can be configured using the prefix `spring.metrics.export.triggers.application` (e.g. `spring.metrics.export.triggers.application.includes=integration**`).

**Note**

Due to Spring Boot's relaxed binding the value of a property being included can be slightly different than the original value.

As a rule of thumb, the metric exporter will attempt to normalize all the properties in a consistent format using the dot notation (e.g. `JAVA_HOME` becomes `java.home`).

The goal of normalization is to make downstream consumers of those metrics capable of receiving property names consistently, regardless of how they are set on the monitored

application (`--spring.application.name` or `SPRING_APPLICATION_NAME` would always yield `spring.application.name`).

Below is a sample of the data published to the channel in JSON format by the following command:

```
java -jar time-source.jar \
    --spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \
    --spring.cloud.stream.metrics.properties=spring.application** \
    --spring.metrics.export.includes=integration.channel.input**,integration.channel.output**
```

The resulting JSON is:

```
{
   "name":"time-source",
   "metrics":[
      {
         "name":"integration.channel.output.errorRate.mean",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.max",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.min",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.stdev",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.errorRate.count",
         "value":0.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendCount",
         "value":6.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.mean",
         "value":0.994885872292989,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.max",
         "value":1.006247080013156,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.min",
         "value":1.0012035220116378,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.stdev",
         "value":6.505181111084848E-4,
         "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
         "name":"integration.channel.output.sendRate.count",
         "value":6.0,
         "timestamp":"2017-04-11T16:56:35.790Z"
```

```
      }
   ],
   "createdTime":"2017-04-11T20:56:35.790Z",
   "properties":{
      "spring.application.name":"time-source",
      "spring.application.index":"0"
   }
}
```

# 14. Samples

For Spring Cloud Stream samples, please refer to the spring-cloud-stream-samples repository on GitHub.

## 14.1 Deploying Stream applications on CloudFoundry

On CloudFoundry services are usually exposed via a special environment variable called VCAP_SERVICES.

When configuring your binder connections, you can use the values from an environment variable as explained on the dataflow cloudfoundry server docs.

# Part II. Binder Implementations

# 15. Apache Kafka Binder

## 15.1 Usage

To use Apache Kafka binder all you need is to add `spring-cloud-stream-binder-kafka` as a dependency to your Spring Cloud Stream application. Below is a Maven example:

```
<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>
```

## 15.2 Apache Kafka Binder Overview

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

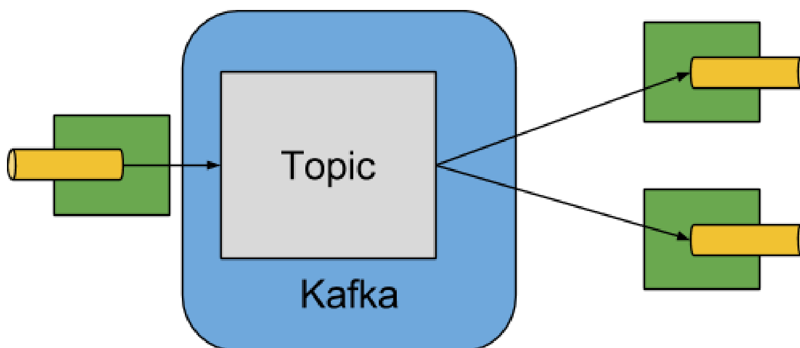


*Figure 15.1. Kafka Binder*

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.

The binder currently uses the Apache Kafka `kafka-clients` 1.0.0 jar and is designed to be used with a broker at least that version. This client can communicate with older brokers (refer to the Kafka documentation), but certain features may not be available. For example, with versions earlier than 0.11.x.x, native headers are not supported. Also, 0.11.x.x does not support the `autoAddPartitions` property.

## 15.3 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 documentation.

### 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.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, for example, security settings.

    Default: Empty map.

spring.cloud.stream.kafka.binder.headers
: The list of custom headers that will be transported by the binder. Only required when communicating with older applications (⇐ 1.3.x) with a `kafka-clients` version < 0.11.0.0; newer versions support headers natively.

    Default: empty.

spring.cloud.stream.kafka.binder.healthTimeout
: The time to wait to get partition information in seconds; default 60. Health will report as down if this timer expires.

    Default: 10.

spring.cloud.stream.kafka.binder.requiredAcks
: The number of required acks on the broker. See the Kafka documentation for the producer `acks` property.

    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. Can be overriden on each binding.

    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.transaction.transactionIdPrefix
: Enable transactions in the binder; see `transaction.id` in the Kafka documentation and Transactions in the `spring-kafka` documentation. When transactions are enabled, individual `producer` properties are ignored and all producers use the `spring.cloud.stream.kafka.binder.transaction.producer.*` properties.

  Default `null` (no transactions)

spring.cloud.stream.kafka.binder.transaction.producer.*
: Global producer properties for producers in a transactional binder. See `spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix` and the section called “Kafka Producer Properties” and the general producer properties supported by all binders.

  Default: See individual producer properties.

spring.cloud.stream.kafka.binder.headerMapperBeanName
: The bean name of a `KafkaHeaderMapper` used for mapping `spring-messaging` headers to/ from Kafka headers. Use this, for example, if you wish to customize the trusted packages in a `DefaultKafkaHeaderMapper`, which uses JSON deserialization for the headers.

  Default: none.

## Kafka Consumer Properties

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

admin.configuration
: A `Map` of Kafka topic properties used when provisioning topics. e.g. `spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format`

  Default: none.

admin.replicas-assignment
: A Map<Integer, List<Integer>> of replica assignments, with the key being the partition and value the assignments. Used when provisioning new topics. See `NewTopic` javadocs in the `kafka-clients` jar.

  Default: none.

admin.replication-factor
: The replication factor to use when provisioning topics; overrides the binder-wide setting. Ignored if `replicas-assignments` is present.

  Default: none (the binder-wide default of 1 is used).

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`.

ackEachRecord
: When `autoCommitOffset` is `true`, whether to commit the offset after each record is processed. By default, offsets are committed after all records in the batch of records returned by `consumer.poll()` have been processed. The number of records returned by a poll can be controlled with the `max.poll.recods` Kafka property, set via the consumer `configuration` property. Setting this to true may cause a degradation in performance, but reduces the likelihood of redelivered records when a failure occurs. Also see the binder `requiredAcks` property, which also affects the performance of committing offsets.

  Default: `false`.

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` and the application is responsible for acknowledging records. Also see `ackEachRecord`.

  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.

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

  Default: `false`.

startOffset
: The starting offset for new groups. Allowed values: `earliest`, `latest`. If the consumer group is set explicitly for the consumer 'binding' (via `spring.cloud.stream.bindings.<channelName>.group`), then 'startOffset' is set to

`earliest`; otherwise it is set to `latest` for the `anonymous` consumer group. Also see `resetOffsets`.

Default: null (equivalent to `earliest`).

enableDlq
: When set to true, it will send enable DLQ behavior for the consumer. By default, messages that result in errors will be forwarded to a topic named `error.<destination>.<group>`. The DLQ topic name can be configurable via the property `dlqName`. 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. See Section 15.6, "Dead-Letter Topic Processing" processing for more information. Starting with *version 2.0*, messages sent to the DLQ topic are enhanced with the following headers: `x-original-topic`, `x-exception-message` and `x-exception-stacktrace` as `byte[]`.

  Default: `false`.

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

  Default: Empty map.

dlqName
: The name of the DLQ topic to receive the error messages.

  Default: null (If not specified, messages that result in errors will be forwarded to a topic named `error.<destination>.<group>`).

dlqProducerProperties
: Using this, dlq specific producer properties can be set. All the properties available through kafka producer properties can be set through this property.

  Default: Default Kafka producer properties.

standardHeaders
: Indicates which standard headers are populated by the inbound channel adapter. `none`, `id`, `timestamp` or `both`. Useful if using native deserialization and the first component to receive a message needs an `id` (such as an aggregator that is configured to use a JDBC message store).

  Default: `none`

converterBeanName
: The name of a bean that implements `RecordMessageConverter`; used in the inbound channel adapter to replace the default `MessagingMessageConverter`.

  Default: `null`

idleEventInterval
: The interval, in milliseconds between events indicating that no messages have recently been received. Use an `ApplicationListener<ListenerContainerIdleEvent>` to receive these events. See the section called "Example: Pausing and Resuming the Consumer" for a usage example.

  Default: `30000`

## Kafka Producer Properties

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

admin.configuration
: A `Map` of Kafka topic properties used when provisioning new topics. e.g. `spring.cloud.stream.kafka.bindings.input.consumer.admin.configuration.message.format`

  Default: none.

admin.replicas-assignment
: A Map<Integer, List<Integer>> of replica assignments, with the key being the partition and value the assignments. Used when provisioning new topics. See `NewTopic` javadocs in the `kafka-clients` jar.

  Default: none.

admin.replication-factor
: The replication factor to use when provisioning new topics; overrides the binder-wide setting. Ignored if `replicas-assignments` is present.

  Default: none (the binder-wide default of 1 is used).

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`.

messageKeyExpression
: A SpEL expression evaluated against the outgoing message used to populate the key of the produced Kafka message. For example `headers['myKey']`; the payload cannot be used because by the time this expression is evaluated, the payload is already in the form of a `byte[]`.

  Default: `none`.

headerPatterns
: A comma-delimited list of simple patterns to match spring-messaging headers to be mapped to the kafka `Headers` in the `ProducerRecord`. Patterns can begin or end with the wildcard character (asterisk). Patterns can be negated by prefixing with `!`; matching stops after the first match (positive

or negative). For example `!foo,fo*` will pass `fox` but not `foo`. `id` and `timestamp` are never mapped.

Default: `*` (all headers - except the `id` and `timestamp`)

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.

## 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.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.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.

**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.

### Example: Pausing and Resuming the Consumer

If you wish to suspend consumption, but not cause a partition rebalance, you can pause and resume the consumer. This is facilitated by adding the `Consumer` as a parameter to your `@StreamListener`. To resume, you need an `ApplicationListener` for `ListenerContainerIdleEvent` s; the frequency at which events are published is controlled by the `idleEventInterval` property. Since the consumer is not thread-safe, you must call these methods on the calling thread.

The following simple application shows how to pause and resume.

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

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

 @StreamListener(Sink.INPUT)
 public void in(String in, @Header(KafkaHeaders.CONSUMER) Consumer<?, ?> consumer) {
  System.out.println(in);
  consumer.pause(Collections.singleton(new TopicPartition("myTopic", 0)));
 }

 @Bean
 public ApplicationListener<ListenerContainerIdleEvent> idleListener() {
  return event -> {
   System.out.println(event);
   if (event.getConsumer().paused().size() > 0) {
    event.getConsumer().resume(event.getConsumer().paused());
   }
  };
 }

}
```

## 15.4 Error Channels

Starting with *version 1.3*, the binder unconditionally sends exceptions to an error channel for each consumer destination, and can be configured to send async producer send failures to an error channel too. See the section called "Message Channel Binders and Error Channels" for more information.

The payload of the `ErrorMessage` for a send failure is a `KafkaSendFailureException` with properties:

- `failedMessage` - the spring-messaging `Message<?>` that failed to be sent.
- `record` - the raw `ProducerRecord` that was created from the `failedMessage`

There is no automatic handling of producer exceptions (such as sending to a Dead-Letter queue); you can consume these exceptions with your own Spring Integration flow.

## 15.5 Kafka Metrics

Kafka binder module exposes the following metrics:

`spring.cloud.stream.binder.kafka.someGroup.someTopic.lag` - this metric indicates how many messages have not been yet consumed from given binder's topic by given consumer group. For example if the value of the metric `spring.cloud.stream.binder.kafka.myGroup.myTopic.lag` is `1000`, then consumer group `myGroup` has `1000` messages to waiting to be consumed from topic `myTopic`. This metric is particularly useful to provide auto-scaling feedback to PaaS platform of your choice.

## 15.6 Dead-Letter Topic Processing

Because it can't be anticipated how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original topic. However, if the problem is a permanent issue, that could cause an infinite loop. The following `spring-boot` application is an example of how to route those messages back to the original topic, but moves them to a third "parking lot" topic after three attempts. The application is simply another spring-cloud-stream application that reads from the dead-letter topic. It terminates when no messages are received for 5 seconds.

The examples assume the original destination is `so8400out` and the consumer group is `so8400`.

There are several considerations.

- Consider only running the rerouting when the main application is not running. Otherwise, the retries for transient errors will be used up very quickly.
- Alternatively, use a two-stage approach - use this application to route to a third topic, and another to route from there back to the main topic.
- Since this technique uses a message header to keep track of retries, it won't work with `headerMode=raw`. In that case, consider adding some data to the payload (that can be ignored by the main application).
- `x-retries` has to be added to the `headers` property `spring.cloud.stream.kafka.binder.headers=x-retries` on both this, and the main application so that the header is transported between the applications.

- Since kafka is publish/subscribe, replayed messages will be sent to each consumer group, even those that successfully processed a message the first time around.

**application.properties.**

```
spring.cloud.stream.bindings.input.group=so8400replay
spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out
spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot
spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries
```

**Application.**

```
@SpringBootApplication
@EnableBinding(TwoOutputProcessor.class)
public class ReRouteDlqKApplication implements CommandLineRunner {

    private static final String X_RETRIES_HEADER = "x-retries";

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

    private final AtomicInteger processed = new AtomicInteger();

    @Autowired
    private MessageChannel parkingLot;

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Message<?> reRoute(Message<?> failed) {
        processed.incrementAndGet();
        Integer retries = failed.getHeaders().get(X_RETRIES_HEADER, Integer.class);
        if (retries == null) {
            System.out.println("First retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else if (retries.intValue() < 3) {
            System.out.println("Another retry for " + failed);
            return MessageBuilder.fromMessage(failed)
                    .setHeader(X_RETRIES_HEADER, new Integer(retries.intValue() + 1))
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build();
        }
        else {
            System.out.println("Retries exhausted for " + failed);
            parkingLot.send(MessageBuilder.fromMessage(failed)
                    .setHeader(BinderHeaders.PARTITION_OVERRIDE,
                            failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_ID))
                    .build());
        }
        return null;
    }

    @Override
    public void run(String... args) throws Exception {
        while (true) {
```

```
            int count = this.processed.get();
            Thread.sleep(5000);
            if (count == this.processed.get()) {
                System.out.println("Idle, terminating");
                return;
            }
        }
    }

    public interface TwoOutputProcessor extends Processor {

        @Output("parkingLot")
        MessageChannel parkingLot();

    }

}
```

## 15.7 Partitioning with the Kafka Binder

Apache Kafka supports topic partitioning natively.

Sometimes it is advantageous to send data to specific partitions, for example when you want to strictly order message processing - all messages for a particular customer should go to the same partition.

The following illustrates how to configure the producer and consumer side:

```
@SpringBootApplication
@EnableBinding(Source.class)
public class KafkaPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
            "foo1", "bar1", "qux1",
            "foo2", "bar2", "qux2",
            "foo3", "bar3", "qux3",
            "foo4", "bar4", "qux4",
            };

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
                .setHeader("partitionKey", value)
                .build();
    }

}
```

**application.yml.**

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: partitioned.topic
          producer:
            partitioned: true
```

```
          partition-key-expression: headers['partitionKey']
          partition-count: 12
```

### Important

The topic must be provisioned to have enough partitions to achieve the desired concurrency for all consumer groups. The above configuration will support up to 12 consumer instances (or 6 if their `concurrency` is 2, etc.). It is generally best to "over provision" the partitions to allow for future increases in consumers and/or concurrency.

### Note

The above configuration uses the default partitioning (`key.hashCode() % partitionCount`). This may or may not provide a suitably balanced algorithm, depending on the key values; you can override this default by using the `partitionSelectorExpression` or `partitionSelectorClass` properties.

Since partitions are natively handled by Kafka, no special configuration is needed on the consumer side. Kafka will allocate partitions across the instances.

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

    public static void main(String[] args) {
        new SpringApplicationBuilder(KafkaPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
        System.out.println(in + " received from partition " + partition);
    }

}
```

**application.yml.**

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: partitioned.topic
          group: myGroup
```

You can add instances as needed; Kafka will rebalance the partition allocations. If the instance count (or `instance count * concurrency`) exceeds the number of partitions, some consumers will be idle.

# 16. Apache Kafka Streams Binder

## 16.1 Usage

For using the Kafka Streams 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-streams</artifactId>
</dependency>
```

## 16.2 Kafka Streams Binder Overview

Spring Cloud Stream's Apache Kafka support also includes a binder implementation designed explicitly for Apache Kafka Streams binding. With this native integration, a Spring Cloud Stream "processor" application can directly use the Apache Kafka Streams APIs in the core business logic.

Kafka Streams binder implementation builds on the foundation provided by the Kafka Streams in Spring Kafka project.

As part of this native integration, the high-level Streams DSL provided by the Kafka Streams API is available for use in the business logic, too.

An early version of the Processor API support is available as well.

As noted early-on, Kafka Streams support in Spring Cloud Stream strictly only available for use in the Processor model. A model in which the messages read from an inbound topic, business processing can be applied, and the transformed messages can be written to an outbound topic. It can also be used in Processor applications with a no-outbound destination.

### Streams DSL

This application consumes data from a Kafka topic (e.g., `words`), computes word count for each unique word in a 5 seconds time window, and the computed results are sent to a downstream topic (e.g., `counts`) for further processing.

```
@SpringBootApplication
@EnableBinding(KStreamProcessor.class)
public class WordCountProcessorApplication {

 @StreamListener("input")
 @SendTo("output")
 public KStream<?, WordCount> process(KStream<?, String> input) {
  return input
               .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
               .groupBy((key, value) -> value)
               .windowedBy(TimeWindows.of(5000))
               .count(Materialized.as("WordCounts-multi"))
               .toStream()
               .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new
 Date(key.window().start()), new Date(key.window().end()))));
    }

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

Once built as a uber-jar (e.g., `wordcount-processor.jar`), you can run the above example like the following.

```
java -jar wordcount-processor.jar  --spring.cloud.stream.bindings.input.destination=words --
spring.cloud.stream.bindings.output.destination=counts
```

This application will consume messages from the Kafka topic `words` and the computed results are published to an output topic `counts`.

Spring Cloud Stream will ensure that the messages from both the incoming and outgoing topics are automatically bound as KStream objects. As a developer, you can exclusively focus on the business aspects of the code, i.e. writing the logic required in the processor. Setting up the Streams DSL specific configuration required by the Kafka Streams infrastructure is automatically handled by the framework.

## 16.3 Configuration Options

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

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

### Kafka Streams Properties

The following properties are available at the binder level and must be prefixed with `spring.cloud.stream.kafka.binder.` literal.

configuration
: Map with a key/value pair containing properties pertaining to Apache Kafka Streams API. This property must be prefixed with `spring.cloud.stream.kafka.streams.binder.`. Following are some examples of using this property.

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde=org.apache.kafka.common.serialization.Serdes
$StringSerde
spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde=org.apache.kafka.common.serialization.Serdes
$StringSerde
spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms=1000
```

For more information about all the properties that may go into streams configuration, see StreamsConfig JavaDocs in Apache Kafka Streams docs.

brokers
: Broker URL

  Default: `localhost`

zkNodes
: Zookeeper URL

  Default: `localhost`

serdeError
: Deserialization error handler type. Possible values are - `logAndContinue`, `logAndFail` or `sendToDlq`

  Default: `logAndFail`

applicationId
: Application ID for all the stream configurations in the current application context. You can override the application id for an individual `StreamListener` method using the `group` property on the binding. You have to ensure that you are using the same group name for all input bindings in the case of multiple inputs on the same methods.

  Default: `default`

The following properties are *only* available for Kafka Streams producers and must be prefixed with `spring.cloud.stream.kafka.streams.bindings.<binding name>.producer.` literal.

keySerde
: key serde to use

  Default: `none`.

valueSerde
: value serde to use

  Default: `none`.

useNativeEncoding
: flag to enable native encoding

  Default: `false`.

The following properties are *only* available for Kafka Streams consumers and must be prefixed with `spring.cloud.stream.kafka.streams.bindings.<binding name>.consumer.` literal.

keySerde
: key serde to use

  Default: `none`.

valueSerde
: value serde to use

  Default: `none`.

materializedAs
: state store to materialize when using incoming KTable types

  Default: `none`.

useNativeDecoding
: flag to enable native decoding

  Default: `false`.

dlqName
: DLQ topic name.

  Default: `none`.

### TimeWindow properties:

Windowing is an important concept in stream processing applications. Following properties are available to configure time-window computations.

spring.cloud.stream.kafka.streams.timeWindow.length
: When this property is given, you can autowire a `TimeWindows` bean into the application. The value is expressed in milliseconds.

  Default: `none`.

spring.cloud.stream.kstream.timeWindow.advanceBy
: Value is given in milliseconds.

  Default: `none`.

## 16.4 Multiple Input Bindings

For use cases that requires multiple incoming KStream objects or a combination of KStream and KTable objects, the Kafka Streams binder provides multiple bindings support.

Let's see it in action.

### Multiple Input Bindings as a Sink

```
@EnableBinding(KStreamKTableBinding.class)
.....
.....
@StreamListener
public void process(@Input("inputStream") KStream<String, PlayEvent> playEvents,
                    @Input("inputTable") KTable<Long, Song> songTable) {
                    ....
                    ....
}

interface KStreamKTableBinding {

    @Input("inputStream")
    KStream<?, ?> inputStream();

    @Input("inputTable")
    KTable<?, ?> inputTable();
}
```

In the above example, the application is written as a sink, i.e. there are no output bindings and the application has to decide concerning downstream processing. When you write applications in this style, you might want to send the information downstream or store them in a state store (See below for Queryable State Stores).

In the case of incoming KTable, if you want to materialize the computations to a state store, you have to express it through the following property.

```
spring.cloud.stream.kafka.streams.bindings.inputTable.consumer.materializedAs: all-songs
```

### Multiple Input Bindings as a Processor

```
@EnableBinding(KStreamKTableBinding.class)
....
....

@StreamListener
@SendTo("output")
public KStream<String, Long> process(@Input("input") KStream<String, Long> userClicksStream,
                                     @Input("inputTable") KTable<String, String> userRegionsTable) {
....
```

```
....
}

interface KStreamKTableBinding extends KafkaStreamsProcessor {

    @Input("inputX")
    KTable<?, ?> inputTable();
}
```

## 16.5 Multiple Output Bindings (aka Branching)

Kafka Streams allow outbound data to be split into multiple topics based on some predicates. The Kafka Streams binder provides support for this feature without compromising the programming model exposed through `StreamListener` in the end user application.

You can write the application in the usual way as demonstrated above in the word count example. However, when using the branching feature, you are required to do a few things. First, you need to make sure that your return type is `KStream[]` instead of a regular `KStream`. Second, you need to use the `SendTo` annotation containing the output bindings in the order (see example below). For each of these output bindings, you need to configure destination, content-type etc., complying with the standard Spring Cloud Stream expectations.

Here is an example:

```
@EnableBinding(KStreamProcessorWithBranches.class)
@EnableAutoConfiguration
public static class WordCountProcessorApplication {

    @Autowired
    private TimeWindows timeWindows;

    @StreamListener("input")
    @SendTo({"output1","output2","output3})
    public KStream<?, WordCount>[] process(KStream<Object, String> input) {

   Predicate<Object, WordCount> isEnglish = (k, v) -> v.word.equals("english");
   Predicate<Object, WordCount> isFrench =  (k, v) -> v.word.equals("french");
   Predicate<Object, WordCount> isSpanish = (k, v) -> v.word.equals("spanish");

   return input
     .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
     .groupBy((key, value) -> value)
     .windowedBy(timeWindows)
     .count(Materialized.as("WordCounts-1"))
     .toStream()
     .map((key, value) -> new KeyValue<>(null, new WordCount(key.key(), value, new
 Date(key.window().start()), new Date(key.window().end()))))
     .branch(isEnglish, isFrench, isSpanish);
    }

    interface KStreamProcessorWithBranches {

      @Input("input")
      KStream<?, ?> input();

      @Output("output1")
      KStream<?, ?> output1();

      @Output("output2")
      KStream<?, ?> output2();

      @Output("output3")
      KStream<?, ?> output3();
     }
}
```

Properties:

```
spring.cloud.stream.bindings.output1.contentType: application/json
spring.cloud.stream.bindings.output2.contentType: application/json
spring.cloud.stream.bindings.output3.contentType: application/json
spring.cloud.stream.kafka.streams.binder.configuration.commit.interval.ms: 1000
spring.cloud.stream.kafka.streams.binder.configuration:
  default.key.serde: org.apache.kafka.common.serialization.Serdes$StringSerde
  default.value.serde: org.apache.kafka.common.serialization.Serdes$StringSerde
spring.cloud.stream.bindings.output1:
  destination: foo
  producer:
    headerMode: raw
spring.cloud.stream.bindings.output2:
  destination: bar
  producer:
    headerMode: raw
spring.cloud.stream.bindings.output3:
  destination: fox
  producer:
    headerMode: raw
spring.cloud.stream.bindings.input:
  destination: words
  consumer:
    headerMode: raw
```

## 16.6 Message Conversion

Similar to message-channel based binder applications, the Kafka Streams binder adapts to the out-of-the-box content-type conversions without any compromise.

It is typical for Kafka Streams operations to know the type of SerDe's used to transform the key and value correctly. Therefore, it may be more natural to rely on the SerDe facilities provided by the Apache Kafka Streams library itself at the inbound and outbound conversions rather than using the content-type conversions offered by the framework. On the other hand, you might be already familiar with the content-type conversion patterns provided by the framework, and that, you'd like to continue using for inbound and outbound conversions.

Both the options are supported in the Kafka Streams binder implementation.

### Outbound serialization

If native encoding is disabled (which is the default), then the framework will convert the message using the contentType set by the user (otherwise, the default `application/json` will be applied). It will ignore any SerDe set on the outbound in this case for outbound serialization.

Here is the property to set the contentType on the outbound.

```
spring.cloud.stream.bindings.output.contentType: application/json
```

Here is the property to enable native encoding.

```
spring.cloud.stream.bindings.output.nativeEncoding: true
```

If native encoding is enabled on the output binding (user has to enable it as above explicitly), then the framework will skip any form of automatic message conversion on the outbound. In that case, it will switch to the Serde set by the user. The `valueSerde` property set on the actual output binding will be used. Here is an example.

```
spring.cloud.stream.kafka.streams.bindings.output.producer.valueSerde:
 org.apache.kafka.common.serialization.Serdes$StringSerde
```

If this property is not set, then it will use the "default" SerDe: `spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde`.

It is worth to mention that Kafka Streams binder does not serialize the keys on outbound - it simply relies on Kafka itself. Therefore, you either have to specify the `keySerde` property on the binding or it will default to the application-wide common `keySerde`.

Binding level key serde:

```
spring.cloud.stream.kafka.streams.bindings.output.producer.keySerde
```

Common Key serde:

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde
```

If branching is used, then you need to use multiple output bindings. For example,

```
interface KStreamProcessorWithBranches {

      @Input("input")
      KStream<?, ?> input();

      @Output("output1")
      KStream<?, ?> output1();

      @Output("output2")
      KStream<?, ?> output2();

      @Output("output3")
      KStream<?, ?> output3();
     }
```

If `nativeEncoding` is set, then you can set different SerDe's on individual output bindings as below.

```
spring.cloud.stream.kstream.bindings.output1.producer.valueSerde=IntegerSerde
spring.cloud.stream.kstream.bindings.output2.producer.valueSerde=StringSerde
spring.cloud.stream.kstream.bindings.output3.producer.valueSerde=JsonSerde
```

Then if you have `SendTo` like this, @SendTo({"output1", "output2", "output3"}), the `KStream[]` from the branches are applied with proper SerDe objects as defined above. If you are not enabling `nativeEncoding`, you can then set different contentType values on the output bindings as below. In that case, the framework will use the appropriate message converter to convert the messages before sending to Kafka.

```
spring.cloud.stream.bindings.output1.contentType: application/json
spring.cloud.stream.bindings.output2.contentType: application/java-serialzied-object
spring.cloud.stream.bindings.output3.contentType: application/octet-stream
```

## Inbound Deserialization

Similar rules apply to data deserialization on the inbound.

If native decoding is disabled (which is the default), then the framework will convert the message using the contentType set by the user (otherwise, the default `application/json` will be applied). It will ignore any SerDe set on the inbound in this case for inbound deserialization.

Here is the property to set the contentType on the inbound.

```
spring.cloud.stream.bindings.input.contentType: application/json
```

Here is the property to enable native decoding.

```
spring.cloud.stream.bindings.input.nativeDecoding: true
```

If native decoding is enabled on the input binding (user has to enable it as above explicitly), then the framework will skip doing any message conversion on the inbound. In that case, it will switch to the SerDe set by the user. The `valueSerde` property set on the actual output binding will be used. Here is an example.

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.valueSerde:
 org.apache.kafka.common.serialization.Serdes$StringSerde
```

If this property is not set, it will use the default SerDe: `spring.cloud.stream.kafka.streams.binder.configuration.default.value.serde`.

It is worth to mention that Kafka Streams binder does not deserialize the keys on inbound - it simply relies on Kafka itself. Therefore, you either have to specify the `keySerde` property on the binding or it will default to the application-wide common `keySerde`.

Binding level key serde:

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.keySerde
```

Common Key serde:

```
spring.cloud.stream.kafka.streams.binder.configuration.default.key.serde
```

As in the case of KStream branching on the outbound, the benefit of setting value SerDe per binding is that if you have multiple input bindings (multiple KStreams object) and they all require separate value SerDe's, then you can configure them individually. If you use the common configuration approach, then this feature won't be applicable.

## 16.7 Error Handling

Apache Kafka Streams provide the capability for natively handling exceptions from deserialization errors. For details on this support, please see this Out of the box, Apache Kafka Streams provide two kinds of deserialization exception handlers - `logAndContinue` and `logAndFail`. As the name indicates, the former will log the error and continue processing the next records and the latter will log the error and fail. `LogAndFail` is the default deserialization exception handler.

### Handling Deserialization Exceptions

Kafka Streams binder supports a selection of exception handlers through the following properties.

```
spring.cloud.stream.kafka.streams.binder.serdeError: logAndContinue
```

In addition to the above two deserialization exception handlers, the binder also provides a third one for sending the erroneous records (poison pills) to a DLQ topic. Here is how you enable this DLQ exception handler.

```
spring.cloud.stream.kafka.streams.binder.serdeError: sendToDlq
```

When the above property is set, all the deserialization error records are automatically sent to the DLQ topic.

```
spring.cloud.stream.kafka.streams.bindings.input.consumer.dlqName: foo-dlq
```

If this is set, then the error records are sent to the topic `foo-dlq`. If this is not set, then it will create a DLQ topic with the name `error.<input-topic-name>.<group-name>`.

A couple of things to keep in mind when using the exception handling feature in Kafka Streams binder.

- The property `spring.cloud.stream.kafka.streams.binder.serdeError` is applicable for the entire application. This implies that if there are multiple `StreamListener` methods in the same application, this property is applied to all of them.
- The exception handling for deserialization works consistently with native deserialization and framework provided message conversion.

## Handling Non-Deserialization Exceptions

For general error handling in Kafka Streams binder, it is up to the end user applications to handle application level errors. As a side effect of providing a DLQ for deserialization exception handlers, Kafka Streams binder provides a way to get access to the DLQ sending bean directly from your application. Once you get access to that bean, you can programmatically send any exception records from your application to the DLQ.

It continues to remain hard to robust error handling using the high-level DSL; Kafka Streams doesn't natively support error handling yet.

However, when you use the low-level Processor API in your application, there are options to control this behavior. See below.

```
@Autowired
private SendToDlqAndContinue dlqHandler;

@StreamListener("input")
@SendTo("output")
public KStream<?, WordCount> process(KStream<Object, String> input) {

    input.process(() -> new Processor() {
       ProcessorContext context;

       @Override
       public void init(ProcessorContext context) {
        this.context = context;
       }

       @Override
       public void process(Object o, Object o2) {

           try {
               .....
               .....
           }
           catch(Exception e) {
               //explicitly provide the kafka topic corresponding to the input binding as the first
 argument.
                        //DLQ handler will correctly map to the dlq topic from the actual incoming
 destination.
                        dlqHandler.sendToDlq("topic-name", (byte[]) o1, (byte[]) o2,
 context.partition());
           }
       }

       .....
       .....
    });
}
```

## 16.8 Interactive Queries

As part of the public Kafka Streams binder API, we expose a class called `QueryableStoreRegistry`. You can access this as a Spring bean in your application. An easy way to get access to this bean from your application is to "autowire" the bean in your application.

```
@Autowired
private QueryableStoreRegistry queryableStoreRegistry;
```

Once you gain access to this bean, then you can query for the particular state-store that you are interested. See below.

```
ReadOnlyKeyValueStore<Object, Object> keyValueStore =
      queryableStoreRegistry.getQueryableStoreType("my-store", QueryableStoreTypes.keyValueStore());
```

# 17. RabbitMQ Binder

## 17.1 Usage

For using the RabbitMQ 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-rabbit</artifactId>
</dependency>
```

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

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

## 17.2 RabbitMQ Binder Overview

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

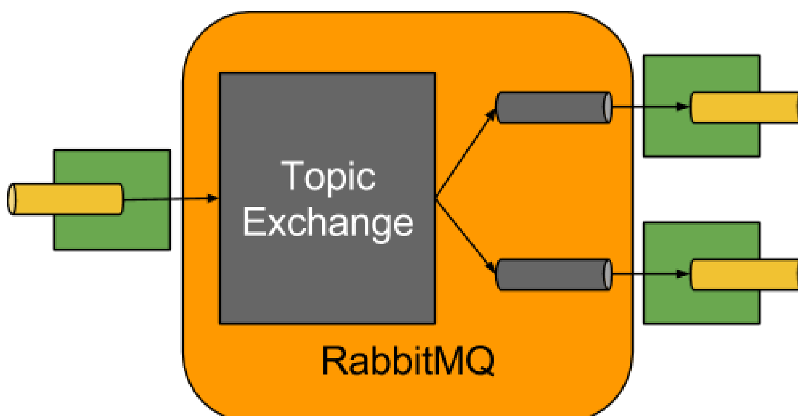


*Figure 17.1. RabbitMQ Binder*

The RabbitMQ Binder implementation maps each destination to a `TopicExchange` (by default). For each consumer group, a `Queue` will be bound to that `TopicExchange`. Each consumer instance have a corresponding RabbitMQ `Consumer` instance for its group's `Queue`. For partitioned producers/consumers the queues are suffixed with the partition index and use the partition index as routing key. For anonymous consumers (no `group` property) an auto-delete queue is used, with a randomized unique name.

Using the optional `autoBindDlq` option, you can configure the binder to create and configure dead-letter queues (DLQs) (and a dead-letter exchange `DLX` as well as routing infrastructure). By default, the dead letter queue has the name of the destination, appended with `.dlq`. If retry is enabled (`maxAttempts > 1`) failed messages will be delivered to the DLQ after retries are exhausted. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (default) so that a failed message will be routed to the DLQ, instead of being requeued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it); this enables additional information to be added to the message in headers, such as the stack trace in the `x-exception-`

stacktrace header. This option does not need retry enabled; you can republish a failed message after just one attempt. Starting with *version 1.2*, you can configure the delivery mode of republished messages; see property republishDeliveryMode.

**Important**

Setting requeueRejected to true (with republishToDlq=false ) will cause the message to be requeued and redelivered continually, which is likely not what you want unless the reason for the failure is transient. In general, it's better to enable retry within the binder by setting maxAttempts to greater than one, or set republishToDlq to true.

See the section called "RabbitMQ Binder Properties" for more information about these properties.

The framework does not provide any standard mechanism to consume dead-letter messages (or to re-route them back to the primary queue). Some options are described in Section 17.6, "Dead-Letter Queue Processing".

**Note**

When **multiple** RabbitMQ binders are used in a Spring Cloud Stream application, it is important to disable 'RabbitAutoConfiguration' to avoid the same configuration from RabbitAutoConfiguration being applied to the two binders. Exclude the class using the @SpringBootApplication annotation.

Starting with *version 2.0*, the RabbitMessageChannelBinder sets the RabbitTemplate.userPublisherConnection property to true so that the non-transactional producers will avoid dead locks on consumers which can happen if cached connections are blocked because of Memory Alarm on Broker.

## 17.3 Configuration Options

This section contains settings specific to the RabbitMQ Binder and bound channels.

For general binding configuration options and properties, please refer to the Spring Cloud Stream core documentation.

### RabbitMQ Binder Properties

By default, the RabbitMQ binder uses Spring Boot's ConnectionFactory, and it therefore supports all Spring Boot configuration options for RabbitMQ. (For reference, consult the Spring Boot documentation). RabbitMQ configuration options use the spring.rabbitmq prefix.

In addition to Spring Boot options, the RabbitMQ binder supports the following properties:

spring.cloud.stream.rabbit.binder.adminAddresses
: A comma-separated list of RabbitMQ management plugin URLs. Only used when nodes contains more than one entry. Each entry in this list must have a corresponding entry in spring.rabbitmq.addresses. **Only needed if you are using a RabbitMQ cluster and wish to consume from the node that hosts the queue. See Queue Affinity and the LocalizedQueueConnectionFactory for more information.**

  Default: empty.

spring.cloud.stream.rabbit.binder.nodes
: A comma-separated list of RabbitMQ node names. When more than one entry, used to locate the server address where a queue is located. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`. **Only needed if you are using a RabbitMQ cluster and wish to consume from the node that hosts the queue. See Queue Affinity and the LocalizedQueueConnectionFactory for more information.**

    Default: empty.

spring.cloud.stream.rabbit.binder.compressionLevel
: Compression level for compressed bindings. See `java.util.zip.Deflater`.

    Default: `1` (BEST_LEVEL).

spring.cloud.stream.binder.connection-name-prefix
: A connection name prefix used to name the connection(s) created by this binder. The name will be this prefix followed by `#n`, where n increments each time a new connection is opened.

    Default: none (Spring AMQP default).

## RabbitMQ Consumer Properties

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

acknowledgeMode
: The acknowledge mode.

    Default: `AUTO`.

autoBindDlq
: Whether to automatically declare the DLQ and bind it to the binder DLX.

    Default: `false`.

bindingRoutingKey
: The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). for partitioned destinations `-<instanceIndex>` will be appended.

    Default: `#`.

bindQueue
: Whether to bind the queue to the destination exchange; set to `false` if you have set up your own infrastructure and have previously created/bound the queue.

    Default: `true`.

deadLetterQueueName
: name of the DLQ

    Default: `prefix+destination.dlq`

deadLetterExchange
: a DLX to assign to the queue; if autoBindDlq is true

Default: 'prefix+DLX'

deadLetterRoutingKey
: a dead letter routing key to assign to the queue; if autoBindDlq is true

  Default: `destination`

declareExchange
: Whether to declare the exchange for the destination.

  Default: `true`.

delayedExchange
: Whether to declare the exchange as a `Delayed Message Exchange` - requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

  Default: `false`.

dlqDeadLetterExchange
: if a DLQ is declared, a DLX to assign to that queue

  Default: `none`

dlqDeadLetterRoutingKey
: if a DLQ is declared, a dead letter routing key to assign to that queue; default none

  Default: `none`

dlqExpires
: how long before an unused dead letter queue is deleted (ms)

  Default: `no expiration`

dlqLazy
: Declare the dead letter queue with the `x-queue-mode=lazy` argument. See [Lazy Queues](). Consider using a policy instead of this setting because using a policy allows changing the setting without deleting the queue.

  Default: `false`.

dlqMaxLength
: maximum number of messages in the dead letter queue

  Default: `no limit`

dlqMaxLengthBytes
: maximum number of total bytes in the dead letter queue from all messages

  Default: `no limit`

dlqMaxPriority
: maximum priority of messages in the dead letter queue (0-255)

  Default: `none`

dlqTtl
: default time to live to apply to the dead letter queue when declared (ms)

  Default: `no limit`

durableSubscription
: Whether subscription should be durable. Only effective if `group` is also set.

  Default: `true`.

exchangeAutoDelete
: If `declareExchange` is true, whether the exchange should be auto-delete (removed after the last queue is removed).

  Default: `true`.

exchangeDurable
: If `declareExchange` is true, whether the exchange should be durable (survives broker restart).

  Default: `true`.

exchangeType
: The exchange type; `direct`, `fanout` or `topic` for non-partitioned destinations; `direct` or `topic` for partitioned destinations.

  Default: `topic`.

exclusive
: Create an exclusive consumer; concurrency should be 1 when this is `true`; often used when strict ordering is required but enabling a hot standby instance to take over after a failure. See `recoveryInterval`, which controls how often a standby instance will attempt to consume.

  Default: `false`.

expires
: how long before an unused queue is deleted (ms)

  Default: `no expiration`

failedDeclarationRetryInterval
: The interval (ms) between attempts to consume from a queue if it is missing.

  Default: 5000

headerPatterns
: Patterns for headers to be mapped from inbound messages.

  Default: `['*']` (all headers).

lazy
: Declare the queue with the `x-queue-mode=lazy` argument. See Lazy Queues. Consider using a policy instead of this setting because using a policy allows changing the setting without deleting the queue.

  Default: `false`.

maxConcurrency
: the maximum number of consumers

  Default: `1`.

maxLength
: maximum number of messages in the queue

  Default: `no limit`

maxLengthBytes
: maximum number of total bytes in the queue from all messages

  Default: `no limit`

maxPriority
: maximum priority of messages in the queue (0-255)

  Default: `none`

missingQueuesFatal
: If the queue cannot be found, treat the condition as fatal and stop the listener container. Defaults to `false` so that the container keeps trying to consume from the queue, for example when using a cluster and the node hosting a non HA queue is down.

  Default: `false`

prefetch
: Prefetch count.

  Default: `1`.

prefix
: A prefix to be added to the name of the `destination` and queues.

  Default: "".

queueDeclarationRetries
: The number of times to retry consuming from a queue if it is missing. Only relevant if `missingQueuesFatal` is `true`; otherwise the container keeps retrying indefinitely.

  Default: `3`

queueNameGroupOnly
: When true, consume from a queue with a name equal to the `group`; otherwise the queue name is `destination.group`. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue.

  Default: false.

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

  Default: `5000`.

requeueRejected
: Whether delivery failures should be requeued when retry is disabled or republishToDlq is false.

  Default: `false`.

republishDeliveryMode
: When `republishToDlq` is `true`, specify the delivery mode of the republished message.

  Default: `DeliveryMode.PERSISTENT`

republishToDlq
: By default, messages which fail after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, RabbitMQ will route the failed message (unchanged) to the DLQ. If set to `true`, the binder will republish failed messages to the DLQ with additional headers, including the exception message and stack trace from the cause of the final failure.

  Default: false

transacted
: Whether to use transacted channels.

  Default: `false`.

ttl
: default time to live to apply to the queue when declared (ms)

  Default: `no limit`

txSize
: The number of deliveries between acks.

  Default: `1`.

### Rabbit Producer Properties

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

autoBindDlq
: Whether to automatically declare the DLQ and bind it to the binder DLX.

  Default: `false`.

batchingEnabled
: Whether to enable message batching by producers. Messages are batched into one message according to the following properties. Refer to Batching for more information.

  Default: `false`.

batchSize
: The number of messages to buffer when batching is enabled.

  Default: `100`.

batchBufferLimit
: The maximum buffer size when batching is enabled.

```
Default: `10000`.
```

batchTimeout
: The batch timeout when batching is enabled.

```
Default: `5000`.
```

bindingRoutingKey
: The routing key with which to bind the queue to the exchange (if `bindQueue` is `true`). Only applies to non-partitioned destinations. Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `#`.

bindQueue
: Whether to bind the queue to the destination exchange; set to `false` if you have set up your own infrastructure and have previously created/bound the queue. Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `true`.

compress
: Whether data should be compressed when sent.

  Default: `false`.

deadLetterQueueName
: name of the DLQ Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `prefix+destination.dlq`

deadLetterExchange
: a DLX to assign to the queue; if autoBindDlq is true Only applies if `requiredGroups` are provided and then only to those groups.

  Default: 'prefix+DLX'

deadLetterRoutingKey
: a dead letter routing key to assign to the queue; if autoBindDlq is true Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `destination`

declareExchange
: Whether to declare the exchange for the destination.

  Default: `true`.

delayExpression
: A SpEL expression to evaluate the delay to apply to the message (`x-delay` header) - has no effect if the exchange is not a delayed message exchange.

Default: No `x-delay` header is set.

delayedExchange
: Whether to declare the exchange as a `Delayed Message Exchange` - requires the delayed message exchange plugin on the broker. The `x-delayed-type` argument is set to the `exchangeType`.

  Default: `false`.

deliveryMode
: Delivery mode.

  Default: `PERSISTENT`.

dlqDeadLetterExchange
: if a DLQ is declared, a DLX to assign to that queue Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `none`

dlqDeadLetterRoutingKey
: if a DLQ is declared, a dead letter routing key to assign to that queue; default none Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `none`

dlqExpires
: how long before an unused dead letter queue is deleted (ms) Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no expiration`

dlqLazy
: Declare the dead letter queue with the `x-queue-mode=lazy` argument. See [Lazy Queues](). Consider using a policy instead of this setting because using a policy allows changing the setting without deleting the queue. Only applies if `requiredGroups` are provided and then only to those groups.

dlqMaxLength
: maximum number of messages in the dead letter queue Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

dlqMaxLengthBytes
: maximum number of total bytes in the dead letter queue from all messages Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

dlqMaxPriority
: maximum priority of messages in the dead letter queue (0-255) Only applies if `requiredGroups` are provided and then only to those groups.

Default: `none`

dlqTtl
: default time to live to apply to the dead letter queue when declared (ms) Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

exchangeAutoDelete
: If `declareExchange` is true, whether the exchange should be auto-delete (removed after the last queue is removed).

  Default: `true`.

exchangeDurable
: If `declareExchange` is true, whether the exchange should be durable (survives broker restart).

  Default: `true`.

exchangeType
: The exchange type; `direct`, `fanout` or `topic` for non-partitioned destinations; `direct` or `topic` for partitioned destinations.

  Default: `topic`.

expires
: how long before an unused queue is deleted (ms) Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no expiration`

headerPatterns
: Patterns for headers to be mapped to outbound messages.

  Default: `['*']` (all headers).

lazy
: Declare the queue with the `x-queue-mode=lazy` argument. See Lazy Queues. Consider using a policy instead of this setting because using a policy allows changing the setting without deleting the queue. Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `false`.

maxLength
: maximum number of messages in the queue Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

maxLengthBytes
: maximum number of total bytes in the queue from all messages Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

maxPriority
: maximum priority of messages in the queue (0-255) Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `none`

prefix
: A prefix to be added to the name of the `destination` exchange.

  Default: "".

queueNameGroupOnly
: When true, consume from a queue with a name equal to the `group`; otherwise the queue name is `destination.group`. This is useful, for example, when using Spring Cloud Stream to consume from an existing RabbitMQ queue. Only applies if `requiredGroups` are provided and then only to those groups.

  Default: false.

routingKeyExpression
: A SpEL expression to determine the routing key to use when publishing messages. For a fixed routing key, use a literal expression, e.g. `routingKeyExpression='my.routingKey'` in a properties file, or `routingKeyExpression: '''my.routingKey'''` in a YAML file.

  Default: `destination` or `destination-<partition>` for partitioned destinations.

transacted
: Whether to use transacted channels.

  Default: `false`.

ttl
: default time to live to apply to the queue when declared (ms) Only applies if `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

**Note**

In the case of RabbitMQ, content type headers can be set by external applications. Spring Cloud Stream supports them as part of an extended internal protocol used for any type of transport (including transports, such as Kafka (prior to 0.11), that do not natively support headers).

## 17.4 Retry With the RabbitMQ Binder

### Overview

When retry is enabled within the binder, the listener container thread is suspended for any back off periods that are configured. This might be important when strict ordering is required with a single consumer but for other use cases it prevents other messages from being processed on that thread. An alternative to using binder retry is to set up dead lettering with time to live on the dead-letter queue (DLQ), as well as dead-letter configuration on the DLQ itself. See the section called “RabbitMQ Binder Properties” for more information about the properties discussed here. Example configuration to enable this feature:

- Set `autoBindDlq` to `true` - the binder will create a DLQ; you can optionally specify a name in `deadLetterQueueName`
- Set `dlqTtl` to the back off time you want to wait between redeliveries
- Set the `dlqDeadLetterExchange` to the default exchange - expired messages from the DLQ will be routed to the original queue since the default `deadLetterRoutingKey` is the queue name (`destination.group`) - setting to the default exchange is achieved by setting the property with no value, as is shown in the example below

To force a message to be dead-lettered, either throw an `AmqpRejectAndDontRequeueException`, or set `requeueRejected` to `true` (default) and throw any exception.

The loop will continue without end, which is fine for transient problems but you may want to give up after some number of attempts. Fortunately, RabbitMQ provides the `x-death` header which allows you to determine how many cycles have occurred.

To acknowledge a message after giving up, throw an `ImmediateAcknowledgeAmqpException`.

## Putting it All Together

```
---
spring.cloud.stream.bindings.input.destination=myDestination
spring.cloud.stream.bindings.input.group=consumerGroup
#disable binder retries
spring.cloud.stream.bindings.input.consumer.max-attempts=1
#dlx/dlq setup
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-ttl=5000
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-dead-letter-exchange=
---
```

This configuration creates an exchange `myDestination` with queue `myDestination.consumerGroup` bound to a topic exchange with a wildcard routing key `#`. It creates a DLQ bound to a direct exchange `DLX` with routing key `myDestination.consumerGroup`. When messages are rejected, they are routed to the DLQ. After 5 seconds, the message expires and is routed to the original queue using the queue name as the routing key.

**Spring Boot application.**

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

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

    @StreamListener(Sink.INPUT)
    public void listen(String in, @Header(name = "x-death", required = false) Map<?,?> death) {
        if (death != null && death.get("count").equals(3L)) {
            // giving up - don't send to DLX
            throw new ImmediateAcknowledgeAmqpException("Failed after 4 attempts");
        }
        throw new AmqpRejectAndDontRequeueException("failed");
    }

}
```

Notice that the count property in the `x-death` header is a `Long`.

## 17.5 Error Channels

Starting with *version 1.3*, the binder unconditionally sends exceptions to an error channel for each consumer destination, and can be configured to send async producer send failures to an error channel too. See the section called “Message Channel Binders and Error Channels” for more information.

With rabbitmq, there are two types of send failures:

- returned messages
- negatively acknowledged Publisher Confirms

The latter is rare; quoting the RabbitMQ documentation "[A nack] will only be delivered if an internal error occurs in the Erlang process responsible for a queue.".

As well as enabling producer error channels as described in the section called “Message Channel Binders and Error Channels”, the RabbitMQ binder will only send messages to the channels if the connection factory is appropriately configured:

- `ccf.setPublisherConfirms(true);`
- `ccf.setPublisherReturns(true);`

When using spring boot configuration for the connection factory, set properties:

- `spring.rabbitmq.publisher-confirms`
- `spring.rabbitmq.publisher-returns`

The payload of the `ErrorMessage` for a returned message is a `ReturnedAmqpMessageException` with properties:

- `failedMessage` - the spring-messaging `Message<?>` that failed to be sent.
- `amqpMessage` - the raw spring-amqp `Message`
- `replyCode` - an integer value indicating the reason for the failure (e.g. 312 - No route)
- `replyText` - a text value indicating the reason for the failure e.g. `NO_ROUTE`.
- `exchange` - the exchange to which the message was published.
- `routingKey` - the routing key used when the message was published.

For negatively acknowledged confirms, the payload is a `NackedAmqpMessageException` with properties:

- `failedMessage` - the spring-messaging `Message<?>` that failed to be sent.
- `nackReason` - a reason (if available; you may need to examine the broker logs for more information).

There is no automatic handling of these exceptions (such as sending to a Dead-Letter queue); you can consume these exceptions with your own Spring Integration flow.

## 17.6 Dead-Letter Queue Processing

Because it can’t be anticipated how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering

is transient, you may wish to route the messages back to the original queue. However, if the problem is a permanent issue, that could cause an infinite loop. The following `spring-boot` application is an example of how to route those messages back to the original queue, but moves them to a third "parking lot" queue after three attempts. The second example utilizes the [RabbitMQ Delayed Message Exchange] to introduce a delay to the requeued message. In this example, the delay increases for each attempt. These examples use a `@RabbitListener` to receive messages from the DLQ, you could also use `RabbitTemplate.receive()` in a batch process.

The examples assume the original destination is `so8400in` and the consumer group is `so8400`.

## Non-Partitioned Destinations

The first two examples are when the destination is **not** partitioned.

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class,
 args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Integer retriesHeader = (Integer)
 failedMessage.getMessageProperties().getHeaders().get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            failedMessage.getMessageProperties().getHeaders().put(X_RETRIES_HEADER, retriesHeader + 1);
            this.rabbitTemplate.send(ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE = "so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE + ".dlq";
```

```
    private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-retries";

    private static final String DELAY_EXCHANGE = "dlqReRouter";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class,
 args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            headers.put("x-delay", 5000 * retriesHeader);
            this.rabbitTemplate.send(DELAY_EXCHANGE, ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public DirectExchange delayExchange() {
        DirectExchange exchange = new DirectExchange(DELAY_EXCHANGE);
        exchange.setDelayed(true);
        return exchange;
    }

    @Bean
    public Binding bindOriginalToDelay() {
        return BindingBuilder.bind(new Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_QUEUE);
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

## Partitioned Destinations

With partitioned destinations, there is one DLQ for all partitions and we determine the original queue from the headers.

### republishToDlq=false

When `republishToDlq` is `false`, RabbitMQ publishes the message to the DLX/DLQ with an `x-death` header containing information about the original destination.

```
@SpringBootApplication
public class ReRouteDlqApplication {

 private static final String ORIGINAL_QUEUE = "so8400in.so8400";
```

```
 private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

 private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

 private static final String X_DEATH_HEADER = "x-death";

 private static final String X_RETRIES_HEADER = "x-retries";

 public static void main(String[] args) throws Exception {
  ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
  System.out.println("Hit enter to terminate");
  System.in.read();
  context.close();
 }

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @SuppressWarnings("unchecked")
 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X_DEATH_HEADER);
   String exchange = (String) xDeath.get(0).get("exchange");
   List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");
   this.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

 @Bean
 public Queue parkingLot() {
  return new Queue(PARKING_LOT);
 }

}
```

### republishToDlq=true

When `republishToDlq` is `true`, the republishing recoverer adds the original exchange and routing key to headers.

```
@SpringBootApplication
public class ReRouteDlqApplication {

 private static final String ORIGINAL_QUEUE = "so8400in.so8400";

 private static final String DLQ = ORIGINAL_QUEUE + ".dlq";

 private static final String PARKING_LOT = ORIGINAL_QUEUE + ".parkingLot";

 private static final String X_RETRIES_HEADER = "x-retries";

 private static final String X_ORIGINAL_EXCHANGE_HEADER = RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

 private static final String X_ORIGINAL_ROUTING_KEY_HEADER =
 RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

 public static void main(String[] args) throws Exception {
  ConfigurableApplicationContext context = SpringApplication.run(ReRouteDlqApplication.class, args);
  System.out.println("Hit enter to terminate");
  System.in.read();
```

```
  context.close();
 }

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   String exchange = (String) headers.get(X_ORIGINAL_EXCHANGE_HEADER);
   String originalRoutingKey = (String) headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
   this.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

 @Bean
 public Queue parkingLot() {
  return new Queue(PARKING_LOT);
 }

}
```

## 17.7 Partitioning with the RabbitMQ Binder

RabbitMQ does not support partitioning natively.

Sometimes it is advantageous to send data to specific partitions, for example when you want to strictly order message processing - all messages for a particular customer should go to the same partition.

The `RabbitMessageChannelBinder` provides partitioning by binding a queue for each partition to the destination exchange.

The following illustrates how to configure the producer and consumer side:

**Producer.**

```
@SpringBootApplication
@EnableBinding(Source.class)
public class RabbitPartitionProducerApplication {

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String[] data = new String[] {
            "foo1", "bar1", "qux1",
            "foo2", "bar2", "qux2",
            "foo3", "bar3", "qux3",
            "foo4", "bar4", "qux4",
            };

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionProducerApplication.class)
            .web(false)
            .run(args);
    }

    @InboundChannelAdapter(channel = Source.OUTPUT, poller = @Poller(fixedRate = "5000"))
    public Message<?> generate() {
        String value = data[RANDOM.nextInt(data.length)];
```

```
        System.out.println("Sending: " + value);
        return MessageBuilder.withPayload(value)
                .setHeader("partitionKey", value)
                .build();
    }

}
```

**application.yml.**

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: partitioned.destination
          producer:
            partitioned: true
            partition-key-expression: headers['partitionKey']
            partition-count: 2
            required-groups:
            - myGroup
```

**Note**

The above configuration uses the default partitioning (`key.hashCode() % partitionCount`). This may or may not provide a suitably balanced algorithm, depending on the key values; you can override this default by using the `partitionSelectorExpression` or `partitionSelectorClass` properties.

The `required-groups` property is only required if you need the consumer queues to be provisioned when the producer is deployed. Otherwise, any messages sent to a partition will be lost until the corresponding consumer is deployed.

This configuration provisions a topic exchange:

| partitioned.destination | topic | D |
|---|---|---|

and these queues bound to that exchange:

| partitioned.destination.myGroup-0 | D |
|---|---|
| partitioned.destination.myGroup-1 | D |

with these bindings:

Bindings

This exchange

| To | Routing key | Arguments | |
|---|---|---|---|
| partitioned.destination.myGroup-0 | partitioned.destination-0 | | Unbind |
| partitioned.destination.myGroup-1 | partitioned.destination-1 | | Unbind |

**Consumer.**

```java
@SpringBootApplication
@EnableBinding(Sink.class)
public class RabbitPartitionConsumerApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(RabbitPartitionConsumerApplication.class)
            .web(false)
            .run(args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(@Payload String in, @Header(AmqpHeaders.CONSUMER_QUEUE) String queue) {
        System.out.println(in + " received from queue " + queue);
    }

}
```

**application.yml.**

```yaml
    spring:
      cloud:
        stream:
          bindings:
            input:
              destination: partitioned.destination
              group: myGroup
              consumer:
                partitioned: true
                instance-index: 0
```

### Important

The `RabbitMessageChannelBinder` does not support dynamic scaling; there must be at least one consumer per partition. The consumer's `instanceIndex` is used to indicate which partition will be consumed. On platforms such as Cloud Foundry there can only be one instance with an `instanceIndex`.

# Part III. 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 for Redis, Rabbit, and Kafka bindings you should have those servers 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. See the README in the scripts demo repository for specific instructions about the common cases of mongo, rabbit and redis.

## 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).