Spring Cloud Stream Reference Guide

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, Jay Bryant

Table of Contents

Part I. Spring Cloud Stream Core

1. A Brief History of Spring’s Data Integration Journey

Spring’s journey on Data Integration started with Spring Integration. With its programming model, it provided a consistent developer experience to build applications that can embrace Enterprise Integration Patterns to connect with external systems such as, databases, message brokers, and among others.

Fast forward to the cloud-era, where microservices have become prominent in the enterprise setting. Spring Boot transformed the way how developers built Applications. With Spring’s programming model and the runtime responsibilities handled by Spring Boot, it became seamless to develop stand-alone, production-grade Spring-based microservices.

To extend this to Data Integration workloads, Spring Integration and Spring Boot were put together into a new project. Spring Cloud Stream was born.

With Spring Cloud Stream, developers can: * Build, test, iterate, and deploy data-centric applications in isolation. * Apply modern microservices architecture patterns, including composition through messaging. * Decouple application responsibilities with event-centric thinking. An event can represent something that has happened in time, to which the downstream consumer applications can react without knowing where it originated or the producer’s identity. * Port the business logic onto message brokers (such as RabbitMQ, Apache Kafka, Amazon Kinesis). * Interoperate between channel-based and non-channel-based application binding scenarios to support stateless and stateful computations by using Project Reactor’s Flux and Kafka Streams APIs. * Rely on the framework’s automatic content-type support for common use-cases. Extending to different data conversion types is possible.

2. Quick Start

You can try Spring Cloud Stream in less then 5 min even before you jump into any details by following this three-step guide.

We show you how to create a Spring Cloud Stream application that receives messages coming from the messaging middleware of your choice (more on this later) and logs received messages to the console. We call it `LoggingConsumer`. While not very practical, it provides a good introduction to some of the main concepts and abstractions, making it easier to digest the rest of this user guide.

The three steps are as follows:

1. Section 2.1, “Creating a Sample Application by Using Spring Initializr”
2. Section 2.2, “Importing the Project into Your IDE”
3. Section 2.3, “Adding a Message Handler, Building, and Running”

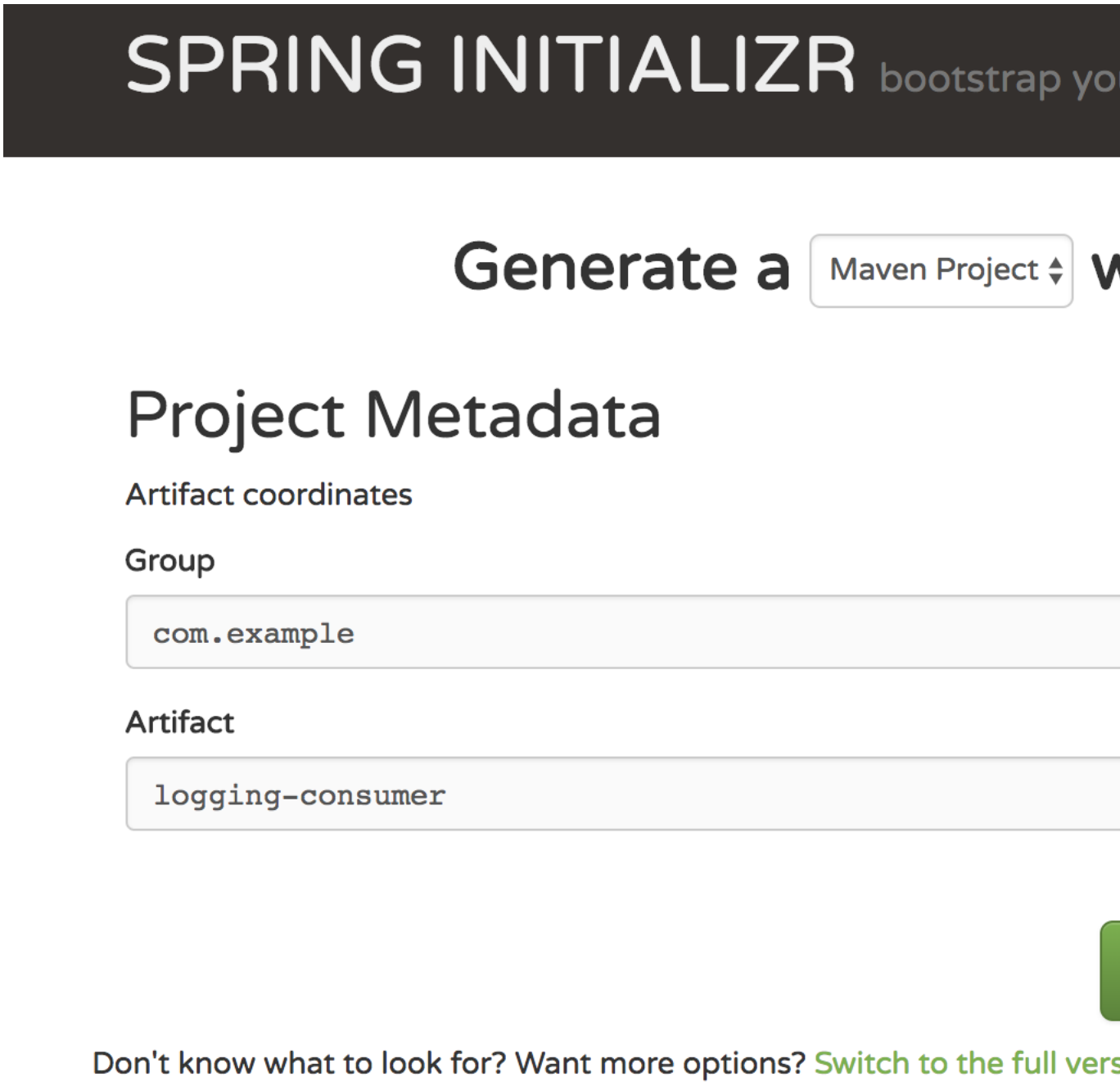
2.1 Creating a Sample Application by Using Spring Initializr

To get started, visit the Spring Initializr. From there, you can generate our `LoggingConsumer` application. To do so:

1. In the **Dependencies** section, start typing `stream`. When the “Cloud Stream” option should appears, select it.
2. Start typing either 'kafka' or 'rabbit'.
3. Select “Kafka” or “RabbitMQ”.

   Basically, you choose the messaging middleware to which your application binds. We recommend using the one you have already installed or feel more comfortable with installing and running. Also, as you can see from the Initilaizer screen, there are a few other options you can choose. For example, you can choose Gradle as your build tool instead of Maven (the default).
4. In the **Artifact** field, type 'logging-consumer'.

   The value of the **Artifact** field becomes the application name. If you chose RabbitMQ for the middleware, your Spring Initializr should now be as follows:



5. Click the **Generate Project** button.

   Doing so downloads the zipped version of the generated project to your hard drive.

6. Unzip the file into the folder you want to use as your project directory.

Tip

We encourage you to explore the many possibilities available in the Spring Initializr. It lets you create many different kinds of Spring applications.

2.2 Importing the Project into Your IDE

Now you can import the project into your IDE. Keep in mind that, depending on the IDE, you may need to follow a specific import procedure. For example, depending on how the project was generated (Maven or Gradle), you may need to follow specific import procedure (for example, in Eclipse or STS, you need to use File → Import → Maven → Existing Maven Project).

Once imported, the project must have no errors of any kind. Also, `src/main/java` should contain `com.example.loggingconsumer.LoggingConsumerApplication`.

Technically, at this point, you can run the application's main class. It is already a valid Spring Boot application. However, it does not do anything, so we want to add some code.

2.3 Adding a Message Handler, Building, and Running

Modify the `com.example.loggingconsumer.LoggingConsumerApplication` class 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 preceding listing:

- We have enabled `Sink` binding (input-no-output) by using `@EnableBinding(Sink.class)`. Doing so signals to the framework to initiate binding to the messaging middleware, where it automatically creates the destination (that is, queue, topic, and others) that are bound to the `Sink.INPUT` channel.
- We have added a `handler` method to receive incoming messages of type `Person`. Doing so lets you see one of the core features of the framework: It tries to automatically convert incoming message payloads to type `Person`.

You now have a fully functional Spring Cloud Stream application that does listens for messages. From here, for simplicity, we assume you selected RabbitMQ in step one. Assuming you have RabbitMQ installed and running, you can start the application by running its `main` method in your IDE.

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 the RabbitMQ management console or any other RabbitMQ client and send a message to `input.anonymous.CbMIwdkJSBO1ZoPDOtHtCg`. The `anonymous.CbMIwdkJSBO1ZoPDOtHtCg` part represents the group name and is generated, so it is bound to be different in your environment. For something more predictable, you can use an explicit group name by setting `spring.cloud.stream.bindings.input.group=hello` (or whatever name you like).

The contents of the message should be a JSON representation of the `Person` class, as follows:

```
{"name":"Sam Spade"}
```

Then, in your console, you should see:

```
Received: Sam Spade
```

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

Now you have a working (albeit very basic) Spring Cloud Stream application.

3. What's New in 2.0?

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

- Section 3.1, "New Features and Components"
- Section 3.2, "Notable Enhancements"

3.1 New Features and Components

- **Polling Consumers**: Introduction of polled consumers, which lets the application control message processing rates. See "the section called "Using Polled Consumers"" for more details. You can also read this blog post for more details.
- **Micrometer Support**: Metrics has been switched to use Micrometer. `MeterRegistry` is also provided as a bean so that custom applications can autowire it to capture custom metrics. See "Chapter 14, *Metrics Emitter*" for more details.
- **New Actuator Binding Controls**: New actuator binding controls let you both visualize and control the Bindings lifecycle. For more details, see Section 7.6, "Binding visualization and control".
- **Configurable RetryTemplate**: Aside from providing properties to configure `RetryTemplate`, we now let you provide your own template, effectively overriding the one provided by the framework. To use it, configure it as a `@Bean` in your application.

3.2 Notable Enhancements

This version includes the following notable enhancements:

- the section called "Both Actuator and Web Dependencies Are Now Optional"
- the section called "Content-type Negotiation Improvements"
- Section 3.3, "Notable Deprecations"

Both Actuator and Web Dependencies Are Now Optional

This change slims down the footprint of the deployed application in the event neither actuator nor web dependencies required. It also lets you switch between the reactive and conventional web paradigms by manually adding one of the following dependencies.

The following listing shows how to add the conventional web framework:

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

The following listing shows how to add the reactive web framework:

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

The following list shows how to add the actuator dependency:

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

Content-type Negotiation Improvements

One of the core themes for verion 2.0 is improvements (in both consistency and performance) around content-type negotiation and message conversion. The following summary outlines the notable changes and improvements in this area. See the "Chapter 9, *Content Type Negotiation*" section for more details. Also this blog post contains more detail.

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

3.3 Notable Deprecations

As of version 2.0, the following items have been deprecated:

- the section called "Java Serialization (Java Native and Kryo)"
- the section called "Deprecated Classes and Methods"

Java Serialization (Java Native and Kryo)

`JavaSerializationMessageConverter` and `KryoMessageConverter` remain for now. However, we plan to move them out of the core packages and support in the future. The main reason for this deprecation is to flag the issue that type-based, language-specific serialization could cause in distributed environments, where Producers and Consumers may depend on different JVM versions or have different versions of supporting libraries (that is, Kryo). We also wanted to draw the attention to the fact that Consumers and Producers may not even be Java-based, so polyglot style serialization (i.e., JSON) is better suited.

Deprecated Classes and Methods

The following is a quick summary of notable deprecations. See the corresponding {spring-cloud-stream-javadoc-current}[javadoc] for more details.

- `SharedChannelRegistry`. Use `SharedBindingTargetRegistry`.

- `Bindings`. Beans qualified by it are already uniquely identified by their type — for example, provided `Source`, `Processor`, or custom bindings:

```
public interface Sample {
 String OUTPUT = "sampleOutput";

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

- `HeaderMode.raw`. Use `none`, `headers` or `embeddedHeaders`
- `ProducerProperties.partitionKeyExtractorClass` in favor of `partitionKeyExtractorName` and `ProducerProperties.partitionSelectorClass` in favor of `partitionSelectorName`. This change ensures that both components are Spring configured and managed and are referenced in a Spring-friendly way.
- `BinderAwareRouterBeanPostProcessor`. While the component remains, it is no longer a `BeanPostProcessor` and will be renamed in the future.
- `BinderProperties.setEnvironment(Properties environment)`. Use `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.

4. 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 example shows a sink application that 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 output channels. Spring Cloud Stream provides the `Source`, `Sink`, and `Processor` interfaces. You can also define your own interfaces.

The following listing shows 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 is used.

Spring Cloud Stream creates an implementation of the interface for you. You can use this in the application by autowiring it, as shown in the following example (from 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());
  }
```

```
}
```

5. 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
- Section 5.2, "The Binder Abstraction"
- Persistent publish-subscribe support
- Consumer group support
- Partitioning support
- A pluggable Binder SPI

5.1 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.

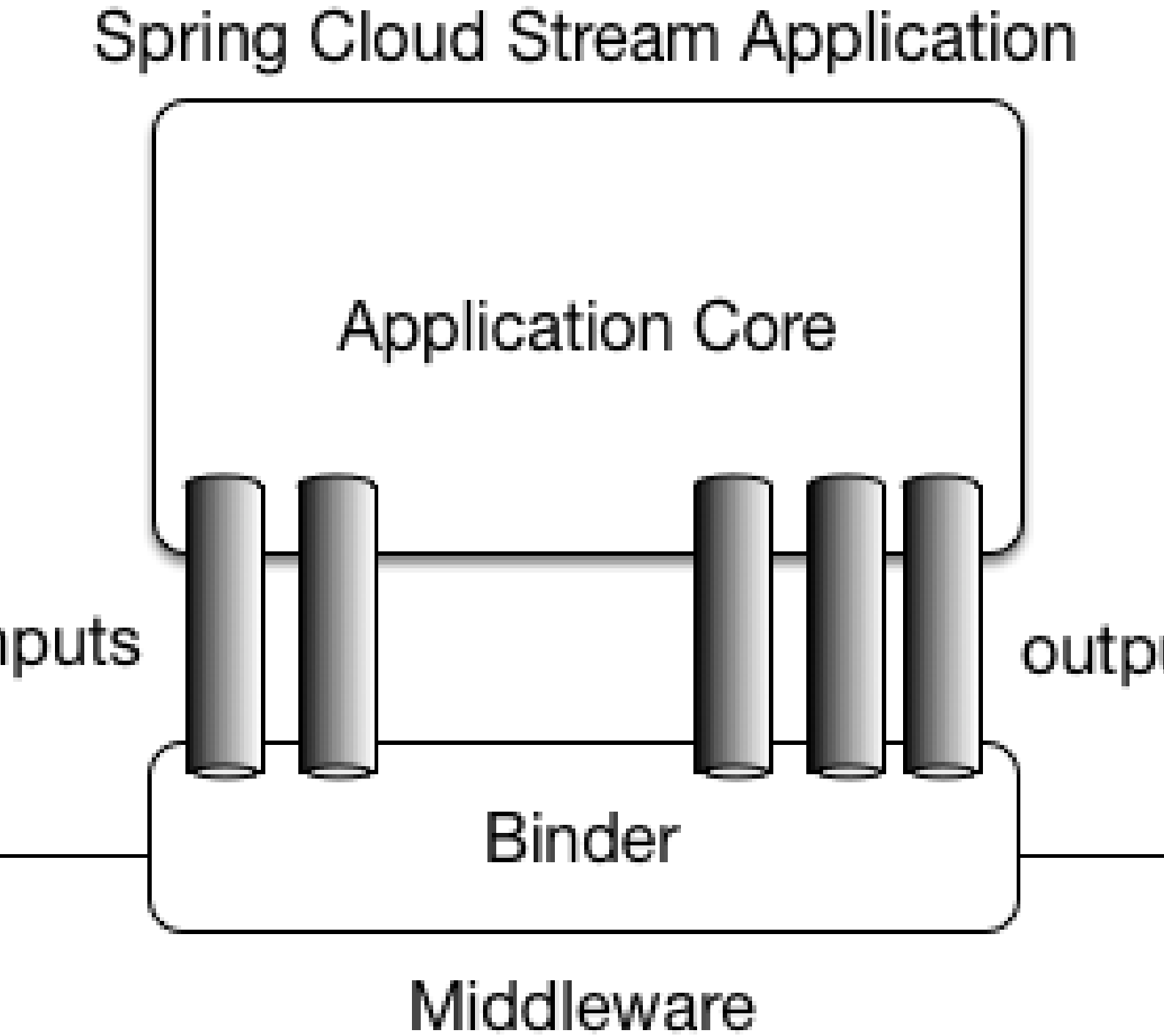


Figure 5.1. Spring Cloud Stream Application

Fat JAR

Spring Cloud Stream applications can be run in stand-alone 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. See the Spring Boot Reference Guide for more details.

5.2 The Binder Abstraction

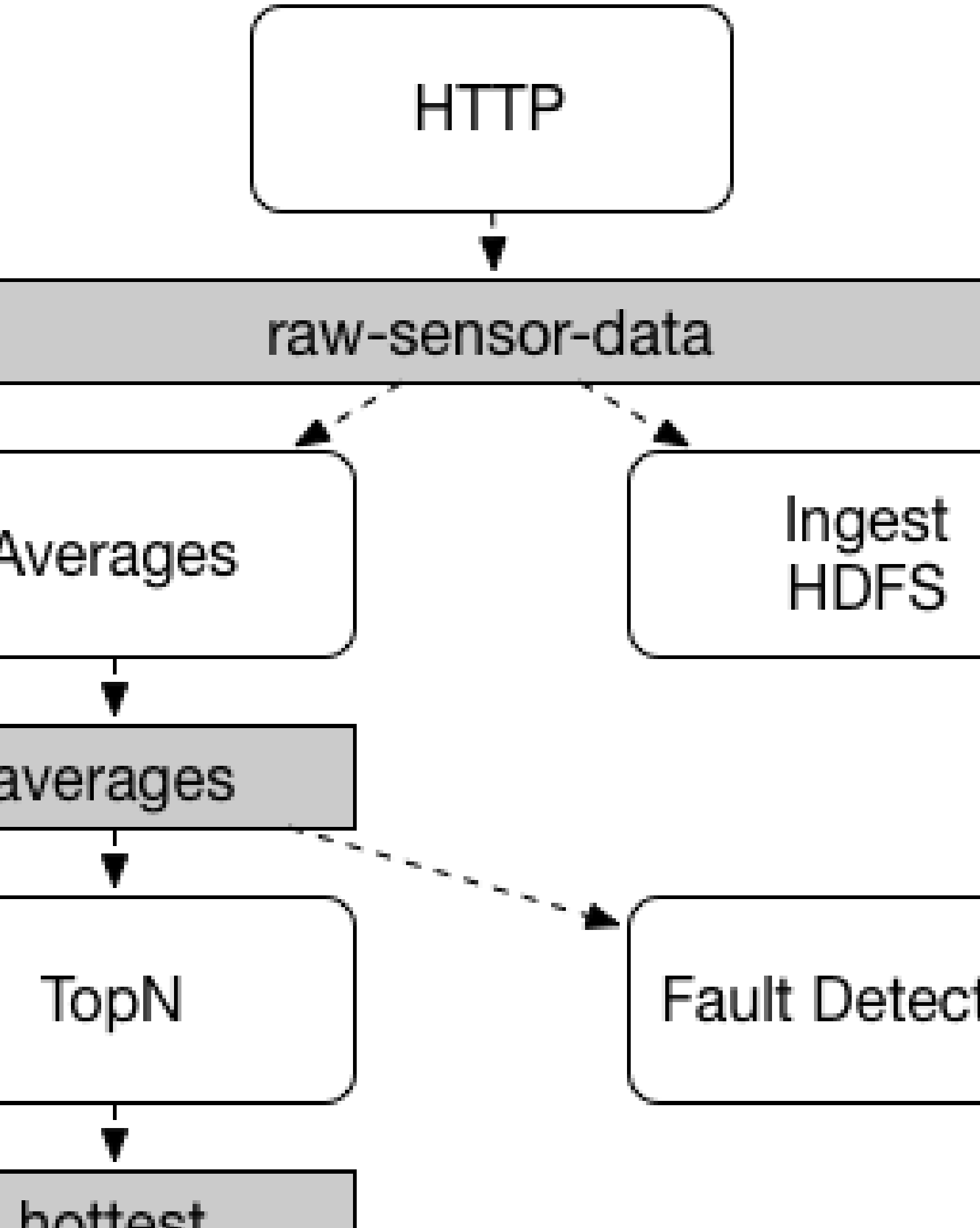
Spring Cloud Stream provides Binder implementations for Kafka and Rabbit MQ. Spring Cloud Stream also includes a TestSupportBinder, which leaves a channel unmodified so that tests can interact with channels directly and reliably assert on what is received. You can also use the extensible API to write 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 (such as 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 4, *Introducing Spring Cloud Stream* section, setting the `spring.cloud.stream.bindings.input.destination` application property to `raw-sensor-data` causes 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 use different types of middleware with the same code. To do so, 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.

5.3 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.


HTTP
raw-sensor-data
Averages
Ingest
HDFS
averages
TopN
Fault Detect

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 (Hadoop Distributed File System). 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 lets new applications 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.

5.4 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 so, 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`.

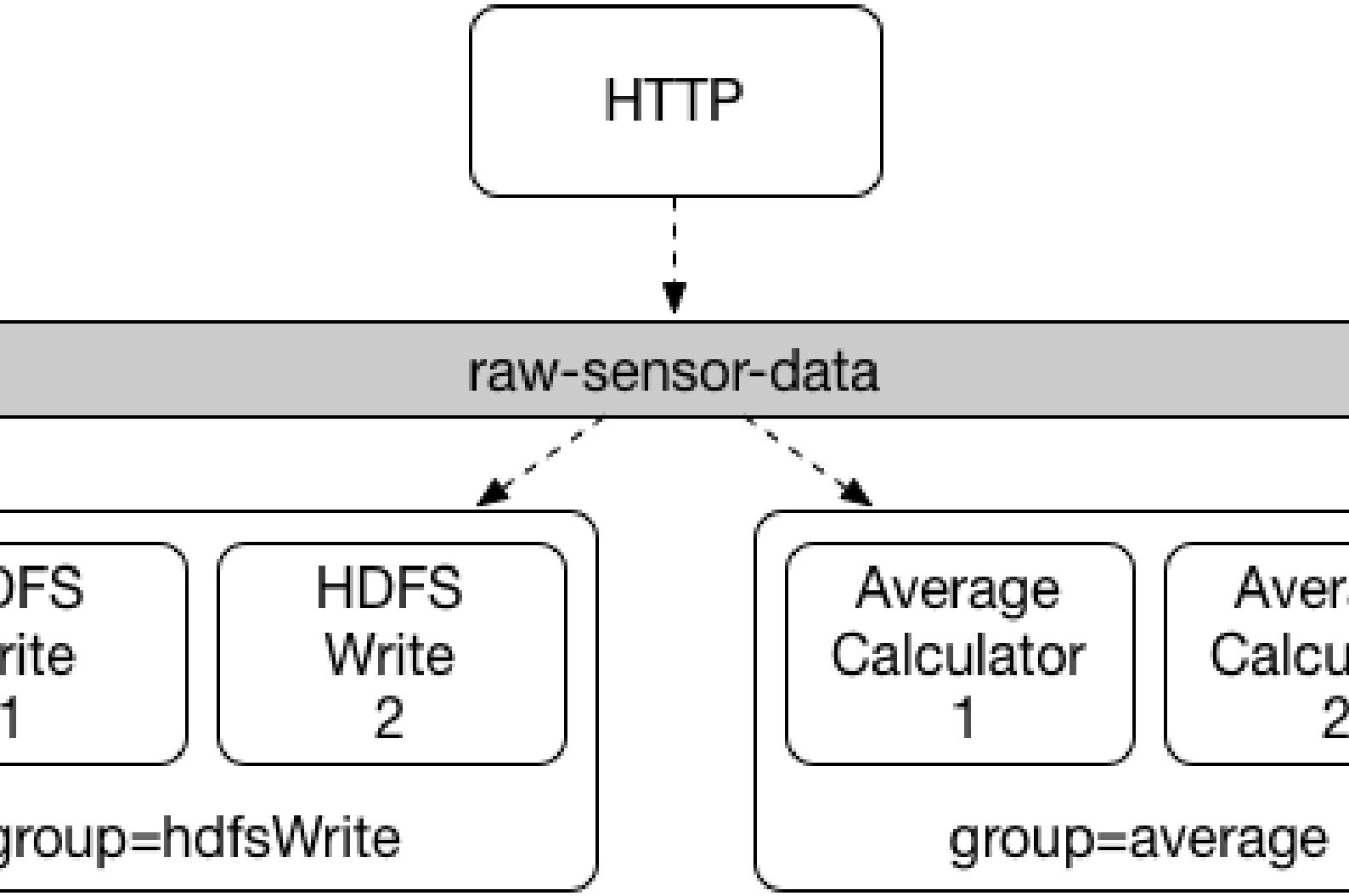


Figure 5.3. Spring Cloud Stream Consumer Groups

All groups that 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.

5.5 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. A message is delivered as soon as it is available and a thread is available to process it.

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

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 that, once at least one subscription for a group has been created, the group receives 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 (such as 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. Doing so prevents the application's instances from receiving duplicate messages (unless that behavior is desired, which is unusual).

5.6 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 (such as 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 (for example, Kafka) or not (for example, RabbitMQ).

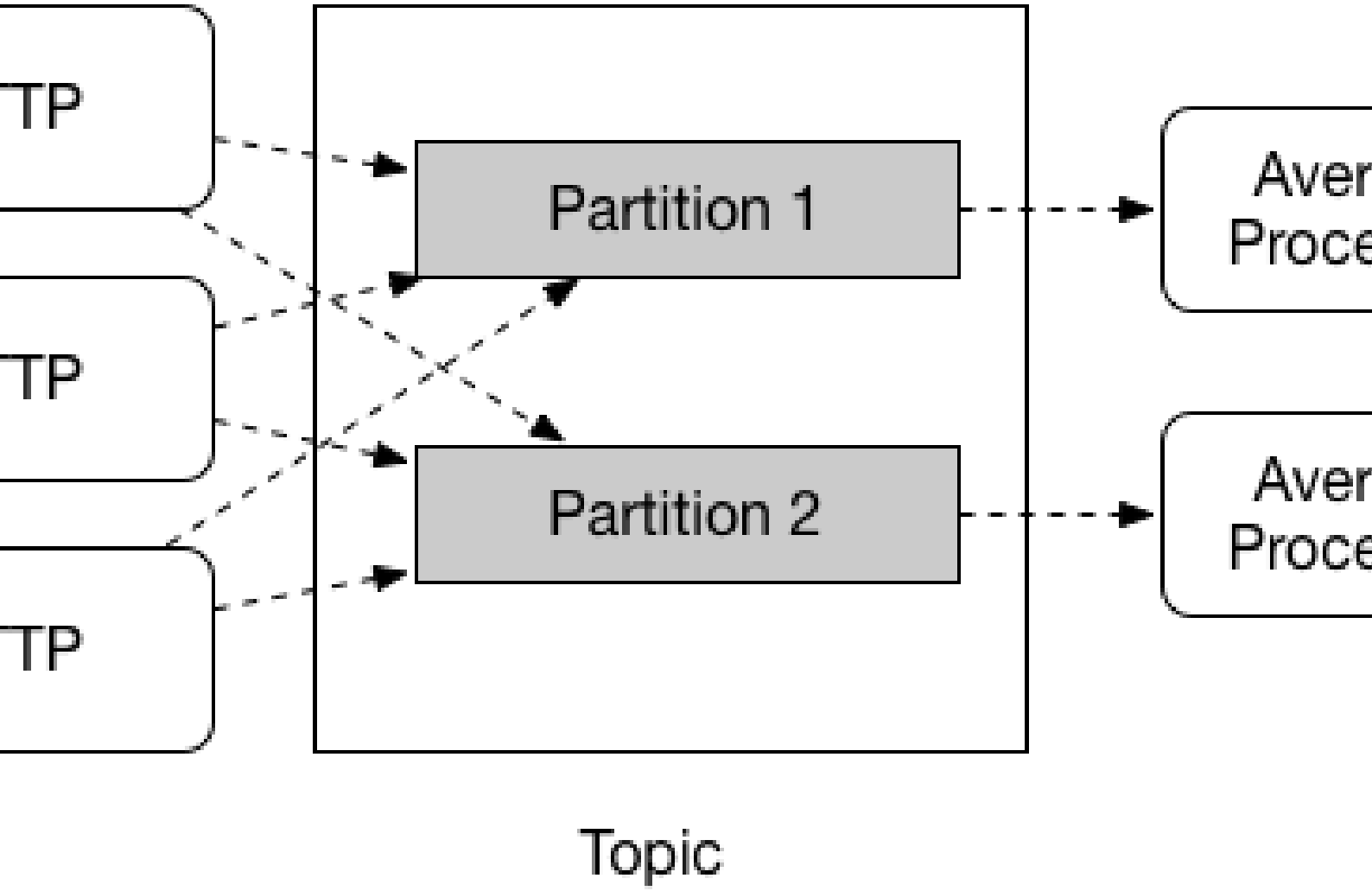


Figure 5.4. Spring Cloud Stream Partitioning

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.

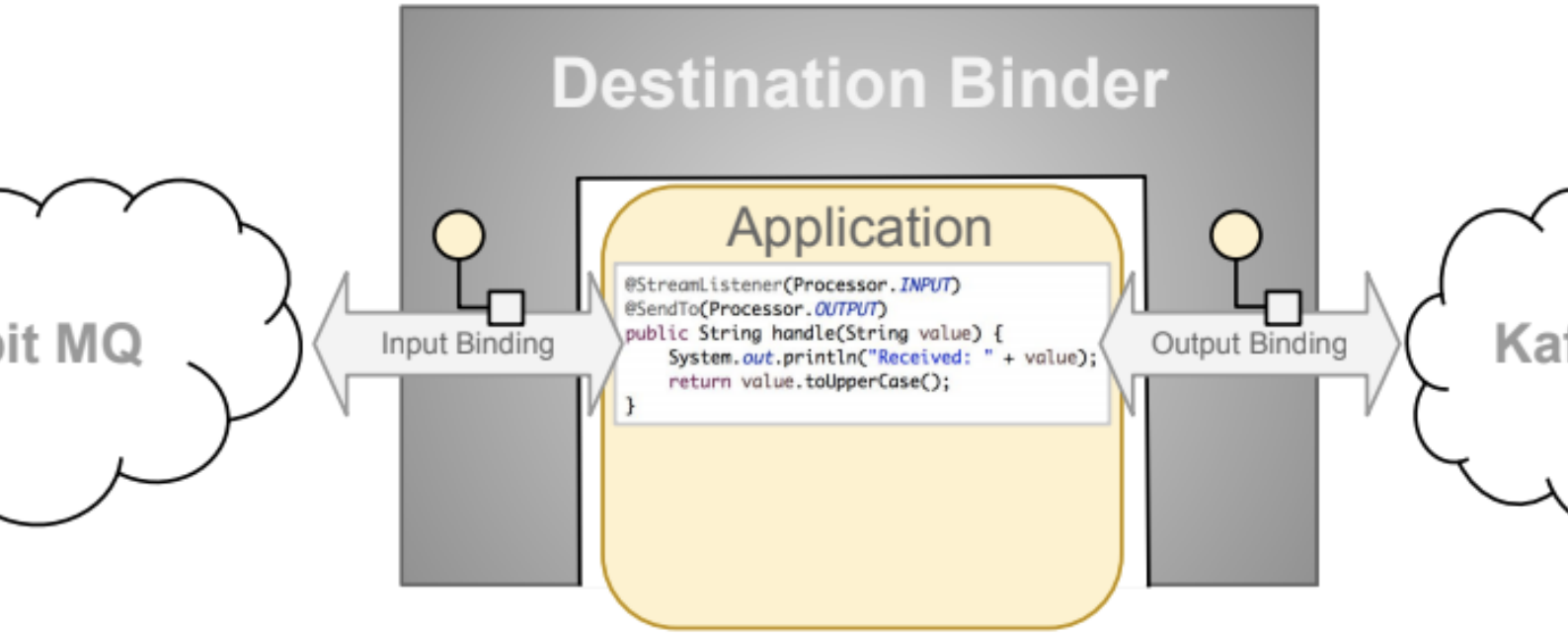
Note

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

6. Programming Model

To understand the programming model, you should be familiar with the following core concepts:

- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Destination Bindings:** Bridge between the external messaging systems and application provided *Producers* and *Consumers* of messages (created by the Destination Binders).
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).



6.1 Destination Binders

Destination Binders are extension components of Spring Cloud Stream responsible for providing the necessary configuration and implementation to facilitate integration with external messaging systems. This integration is responsible for connectivity, delegation, and routing of messages to and from producers and consumers, data type conversion, invocation of the user code, and more.

Binders handle a lot of the boiler plate responsibilities that would otherwise fall on your shoulders. However, to accomplish that, the binder still needs some help in the form of minimalistic yet required set of instructions from the user, which typically come in the form of some type of configuration.

While it is out of scope of this section to discuss all of the available binder and binding configuration options (the rest of the manual covers them extensively), *Destination Binding* does require special attention. The next section discusses it in detail.

6.2 Destination Bindings

As stated earlier, *Destination Bindings* provide a bridge between the external messaging system and application-provided *Producers* and *Consumers*.

Applying the @EnableBinding annotation to one of the application's configuration classes defines a destination binding. The `@EnableBinding` annotation itself is meta-annotated with `@Configuration` and triggers the configuration of the Spring Cloud Stream infrastructure.

The following example shows a fully configured and functioning Spring Cloud Stream application that receives the payload of the message from the `INPUT` destination as a `String` type (see Chapter 9, *Content Type Negotiation* section), logs it to the console and sends it to the `OUTPUT` destination after converting it to upper case.

```
@SpringBootApplication
@EnableBinding(Processor.class)
public class MyApplication {

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

 @StreamListener(Processor.INPUT)
 @SendTo(Processor.OUTPUT)
 public String handle(String value) {
  System.out.println("Received: " + value);
  return value.toUpperCase();
 }
}
```

As you can see the `@EnableBinding` annotation can take one or more interface classes as parameters. The parameters are referred to as *bindings*, and they contain methods representing *bindable components*. These components are typically message channels (see Spring Messaging) for channel-based binders (such as Rabbit, Kafka, and others). However other types of bindings can provide support for the native features of the corresponding technology. For example Kafka Streams binder (formerly known as KStream) allows native bindings directly to Kafka Streams (see Kafka Streams for more details).

Spring Cloud Stream already provides *binding* interfaces for typical message exchange contracts, which include:

- **Sink:** Identifies the contract for the message consumer by providing the destination from which the message is consumed.
- **Source:** Identifies the contract for the message producer by providing the destination to which the produced message is sent.
- **Processor:** Encapsulates both the sink and the source contracts by exposing two destinations that allow consumption and production of messages.

```
public interface Sink {

  String INPUT = "input";

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

```
public interface Source {

  String OUTPUT = "output";

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

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

While the preceding example satisfies the majority of cases, you can also define your own contracts by defining your own bindings interfaces and use `@Input` and `@Output` annotations to identify the actual *bindable components*.

For example:

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

Using the interface shown in the preceding example as a parameter to `@EnableBinding` triggers the creation of the three bound channels named `orders`, `hotDrinks`, and `coldDrinks`, respectively.

You can provide as many binding interfaces as you need, as arguments to the `@EnableBinding` annotation, as shown in the following example:

```
@EnableBinding(value = { Orders.class, Payment.class })
```

In Spring Cloud Stream, the bindable `MessageChannel` components are the Spring Messaging `MessageChannel` (for outbound) and its extension, `SubscribableChannel`, (for inbound).

Pollable Destination Binding

While the previously described bindings support event-based message consumption, sometimes you need more control, such as rate of consumption.

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

The following example shows how to bind a pollable consumer:

```
public interface PolledBarista {

    @Input
    PollableMessageSource orders();
 . . .
}
```

In this case, an implementation of `PollableMessageSource` is bound to the `orders` “channel”. See the section called “Using Polled Consumers” for more details.

Customizing Channel Names

By 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 the preceding example, the created bound channel is named `inboundOrders`.

Normally, you need not access individual channels or bindings directly (other then configuring them via `@EnableBinding` annotation). However there may be times, such as testing or other corner cases, when you do.

Aside from generating channels for each binding and registering them as Spring beans, for each bound interface, Spring Cloud Stream generates a bean that implements the interface. That means you can have access to the interfaces representing the bindings or individual channels by auto-wiring either in your application, as shown in the following two examples:

Autowire Binding interface

```
@Autowire
private Source source

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

Autowire individual channel

```
@Autowire
private MessageChannel output;

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

You can also use standard Spring's `@Qualifier` annotation for cases when channel names are customized or in multiple-channel scenarios that require specifically named channels.

The following example shows how to use the @Qualifier annotation in this way:

```
@Autowire
@Qualifier("myChannel")
private MessageChannel output;
```

6.3 Producing and Consuming Messages

You can write a Spring Cloud Stream application by using either Spring Integration annotations or Spring Cloud Stream native annotation.

Spring Integration Support

Spring Cloud Stream is built on the concepts and patterns defined by Enterprise Integration Patterns and relies in its internal implementation on an already established and popular implementation of Enterprise Integration Patterns within the Spring portfolio of projects: Spring Integration framework.

So its only natural for it to support the foundation, semantics, and configuration options that are already established by Spring Integration

For example, you can attach the output channel of a `Source` to a `MessageSource` and use the familiar `@InboundChannelAdapter` annotation, as follows:

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

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "10", maxMessagesPerPoll =
 "1"))
  public MessageSource<String> timerMessageSource() {
```

```
    return () -> new GenericMessage<>("Hello Spring Cloud Stream");
  }
}
```

Similarly, you can use @Transformer or @ServiceActivator while providing an implementation of a message handler method for a *Processor* binding contract, as shown in the following example:

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

Note

While this may be skipping ahead a bit, it is important to understand that, when you consume from the same binding using `@StreamListener` annotation, a pub-sub model is used. Each method annotated with `@StreamListener` receives its own copy of a message, and each one has its own consumer group. However, if you consume from the same binding by using one of the Spring Integration annotation (such as `@Aggregator`, `@Transformer`, or `@ServiceActivator`), those consume in a competing model. No individual consumer group is created for each subscription.

Using @StreamListener Annotation

Complementary to its Spring Integration support, Spring Cloud Stream provides its own `@StreamListener` annotation, modeled after other Spring Messaging annotations (`@MessageMapping`, `@JmsListener`, `@RabbitListener`, and others) and provides conviniences, such as content-based routing and others.

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

  @Autowired
  VotingService votingService;

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

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

For methods that return data, you must use the `@SendTo` annotation to specify the output binding destination for data returned by the method, as shown in the following example:

```
@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 Content-based routing

Spring Cloud Stream supports dispatching messages to multiple handler methods annotated with `@StreamListener` based on conditions.

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 by a SpEL expression in the `condition` argument of the annotation and is evaluated for each message. All the handlers that match the condition are invoked in the same thread, and no assumption must be made about the order in which the invocations take place.

In the following example of a `@StreamListener` with dispatching conditions, all the messages bearing a header `type` with the value `bogey` are dispatched to the `receiveBogey` method, and all the messages bearing a header `type` with the value `bacall` are dispatched to the `receiveBacall` method.

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

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bogey'")
    public void receiveBogey(@Payload BogeyPojo bogeyPojo) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "headers['type']=='bacall'")
    public void receiveBacall(@Payload BacallPojo bacallPojo) {
       // handle the message
    }
}
```

Content Type Negotiation in the Context of `condition`

It is important to understand some of the mechanics behind content-based routing using the `condition` argument of `@StreamListener`, especially in the context of the type of the message as a whole. It may also help if you familiarize yourself with the Chapter 9, *Content Type Negotiation* before you proceed.

Consider the following scenario:

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

    @StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Dog'")
    public void bark(Dog dog) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition = "payload.class.simpleName=='Cat'")
    public void purr(Cat cat) {
       // handle the message
    }
}
```

The preceding code is perfectly valid. It compiles and deploys without any issues, yet it never produces the result you expect.

That is because you are testing something that does not yet exist in a state you expect. That is because the payload of the message is not yet converted from the wire format (`byte[]`) to the desired type. In other words, it has not yet gone through the type conversion process described in the Chapter 9, *Content Type Negotiation*.

So, unless you use a SPeL expression that evaluates raw data (for example, the value of the first byte in the byte array), use message header-based expressions (such as `condition = "headers['type']=='dog'"`).

Note

At the moment, dispatching through `@StreamListener` conditions is supported only for channel-based binders (not for reactive programming) support.

Spring Cloud Function support

Since Spring Cloud Stream v2.1, another alternative for defining *stream handlers* and *sources* is to use build-in support for Spring Cloud Function where they can be expressed as beans of type `java.util.function.[Supplier/Function/Consumer]`.

To specify which functional bean to bind to the external destination(s) exposed by the bindings, you must provide `spring.cloud.stream.function.definition` property.

Here is the example of the Processor application exposing message handler as `java.util.function.Function`

```
@SpringBootApplication
@EnableBinding(Processor.class)
public class MyFunctionBootApp {

 public static void main(String[] args) {
  SpringApplication.run(MyFunctionBootApp.class, "--
spring.cloud.stream.function.definition=toUpperCase");
 }

 @Bean
 public Function<String, String> toUpperCase() {
  return s -> s.toUpperCase();
 }
}
```

In the above you we simply define a bean of type `java.util.function.Function` called *toUpperCase* and identify it as a bean to be used as message handler whose 'input' and 'output' must be bound to the external destinations exposed by the Processor binding.

Below are the examples of simple functional applications to support Source, Processor and Sink.

Here is the example of a Source application defined as `java.util.function.Supplier`

```
@SpringBootApplication
@EnableBinding(Source.class)
public static class SourceFromSupplier {
 public static void main(String[] args) {
  SpringApplication.run(SourceFromSupplier.class, "--spring.cloud.stream.function.definition=date");
 }
 @Bean
 public Supplier<Date> date() {
  return () -> new Date(12345L);
 }
}
```

Here is the example of a Processor application defined as `java.util.function.Function`

```
@SpringBootApplication
@EnableBinding(Processor.class)
public static class ProcessorFromFunction {
 public static void main(String[] args) {
  SpringApplication.run(ProcessorFromFunction.class, "--
spring.cloud.stream.function.definition=toUpperCase");
 }
 @Bean
 public Function<String, String> toUpperCase() {
  return s -> s.toUpperCase();
 }
}
```

Here is the example of a Sink application defined as `java.util.function.Consumer`

```
@EnableAutoConfiguration
@EnableBinding(Sink.class)
public static class SinkFromConsumer {
 public static void main(String[] args) {
  SpringApplication.run(SinkFromConsumer.class, "--spring.cloud.stream.function.definition=sink");
 }
 @Bean
 public Consumer<String> sink() {
  return System.out::println;
 }
}
```

Functional Composition

Using this programming model you can also benefit from functional composition where you can dynamically compose complex handlers from a set of simple functions. As an example let's add the following function bean to the application defined above

```
@Bean
public Function<String, String> wrapInQuotes() {
 return s -> "\"" + s + "\"";
}
```

and modify the `spring.cloud.stream.function.definition` property to reflect your intention to compose a new function from both 'toUpperCase' and 'wrapInQuotes'. To do that Spring Cloud Function allows you to use | (pipe) symbol. So to finish our example our property will now look like this:

```
–spring.cloud.stream.function.definition=toUpperCase|wrapInQuotes
```

Using Polled Consumers

Overview

When using polled consumers, you poll the `PollableMessageSource` on demand. Consider the following example of a polled consumer:

```
public interface PolledConsumer {

    @Input
    PollableMessageSource destIn();

    @Output
    MessageChannel destOut();

}
```

Given the polled consumer in the preceding example, you might use it 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
            }
        }
    };
}
```

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 "???".

Normally, the `poll()` method acknowledges the message when the `MessageHandler` exits. If the method exits abnormally, the message is rejected (not re-queued), but see the section called "Handling Errors". You can override that behavior by taking responsibility for the acknowledgment, as shown in the following example:

```
@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 re-queued with `StaticMessageHeaderAccessor.getAcknowledgmentCallback(m).acknowledge(Status.REQUE` any later successfully ack'd messages are redelivered.

There is also an overloaded `poll` method, for which the definition is as follows:

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

The `type` is a conversion hint that allows the incoming message payload to be converted, as shown in the following example:

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

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

Handling Errors

By default, an error channel is configured for the pollable source; if the callback throws an exception, an `ErrorMessage` is sent to the error channel (`<destination>.<group>.errors`); this error channel is also bridged to the global Spring Integration `errorChannel`.

You can subscribe to either error channel with a `@ServiceActivator` to handle errors; without a subscription, the error will simply be logged and the message will be acknowledged as successful. If the error channel service activator throws an exception, the message will be rejected (by default) and won't be redelivered. If the service activator throws a `RequeueCurrentMessageException`, the message will be requeued at the broker and will be again retrieved on a subsequent poll.

If the listener throws a `RequeueCurrentMessageException` directly, the message will be requeued, as discussed above, and will not be sent to the error channels.

6.4 Error Handling

Errors happen, and Spring Cloud Stream provides several flexible mechanisms to handle them. The error handling comes in two flavors:

- **application:** The error handling is done within the application (custom error handler).
- **system:** The error handling is delegated to the binder (re-queue, DL, and others). Note that the techniques are dependent on binder implementation and the capability of the underlying messaging middleware.

Spring Cloud Stream uses the Spring Retry library to facilitate successful message processing. See the section called "Retry Template" for more details. However, when all fails, the exceptions thrown by the message handlers are propagated back to the binder. At that point, binder invokes custom error handler or communicates the error back to the messaging system (re-queue, DLQ, and others).

Application Error Handling

There are two types of application-level error handling. Errors can be handled at each binding subscription or a global handler can handle all the binding subscription errors. Let's review the details.

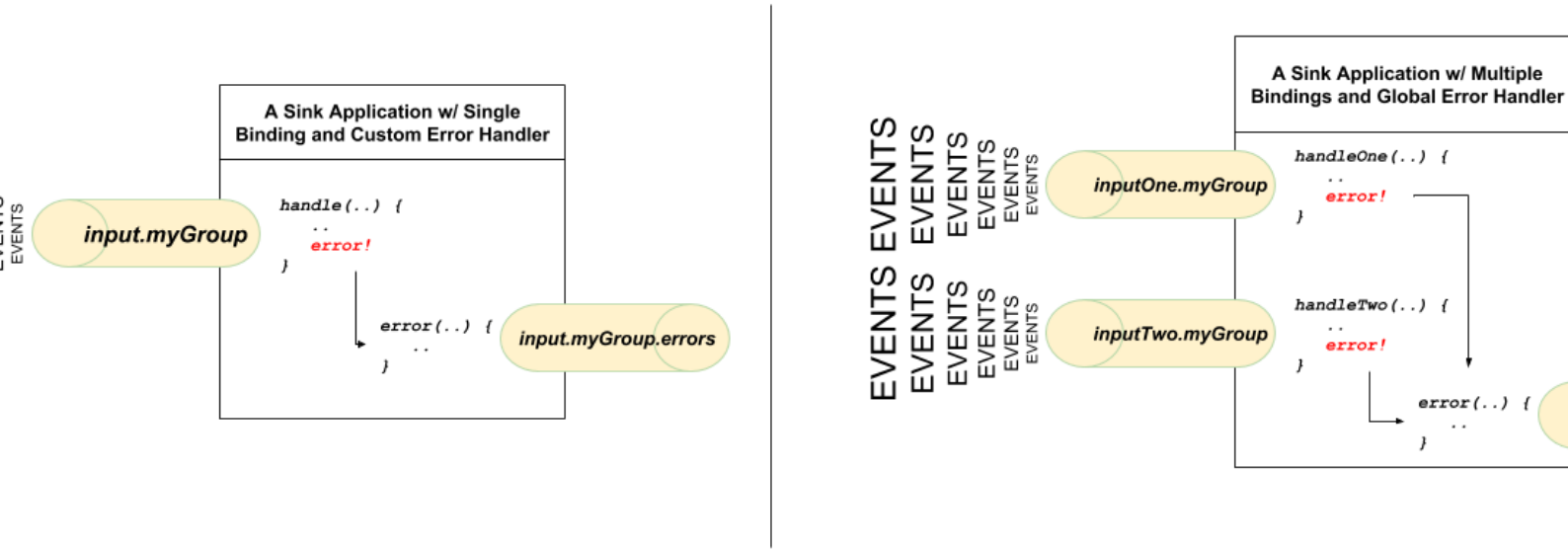


Figure 6.1. A Spring Cloud Stream Sink Application with Custom and Global Error Handlers

For each input binding, Spring Cloud Stream creates a dedicated error channel with the following semantics `<destinationName>.errors`.

Note

The `<destinationName>` consists of the name of the binding (such as `input`) and the name of the group (such as `myGroup`).

Consider the following:

```
spring.cloud.stream.bindings.input.group=myGroup
```

```
@StreamListener(Sink.INPUT) // destination name 'input.myGroup'
public void handle(Person value) {
 throw new RuntimeException("BOOM!");
}

@ServiceActivator(inputChannel = Processor.INPUT + ".myGroup.errors") //channel name
 'input.myGroup.errors'
public void error(Message<?> message) {
 System.out.println("Handling ERROR: " + message);
}
```

In the preceding example the destination name is `input.myGroup` and the dedicated error channel name is `input.myGroup.errors`.

Note

The use of @StreamListener annotation is intended specifically to define bindings that bridge internal channels and external destinations. Given that the destination specific error channel does NOT have an associated external destination, such channel is a prerogative of Spring Integration (SI). This means that the handler for such destination must be defined using one of the SI handler annotations (i.e., @ServiceActivator, @Transformer etc.).

Note

If `group` is not specified anonymous group is used (something like `input.anonymous.2K37rb06Q6m2r51-SPIDDQ`), which is not suitable for error handling scenarious, since you don't know what it's going to be until the destination is created.

Also, in the event you are binding to the existing destination such as:

```
spring.cloud.stream.bindings.input.destination=myFooDestination
spring.cloud.stream.bindings.input.group=myGroup
```

the full destination name is `myFooDestination.myGroup` and then the dedicated error channel name is `myFooDestination.myGroup.errors`.

Back to the example…

The `handle(..)` method, which subscribes to the channel named `input`, throws an exception. Given there is also a subscriber to the error channel `input.myGroup.errors` all error messages are handled by this subscriber.

If you have multiple bindings, you may want to have a single error handler. Spring Cloud Stream automatically provides support for a *global error channel* by bridging each individual error channel to the channel named `errorChannel`, allowing a single subscriber to handle all errors, as shown in the following example:

```
@StreamListener("errorChannel")
public void error(Message<?> message) {
 System.out.println("Handling ERROR: " + message);
}
```

This may be a convenient option if error handling logic is the same regardless of which handler produced the error.

System Error Handling

System-level error handling implies that the errors are communicated back to the messaging system and, given that not every messaging system is the same, the capabilities may differ from binder to binder.

That said, in this section we explain the general idea behind system level error handling and use Rabbit binder as an example. NOTE: Kafka binder provides similar support, although some configuration properties do differ. Also, for more details and configuration options, see the individual binder's documentation.

If no internal error handlers are configured, the errors propagate to the binders, and the binders subsequently propagate those errors back to the messaging system. Depending on the capabilities of the messaging system such a system may *drop* the message, *re-queue* the message for re-processing or *send the failed message to DLQ*. Both Rabbit and Kafka support these concepts. However, other binders may not, so refer to your individual binder's documentation for details on supported system-level error-handling options.

Drop Failed Messages

By default, if no additional system-level configuration is provided, the messaging system drops the failed message. While acceptable in some cases, for most cases, it is not, and we need some recovery mechanism to avoid message loss.

DLQ - Dead Letter Queue

DLQ allows failed messages to be sent to a special destination: - *Dead Letter Queue*.

When configured, failed messages are sent to this destination for subsequent re-processing or auditing and reconciliation.

For example, continuing on the previous example and to set up the DLQ with Rabbit binder, you need to set the following property:

```
spring.cloud.stream.rabbit.bindings.input.consumer.auto-bind-dlq=true
```

Keep in mind that, in the above property, `input` corresponds to the name of the input destination binding. The `consumer` indicates that it is a consumer property and `auto-bind-dlq` instructs the binder to configure DLQ for `input` destination, which results in an additional Rabbit queue named `input.myGroup.dlq`.

Once configured, all failed messages are routed to this queue with an error message similar to the following:

```
delivery_mode: 1
headers:
x-death:
count: 1
reason: rejected
queue: input.hello
time: 1522328151
exchange:
routing-keys: input.myGroup
Payload {"name”:"Bob"}
```

As you can see from the above, your original message is preserved for further actions.

However, one thing you may have noticed is that there is limited information on the original issue with the message processing. For example, you do not see a stack trace corresponding to the original error. To get more relevant information about the original error, you must set an additional property:

```
spring.cloud.stream.rabbit.bindings.input.consumer.republish-to-dlq=true
```

Doing so forces the internal error handler to intercept the error message and add additional information to it before publishing it to DLQ. Once configured, you can see that the error message contains more information relevant to the original error, as follows:

```
delivery_mode: 2
headers:
x-original-exchange:
x-exception-message: has an error
x-original-routingKey: input.myGroup
x-exception-stacktrace: org.springframework.messaging.MessageHandlingException: nested exception is
      org.springframework.messaging.MessagingException: has an error, failedMessage=GenericMessage
 [payload=byte[15],
      headers={amqp_receivedDeliveryMode=NON_PERSISTENT, amqp_receivedRoutingKey=input.hello,
 amqp_deliveryTag=1,
      deliveryAttempt=3, amqp_consumerQueue=input.hello, amqp_redelivered=false,
 id=a15231e6-3f80-677b-5ad7-d4b1e61e486e,
      amqp_consumerTag=amq.ctag-skBFapilvtZhDsn0k3ZmQg, contentType=application/json,
 timestamp=1522327846136}]
      at
 org.spring...integ...han...MethodInvokingMessageProcessor.processMessage(MethodInvokingMessageProcessor.java:107)
      at. . . . .
Payload {"name”:"Bob"}
```

This effectively combines application-level and system-level error handling to further assist with downstream troubleshooting mechanics.

Re-queue Failed Messages

As mentioned earlier, the currently supported binders (Rabbit and Kafka) rely on `RetryTemplate` to facilitate successful message processing. See the section called “Retry Template” for details. However, for cases when `max-attempts` property is set to 1, internal reprocessing of the message is disabled. At this point, you can facilitate message re-processing (re-tries) by instructing the messaging system to re-queue the failed message. Once re-queued, the failed message is sent back to the original handler, essentially creating a retry loop.

This option may be feasible for cases where the nature of the error is related to some sporadic yet short-term unavailability of some resource.

To accomplish that, you must set the following properties:

```
spring.cloud.stream.bindings.input.consumer.max-attempts=1
spring.cloud.stream.rabbit.bindings.input.consumer.requeue-rejected=true
```

In the preceding example, the `max-attempts` set to `1` essentially disabling internal re-tries and `requeue-rejected` (short for *requeue rejected messages*) is set to `true`. Once set, the failed message is resubmitted to the same handler and loops continuously or until the handler throws `AmqpRejectAndDontRequeueException` essentially allowing you to build your own re-try logic within the handler itself.

Retry Template

The `RetryTemplate` is part of the Spring Retry library. While it is out of scope of this document to cover all of the capabilities of the `RetryTemplate`, we will mention the following consumer properties that are specifically related to the `RetryTemplate`:

maxAttempts
: The number of attempts to process the message.

  Default: 3.

backOffInitialInterval
: The backoff initial interval on retry.

  Default 1000 milliseconds.

backOffMaxInterval
: The maximum backoff interval.

  Default 10000 milliseconds.

backOffMultiplier
: The backoff multiplier.

  Default 2.0.

defaultRetryable
: Whether exceptions thrown by the listener that are not listed in the `retryableExceptions` are retryable.

Default: `true`.

retryableExceptions
: A map of Throwable class names in the key and a boolean in the value. Specify those exceptions (and subclasses) that will or won't be retried. Also see `defaultRetriable`. Example: `spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false`.

  Default: empty.

While the preceding settings are sufficient for majority of the customization requirements, they may not satisfy certain complex requirements at, which point you may want to provide your own instance of the `RetryTemplate`. To do so configure it as a bean in your application configuration. The application provided instance will override the one provided by the framework. Also, to avoid conflicts you must qualify the instance of the `RetryTemplate` you want to be used by the binder as `@StreamRetryTemplate`. For example,

```
@StreamRetryTemplate
public RetryTemplate myRetryTemplate() {
    return new RetryTemplate();
}
```

As you can see from the above example you don't need to annotate it with `@Bean` since `@StreamRetryTemplate` is a qualified `@Bean`.

6.5 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 through `spring-cloud-stream-reactive`, which needs to be added explicitly to your project.

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

At present Spring Cloud Stream supports the only the Reactor API. In the future, we intend to support a more generic model based on Reactive Streams.

The reactive programming model also uses 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 the incoming and outgoing data flows connect to, respectively.
- The return value of the method, if any, is annotated with `@Output`, indicating the input where data should 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` transitively retrieves 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 by using Spring Initializr with Spring Boot 1.x, which overrides the Reactor version to `2.0.8.RELEASE`. In such cases, you must ensure that the proper version of the artifact is released. You can do so 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”, currently refers to the reactive APIs being used and not to the execution model being reactive (that is, the bound endpoints still use a 'push' rather than a 'pull' model). While some backpressure support is provided by the use of Reactor, we do intend, in a future release, 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 `Flux` type. 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 that can be the `Message` payload, or a POJO that 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 `FluxSender` type, 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`. In that case, it must be annotated with `@Output`. We recommend using the return value of the method when a single output `Flux` is available.

The following example shows a 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 the following example:

```
@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. By using the `@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 with `@EnableBinding`. You cannot use the `@Input` annotation along with `@StreamEmitter`, as the methods marked with this annotation are not listening for any input. Rather, methods marked with `@StreamEmitter` generate 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, a return type, and other considerations.

The remainder of this section contains examples of using the `@StreamEmitter` annotation in various styles.

The following example emits the `Hello, World` message every millisecond and publishes to a Reactor `Flux`:

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

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

In the preceding example, the resulting messages in the `Flux` are sent to the output channel of the `Source`.

The next example is another flavor of an `@StreamEmmitter` that sends a Reactor `Flux`. Instead of returning a `Flux`, the following method uses a `FluxSender` to programmatically send a `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"));
  }
}
```

The next example is exactly same as the above snippet in functionality and style. However, instead of using an explicit `@Output` annotation on the method, it uses the annotation on the method parameter.

```
@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"));
  }
}
```

The last example in this section is yet another flavor of writing reacting sources by using the Reactive Streams Publisher API and taking advantage of the support for it in Spring Integration Java DSL. The `Publisher` in the following example still uses 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();
  }
}
```

7. 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.

7.1 Producers and Consumers

The following image shows the general relationship of producers and consumers:

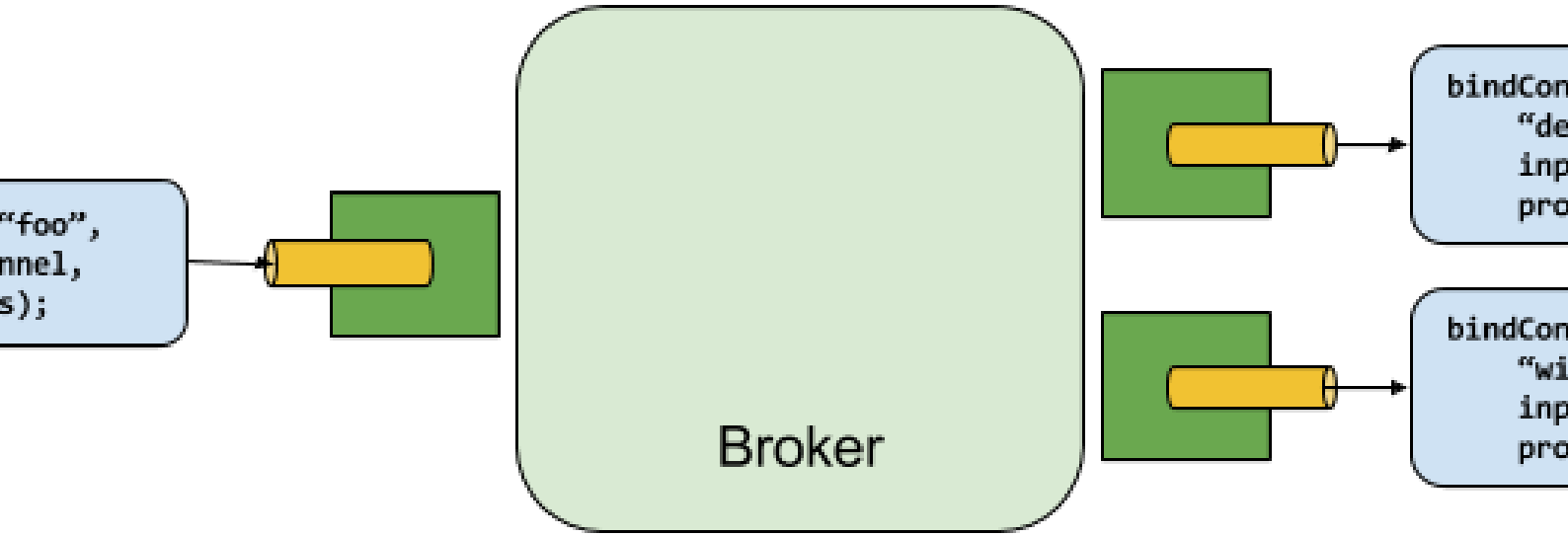


Figure 7.1. Producers and Consumers

A producer is any component that sends messages to a channel. The channel can be bound to an external message broker with 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 sends 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 (that is, it follows normal publish-subscribe semantics). If there are multiple consumer instances bound with the same group name, then messages are 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 (that is, it follows normal queueing semantics).

7.2 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. The following listing shows the definnition of the `Binder` interface:

```
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 that 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 type `Binder` along with the middleware connection infrastructure.
- A `META-INF/spring.binders` file found on the classpath containing one or more binder definitions, as shown in the following example:

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

7.3 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 automatically uses it. For example, a Spring Cloud Stream project that aims to bind only to RabbitMQ can 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, see the documentation of that binder implementation.

7.4 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` file, which is a simple properties file, as shown in the following example:

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

Similar files exist for the other provided binder implementations (such as 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 (for example, `spring.cloud.stream.defaultBinder=rabbit`) or individually, by configuring the binder on each channel binding. For instance, a processor application (that has channels named `input` and `output` for read and write respectively) that 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
```

7.5 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 is 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 disables the default binder configuration process altogether. If you do so, 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 they do not affect the default binder configuration. In order to do so, a binder configuration may have its `defaultCandidate` flag set to false (for example, `spring.cloud.stream.binders.<configurationName>.defaultCandidate=false`). This denotes a configuration that exists independently of the default binder configuration process.

The following example shows a typical configuration for a processor application that connects to two RabbitMQ broker instances:

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: thing1
          binder: rabbit1
        output:
          destination: thing2
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```

7.6 Binding visualization and control

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

Starting with version 2.0 actuator and web are optional, you must first add one of the web dependencies as well as add the actuator dependency manually. The following example shows how to add the dependency for the Web framework:

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

The following example shows how to add the dependency for the WebFlux framework:

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

You can add the Actuator dependency as follows:

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

Note

To run Spring Cloud Stream 2.0 apps in Cloud Foundry, you must add `spring-boot-starter-web` and `spring-boot-starter-actuator` to the classpath. Otherwise, the application will not start due to health check failures.

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

Once those prerequisites are satisfied. you should see the following in the logs when application start:

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

To visualize the current bindings, access the following URL: `<host>:<port>/actuator/bindings`

Alternative, to see a single binding, access one of the URLs similar to the following: `<host>:<port>/actuator/bindings/myBindingName`

You can also stop, start, pause, and resume individual bindings by posting to the same URL while providing a `state` argument as JSON, as shown in the following examples:

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

Note

PAUSED and RESUMED work only when the corresponding binder and its underlying technology supports it. Otherwise, you see the warning message in the logs. Currently, only Kafka binder supports the PAUSED and RESUMED states.

7.7 Binder Configuration Properties

The following properties are available when customizing binder configurations. These properties exposed via `org.springframework.cloud.stream.config.BinderProperties`

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 inherits 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 property is set, the context in which the binder is being created is not a child of the application context. This setting 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 setting allows adding binder configurations without interfering with the default processing.

  Default: `true`.

8. Configuration Options

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

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

8.1 Binding Service Properties

These properties are exposed via `org.springframework.cloud.stream.config.BindingServiceProperties`

spring.cloud.stream.instanceCount
: The number of deployed instances of an application. Must be set for partitioning on the producer side. Must be set on the consumer side when 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 (letting any destination 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 detects a suitable bound service (for example, a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and uses it for creating connections (usually through Spring Cloud Connectors). When set to `true`, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (for example, 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 (in seconds) between retrying binding creation when, for example, the binder does not support late binding and the broker (for example, Apache Kafka) is down. Set it to zero to treat such conditions as fatal, preventing the application from starting.

  Default: `30`

8.2 Binding Properties

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

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

When it comes to avoiding repetitions for extended binding properties, this format should be used - `spring.cloud.stream.<binder-type>.default.<producer|consumer>.<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 ise included at runtime.

Common Binding Properties

These properties are exposed via `org.springframework.cloud.stream.config.BindingProperties`

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

Default values can be set by using the `spring.cloud.stream.default` prefix (for example`spring.cloud.stream.default.contentType=application/json`).

destination
: The target destination of a channel on the bound middleware (for example, 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. See “Chapter 9, *Content Type Negotiation*”.

  Default: `application/json`.

binder
: The binder used by this binding. See “Section 7.4, “Multiple Binders on the Classpath”” for details.

  Default: `null` (the default binder is used, if it exists).

Consumer Properties

These properties are exposed via `org.springframework.cloud.stream.binder.ConsumerProperties`

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

Default values can be set by using the `spring.cloud.stream.default.consumer` prefix (for example, `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`, it uses the middleware's native header mechanism. When set to `embeddedHeaders`, it embeds headers into the message payload.

  Default: depends on the 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`.

defaultRetryable
: Whether exceptions thrown by the listener that are not listed in the `retryableExceptions` are retryable.

  Default: `true`.

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

value, it defaults to `spring.cloud.stream.instanceIndex`. See "Section 11.2, "Instance Index and Instance Count"" for more information.

Default: `-1`.

instanceCount
: When set to a value greater than equal to zero, it allows customizing the instance count of this consumer (if different from `spring.cloud.stream.instanceCount`). When set to a negative value, it defaults to `spring.cloud.stream.instanceCount`. See "Section 11.2, "Instance Index and Instance Count"" for more information.

  Default: `-1`.

retryableExceptions
: A map of Throwable class names in the key and a boolean in the value. Specify those exceptions (and subclasses) that will or won't be retried. Also see `defaultRetriable`. Example: `spring.cloud.stream.bindings.input.consumer.retryable-exceptions.java.lang.IllegalStateException=false`.

  Default: empty.

useNativeDecoding
: When set to `true`, the inbound message is deserialized directly by the client library, which must be configured correspondingly (for example, setting an appropriate Kafka producer value deserializer). When this configuration is being used, the inbound message unmarshalling is not based on the `contentType` of the binding. When native decoding is used, it is the responsibility of the producer to use an appropriate encoder (for example, the Kafka producer value serializer) to serialize the outbound message. Also, when native encoding and decoding is used, the `headerMode=embeddedHeaders` property is ignored and headers are not embedded in the message. See the producer property `useNativeEncoding`.

  Default: `false`.

Producer Properties

These properties are exposed via `org.springframework.cloud.stream.binder.ProducerProperties`

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

Default values can be set by using the prefix `spring.cloud.stream.default.producer` (for example, `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 is partitioned. `partitionCount` must be set to a value greater than 1 to be effective. Mutually exclusive with `partitionKeyExtractorClass`. See "Section 5.6, "Partitioning Support"".

  Default: null.

partitionKeyExtractorClass
: A `PartitionKeyExtractorStrategy` implementation. If set, or if `partitionKeyExpression` is set, outbound data on this channel is partitioned. `partitionCount` must be set to a value greater than 1 to be effective. Mutually exclusive with `partitionKeyExpression`. See "Section 5.6, "Partitioning Support"".

  Default: `null`.

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

  Default: `null`.

partitionSelectorExpression
: A SpEL expression for customizing partition selection. Mutually exclusive with `partitionSelectorClass`. If neither is set, the partition is selected as the `hashCode(key) % partitionCount`, where `key` is computed through 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, it is 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 (for example, by pre-creating durable queues in RabbitMQ).

headerMode
: When set to `none`, it disables header embedding on output. It is 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`, it uses the middleware's native header mechanism. When set to `embeddedHeaders`, it embeds headers into the message payload.

  Default: Depends on the binder implementation.

useNativeEncoding
: When set to `true`, the outbound message is serialized directly by the client library, which must be configured correspondingly (for example, 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 an appropriate decoder (for example, the Kafka consumer value de-serializer) to deserialize the inbound message. Also, when native encoding and decoding is used, the `headerMode=embeddedHeaders` property is ignored and headers are not embedded in the message. See the consumer property `useNativeDecoding`.

Default: `false`.

errorChannelEnabled
: When set to `true`, if the binder supports asynchroous send results, send failures are sent to an error channel for the destination. See “???” for more information.

  Default: `false`.

8.3 Using Dynamically Bound Destinations

Besides the channels defined by using `@EnableBinding`, Spring Cloud Stream lets applications 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 'spring.cloud.stream.dynamicDestinations' property can be used for restricting the dynamic destination names to a known set (whitelisting). If this property is not set, any destination can be bound dynamically.

The `BinderAwareChannelResolver` can be used directly, as shown in the following example of a REST controller using 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))));
    }
}
```

Now consider what happens when we start the application on the default port (8080) and make the following requests with CURL:

```
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 (in the exchange for Rabbit or in the topic for Kafka) with names of '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. The following example includes a router that reads SpEL expressions:

```
@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 is 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 example shows how to use 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.

9. Content Type Negotiation

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`, a message may have to be transformed to a desired shape or 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[]` (that is true for the Kafka and Rabbit binders), but it is governed by the binder implementation.

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

Note

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

9.1 Mechanics

To better understand the mechanics and the necessity behind content-type negotiation, we take a look at a very simple use case by using the following message handler as an example:

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

Note

For simplicity, we assume that this is the only handler in the application (we assume there is no internal pipeline).

The handler shown in the preceding example expects a `Person` object as an argument and produces a `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` type from the wire format to a `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). Consequently, in theory, that should be (and, in some cases, is) enough. However, for the majority of 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 mechanisms to define `contentType` (in order of precedence):

1. **HEADER**: The `contentType` can be communicated through the Message itself. By providing a `contentType` header, you declare the content type to use to locate and apply the appropriate `MessageConverter`.

2. **BINDING**: The `contentType` can be set per destination binding by setting the `spring.cloud.stream.bindings.input.content-type` property.

Note

The `input` segment in the property name corresponds to the actual name of the destination (which is “input” in our case). This approach lets you declare, on a per-binding basis, the content type to use to locate and apply the appropriate `MessageConverter`.

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

As mentioned earlier, the preceding list also demonstrates the order of precedence in case of a tie. For example, a header-provided content type takes precedence over any other content type. The same applies for a content type set on a per-binding basis, which essentially lets you override the default content type. However, it also provides a sensible default (which was determined from 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.

When the non-void handler method returns, if the the return value is already a `Message`, that `Message` becomes the payload. However, when the return value is not a `Message`, the new `Message` is constructed with the return value as the payload while inheriting headers from the input `Message` minus the headers defined or filtered by `SpringIntegrationProperties.messageHandlerNotPropagatedHeaders`. By default, there is only one header set there: `contentType`. This means that the new `Message` does not have `contentType` header set, thus ensuring that the `contentType` can evolve. You can always opt out of 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 by going through the same process of conversion. However, if there is no internal pipeline or you have reached the end of it, the `Message` is sent back to the output destination.

Content Type versus Argument Type

As mentioned earlier, 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`), which trigger right before the invocation of the user-defined handler method (which is when the actual argument type is known to the framework). If the 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 a `contentType` header. When no contentType header was already present, it injects either the per-binding `contentType` header or the default `contentType` header. The combination of `contentType` argument type is the mechanism by which framework determines if message can be converted to a target type. If no appropriate `MessageConverter` is found, an exception is thrown, which you can handle by adding a custom `MessageConverter` (see “Section 9.3, “User-defined Message Converters””).

But what if the payload type matches the target type declared by the handler method? In this case, there is nothing to convert, and the payload is passed unmodified. While this sounds pretty straightforward

and logical, keep in mind handler methods that take a `Message<?>` or `Object` as an argument. By declaring the target type to be `Object` (which is an `instanceof` everything in Java), you essentially forfeit the conversion process.

Note

Do not expect `Message` to be converted into some other type based only on the `contentType`. Remember that the `contentType` is complementary to the target type. If you wish, you can provide a hint, 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 an incoming `Message` to an argument type. The payload of the `Message` could be any type, and it is up to the actual implementation of the `MessageConverter` to support multiple types. For example, some JSON converter may support the payload type as `byte[]`, `String`, and others. This is important when the application contains an internal pipeline (that is, input $\rightarrow$ handler1 $\rightarrow$ handler2 $\rightarrow$ . . . $\rightarrow$ 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 regard the two methods as having the following signatures:

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

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

9.2 Provided MessageConverters

As mentioned earlier, the framework already provides a stack of `MessageConverters` to handle most common use cases. The following list describes the provided `MessageConverters`, in order of precedence (the first `MessageConverter` that works is used):

1. `ApplicationJsonMessageMarshallingConverter`: Variation of the `org.springframework.messaging.converter.MappingJackson2MessageConverter`. Supports conversion of the payload of the `Message` to/from POJO for cases when `contentType` is `application/json` (DEFAULT).
2. `TupleJsonMessageConverter`: **DEPRECATED** Supports conversion of the payload of the `Message` to/from `org.springframework.tuple.Tuple`.
3. `ByteArrayMessageConverter`: Supports conversion of the payload of the `Message` from `byte[]` to `byte[]` for cases when `contentType` is `application/octet-stream`. It is 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`. It invokes Object's `toString()` method or, if the payload is `byte[]`, a 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`. It supports conversion of any type when `contentType` is `application/x-java-object`. It expects the actual type information to be embedded in the `contentType` as an attribute (for example, `application/x-java-object;type=foo.bar.Cat`).

When no appropriate converter is found, the framework throws an exception. When that happens, you should check your code and configuration and ensure you did not miss anything (that is, ensure that you provided a `contentType` by using a binding or a header). However, most likely, you found some uncommon case (such as a custom `contentType` perhaps) and the current stack of provided `MessageConverters` does not know how to convert. If that is the case, you can add custom `MessageConverter`. See Section 9.3, "User-defined Message Converters".

9.3 User-defined Message Converters

Spring Cloud Stream exposes a mechanism to define and register additional `MessageConverters`. To use it, implement `org.springframework.messaging.converter.MessageConverter`, configure it as a `@Bean`, and annotate it with `@StreamMessageConverter`. It is then apended to the existing stack of `MessageConverter`s.

Note

It is important to understand that custom `MessageConverter` implementations are added to the head of the existing stack. Consequently, custom `MessageConverter` implementations take precedence over the existing ones, which lets you override as well as add to the existing converters.

The following example shows how to create a message converter bean to support a new content type called `application/bar`:

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

    ...

    @Bean
    @StreamMessageConverter
    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 "Chapter 10, *Schema Evolution Support*" for details.

10. Schema Evolution Support

Spring Cloud Stream provides support for schema evolution so that the data can be evolved over time and still work with older or newer producers and consumers and vice versa. 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. However, many applications benefit from having access to an explicit schema that describes the binary data format. A schema registry lets you 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

This following sections goes through the details of various components involved in schema evolution process.

10.1 Schema Registry Client

The client-side abstraction for interacting with schema registry servers is the `SchemaRegistryClient` interface, which has 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 and for interacting with the Confluent Schema Registry.

A client for the Spring Cloud Stream schema registry can be configured by 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, which are quite expensive. Because of this, it uses a `DefaultSchemaRegistryClient` that does not cache responses. If you intend to change the default behavior, you can use the client directly

on your code and override it to the desired outcome. To do so, you have to add the property `spring.cloud.stream.schemaRegistryClient.cached=true` to your application properties.

Schema Registry Client Properties

The Schema Registry Client supports the following properties:

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

Default
`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`

10.2 Avro Schema Registry Client Message Converters

For applications that have a SchemaRegistryClient bean registered with the application context, Spring Cloud Stream auto configures an Apache Avro message converter 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, if the content type of the channel is set to `application/*+avro`, the `MessageConverter` is activated, as shown in the following example:

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

During the outbound conversion, the message converter tries to infer the schema of each outbound messages (based on its type) and register it to a subject (based on the payload type) by using the `SchemaRegistryClient`. If an identical schema is already found, then a reference to it is retrieved. If not, the schema is registered, and a new version number is provided. The message is sent with a `contentType` header by using the following 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` might 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 infers the schema reference from the header of the incoming message and tries to retrieve it. The schema is 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 by setting 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). See the 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
: Registers 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`

10.3 Apache Avro Message Converters

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.

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

- Converters that use the class information of the serialized or deserialized objects or a schema with a location known at startup.
- Converters that use a schema registry. They locate the schemas at runtime and dynamically register new schemas as domain objects evolve.

10.4 Converters with Schema Support

The `AvroSchemaMessageConverter` supports serializing and deserializing messages either by using a predefined schema or by using the schema information available in the class (either reflectively or contained in the `SpecificRecord`). If you provide a custom converter, then the default AvroSchemaMessageConverter bean is not created. The following example shows a custom converter:

To use custom converters, you can simply add it to the application context, optionally specifying one or more `MimeTypes` with which to associate it. The default `MimeType` is `application/avro`.

If the target type of the conversion is a `GenericRecord`, a schema must be set.

The following example shows how to configure a converter in a sink application by registering the Apache Avro `MessageConverter` without a predefined schema. In this example, note that the mime type value is `avro/bytes`, not the default `application/avro`.

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

```
  ...

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

Conversely, the following application registers a converter with a predefined schema (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;
  }
}
```

10.5 Schema Registry Server

Spring Cloud Stream provides a schema registry server implementation. To use it, you can add the `spring-cloud-stream-schema-server` artifact to your project and use the `@EnableSchemaRegistryServer` annotation, which adds 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` property. The `spring.cloud.stream.schema.server.path` property 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 property enables the deletion of a 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 by using the Spring Boot SQL database and JDBC configuration options.

The following example shows a Spring Boot application that enables the schema registry:

```
@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 / — see “the section called “Registering a New Schema”””
- 'GET /{subject}/{format}/{version}' — see “the section called “Retrieving an Existing Schema by Subject, Format, and Version”””

- `GET /{subject}/{format}` — see "the section called "Retrieving an Existing Schema by Subject and Format""
- `GET /schemas/{id}` — see "the section called "Retrieving an Existing Schema by ID""
- `DELETE /{subject}/{format}/{version}` — see "the section called "Deleting a Schema by Subject, Format, and Version""
- `DELETE /schemas/{id}` — see "the section called "Deleting a Schema by ID""
- `DELETE /{subject}` — see "the section called "Deleting a Schema by Subject""

Registering a New Schema

To register a new schema, send a `POST` request to the `/` endpoint.

The `/` accepts a JSON payload with the following fields:

- `subject`: The schema subject
- `format`: The schema format
- `definition`: The schema definition

Its response is a schema object in JSON, with the following fields:

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Retrieving an Existing Schema by Subject, Format, and Version

To retrieve an existing schema by subject, format, and version, send `GET` request to the `/{subject}/{format}/{version}` endpoint.

Its response is a schema object in JSON, with the following fields:

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Retrieving an Existing Schema by Subject and Format

To retrieve an existing schema by subject and format, send a `GET` request to the `/subject/format` endpoint.

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

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Retrieving an Existing Schema by ID

To retrieve a schema by its ID, send a `GET` request to the `/schemas/{id}` endpoint.

Its response is a schema object in JSON, with the following fields:

- `id`: The schema ID
- `subject`: The schema subject
- `format`: The schema format
- `version`: The schema version
- `definition`: The schema definition

Deleting a Schema by Subject, Format, and Version

To delete a schema identified by its subject, format, and version, send a `DELETE` request to the `/{subject}/{format}/{version}` endpoint.

Deleting a Schema by ID

To delete a schema by its ID, send a `DELETE` request to the `/schemas/{id}` endpoint.

Deleting a Schema by Subject

`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. `Schema` 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 who upgrade should migrate their existing schemas to the new table before upgrading.

Using Confluent's Schema Registry

The default configuration creates a `DefaultSchemaRegistryClient` bean. If you want to use the Confluent schema registry, you need to create a bean of type `ConfluentSchemaRegistryClient`,

which supersedes the one configured by default by the framework. The following example shows how to create such a bean:

```
@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 4.0.0.

10.6 Schema Registration and Resolution

To better understand how Spring Cloud Stream registers and resolves new schemas and its use of Avro schema comparison features, we provide two separate subsections:

- "the section called "Schema Registration Process (Serialization)""
- "the section called "Schema Resolution Process (Deserialization)""

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 is inferred if the `spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled` property is set to `true` (the default).

Figure 10.1. Schema Writer Resolution Process

Ones a schema is obtained, the converter loads its metadata (version) from the remote server. First, it queries a local cache. If no result is found, it submits the data to the server, which replies with versioning information. The converter always caches the results to avoid the overhead of querying the Schema Server for every new message that needs to be serialized.

Figure 10.2. Schema Registration Process

With the schema version information, the converter sets the `contentType` header of the message to carry the version information — for example: `application/vnd.user.v1+avro`.

Schema Resolution Process (Deserialization)

When reading messages that contain version information (that is, a `contentType` header with a scheme like the one described under "the section called "Schema Registration Process (Serialization)""), the converter queries the Schema server to fetch the writer schema of the message. Once it has found the correct schema of the incoming message, it retrieves the reader schema and, by using Avro's schema resolution support, reads it into the reader definition (setting defaults and any missing properties).

Figure 10.3. Schema Reading Resolution Process

Note

You should understand the difference between a writer schema (the application that wrote the message) and a reader schema (the receiving application). We suggest taking a moment to read the Avro terminology and understand the process. Spring Cloud Stream always fetches 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.

11. Inter-Application Communication

Spring Cloud Stream enables communication between applications. Inter-application communication is a complex issue spanning several concerns, as described in the following topics:

- “Section 11.1, “Connecting Multiple Application Instances””
- “Section 11.2, “Instance Index and Instance Count””
- “Section 11.3, “Partitioning””

11.1 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.

Suppose a design calls for the Time Source application to send data to the Log Sink application. You could use a common destination named `ticktock` for bindings within both applications.

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

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

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

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

11.2 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 have `spring.cloud.stream.instanceCount` set to `3`, and the individual applications have `spring.cloud.stream.instanceIndex` set to `0`, `1`, and `2`, respectively.

When Spring Cloud Stream applications are deployed through 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 (for example, the Kafka binder) in order to ensure that data are split correctly across multiple consumer instances.

11.3 Partitioning

Partitioning in Spring Cloud Stream consists of two tasks:

- “the section called “Configuring Output Bindings for Partitioning”””
- “the section called “Configuring Input Bindings for Partitioning”””

Configuring Output Bindings for Partitioning

You can configure an output binding to send partitioned data by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorName` 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 that example configuration, data is sent to the target partition by 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 that 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 an implementation of `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` and configuring it as a bean (by using the `@Bean` annotation). If 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 with the `partitionKeyExtractorName` property, as shown in the following example:

```
--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` by setting the `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.

Once the message key is calculated, the partition selection process determines the target partition as a value between `0` and `partitionCount - 1`. The default calculation, applicable in most scenarios, is based on the following formula: `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the 'key' (through the `partitionSelectorExpression` property) or by configuring an implementation of `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` as a bean (by using the `@Bean` annotation). Similar to the `PartitionKeyExtractorStrategy`, you can further filter it by using the `spring.cloud.stream.bindings.output.producer.partitionSelectorName` property

when more than one bean of this type is available in the Application Context, as shown in the following example:

```
--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` by setting the `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 shown 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 should be partitioned. The `instanceIndex` must be a unique value across the multiple instances, with a value 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 does not 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 takes 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 subscribes to (you must have at least as many partitions as there are instances). The binder allocates 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 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 in 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 and by letting you rely on the runtime infrastructure to provide information about the instance index and instance count.

12. 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, as shown in the following example:

```
<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, you must make sure that the `test` scope is being used.

The `TestSupportBinder` lets you interact with the bound channels and inspect any messages 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.

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

12.1 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 (for example, integration tests) it is useful to use the actual production binders instead, and that requires disabling the test binder autoconfiguration. To do so, you can exclude the `org.springframework.cloud.stream.test.binder.TestSupportBinderAutoConfiguration` class by using one of the Spring Boot autoconfiguration exclusion mechanisms, as shown 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`, as shown in the following example:

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

13. Health Indicator

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

By default `management.health.binders.enabled` is set to `false`. Setting `management.health.binders.enabled` to `true` enables the health indicator, allowing you to access the `/health` endpoint to retrieve the binder health indicators.

Health indicators are binder-specific and certain binder implementations may not necessarily provide a health indicator.

14. Metrics Emitter

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems.

Spring Cloud Stream provides support for emitting any available micrometer-based metrics to a binding destination, allowing for periodic collection of metric data from stream applications without relying on polling individual endpoints.

Metrics Emitter is activated by defining the `spring.cloud.stream.bindings.applicationMetrics.destination` property, which specifies the name of the binding destination used by the current binder to publish metric messages.

For example:

```
spring.cloud.stream.bindings.applicationMetrics.destination=myMetricDestination
```

The preceding example instructs the binder to bind to `myMetricDestination` (that is, Rabbit exchange, Kafka topic, and others).

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 a unique value per application.

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

spring.cloud.stream.metrics.properties
: Allows white listing application properties that are added to the metrics payload

  Default: null.

spring.cloud.stream.metrics.meter-filter
: Pattern to control the 'meters' one wants to capture. For example, specifying `spring.integration.*` captures metric information for meters whose name starts with `spring.integration.`

  Default: all 'meters' are captured.

spring.cloud.stream.metrics.schedule-interval
: Interval to control the rate of publishing metric data.

  Default: 1 min

Consider the following:

```
java -jar time-source.jar \
    --spring.cloud.stream.bindings.applicationMetrics.destination=someMetrics \
    --spring.cloud.stream.metrics.properties=spring.application** \
    --spring.cloud.stream.metrics.meter-filter=spring.integration.*
```

The following example shows the payload of the data published to the binding destination as a result of the preceding command:

```
{
 "name": "application",
 "createdTime": "2018-03-23T14:48:12.700Z",
 "properties": {
 },
 "metrics": [
  {
   "id": {
    "name": "spring.integration.send",
    "tags": [
     {
      "key": "exception",
      "value": "none"
     },
     {
      "key": "name",
      "value": "input"
     },
     {
      "key": "result",
      "value": "success"
     },
     {
      "key": "type",
      "value": "channel"
     }
    ],
    "type": "TIMER",
    "description": "Send processing time",
    "baseUnit": "milliseconds"
   },
   "timestamp": "2018-03-23T14:48:12.697Z",
   "sum": 130.340546,
   "count": 6,
   "mean": 21.72342433333333,
   "upper": 116.176299,
   "total": 130.340546
  }
 ]
}
```


Note

Given that the format of the Metric message has slightly changed after migrating to Micrometer, the published message will also have a `STREAM_CLOUD_STREAM_VERSION` header set to `2.x` to help distinguish between Metric messages from the older versions of the Spring Cloud Stream.

15. Samples

For Spring Cloud Stream samples, see the spring-cloud-stream-samples repository on GitHub.

15.1 Deploying Stream Applications on CloudFoundry

On CloudFoundry, services are usually exposed through 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 Cloud Foundry Server docs.

Part II. Binder Implementations

16. Apache Kafka Binder

16.1 Usage

To use Apache Kafka binder, you need to add `spring-cloud-stream-binder-kafka` as a dependency to your Spring Cloud Stream application, as shown in the following example for Maven:

```
<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, as shown inn the following example for Maven:

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

16.2 Apache Kafka Binder Overview

The following image shows a simplified diagram of how the Apache Kafka binder operates:

Figure 16.1. 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 of at least that version. This client can communicate with older brokers (see 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.

16.3 Configuration Options

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

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

Kafka Binder Properties

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

  Default: `localhost`.

spring.cloud.stream.kafka.binder.defaultBrokerPort
: `brokers` allows hosts specified with or without port information (for example, `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 are used by both producers and consumers, usage should be restricted to common properties — for example, security settings. Properties here supersede any properties set in boot.

  Default: Empty map.

spring.cloud.stream.kafka.binder.consumerProperties
: Key/Value map of arbitrary Kafka client consumer properties. Properties here supersede any properties set in boot and in the `configuration` property above.

  Default: Empty map.

spring.cloud.stream.kafka.binder.headers
: The list of custom headers that are 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. Health reports 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 configures on topics on which it produces or 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.producerProperties
: Key/Value map of arbitrary Kafka client producer properties. Properties here supersede any properties set in boot and in the `configuration` property above.

  Default: Empty map.

spring.cloud.stream.kafka.binder.replicationFactor
: The replication factor of auto-created topics if `autoCreateTopics` is active. Can be overridden on each binding.

  Default: `1`.

spring.cloud.stream.kafka.binder.autoCreateTopics
: If set to `true`, the binder creates new topics automatically. If set to `false`, the binder relies on the topics being already configured. In the latter case, if the topics do not exist, the binder fails to start.

Note

This setting is independent of the `auto.topic.create.enable` setting of the broker and 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 creates new partitions if required. If set to `false`, the binder relies 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 fails to start.

  Default: `false`.

spring.cloud.stream.kafka.binder.transaction.transactionIdPrefix
: Enables 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 and from Kafka headers. Use this, for example, if you wish to customize the trusted packages in a `DefaultKafkaHeaderMapper` that 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 — for example, `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 the value being the assignments. Used when provisioning new topics. See the `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 is automatically rebalanced between the members of a consumer group. When `false`, each consumer is assigned a fixed set of partitions based on `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex`. This requires both the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties to be set appropriately on each launched instance. The value of the `spring.cloud.stream.instanceCount` property must typically be greater than 1 in this case.

  Default: `true`.

ackEachRecord
: When `autoCommitOffset` is `true`, this setting dictates 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.records` Kafka property, which is set through the consumer `configuration` property. Setting this to `true` may cause a degradation in performance, but doing so 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 is 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 sets 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 commits only for successful messages. It allows a stream to automatically replay from the last successfully processed message, in case of persistent failures. If set to `true`, it always auto-commits (if auto-commit is enabled). If not set (the 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` and `latest`. If the consumer group is set explicitly for the consumer 'binding' (through `spring.cloud.stream.bindings.<channelName>.group`), 'startOffset' is set to `earliest`. Otherwise, it is set to `latest` for the `anonymous` consumer group. Also see `resetOffsets` (earlier in this list).

  Default: null (equivalent to `earliest`).

enableDlq
: When set to true, it enables DLQ behavior for the consumer. By default, messages that result in errors are forwarded to a topic named `error.<destination>.<group>`. The DLQ topic name can be configurable by setting the `dlqName` property. 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 16.6, “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[]`. **Not allowed when `destinationIsPattern` is `true`.**

  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 are 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. Allowed values: `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`

destinationIsPattern
: When true, the destination is treated as a regular expression `Pattern` used to match topic names by the broker. When true, topics are not provisioned, and `enableDlq` is not allowed, because the binder does not know the topic names during the provisioning phase. Note, the time taken to detect new topics that match the pattern is controlled by the consumer property `metadata.max.age.ms`, which (at the time of writing) defaults to 300,000ms (5 minutes). This can be configured using the `configuration` property above.

  Default: `false`

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 — for example, `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 the value being 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 attempts to batch before sending.

  Default: `16384`.

sync
: Whether the producer is synchronous.

  Default: `false`.

batchTimeout
: How long the producer waits to allow more messages to accumulate in the same batch before sending the messages. (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 `!ask,as*` will pass `ash` but not `ask`. `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 uses 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 is used. If a topic already exists with a smaller partition count and `autoAddPartitions` is disabled (the default), the binder fails to start. If a topic already exists with a smaller partition count and `autoAddPartitions` is enabled, new partitions are added. If a topic already exists with a larger number of partitions than the maximum of (`minPartitionCount` or `partitionCount`), the existing partition count is used.

Usage examples

In this section, we show the use of the preceding properties for specific scenarios.

Example: Setting `autoCommitOffset` to `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` be 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, to set `security.protocol` to SASL_SSL, set the following property:

```
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 by 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. The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by 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 by using Spring Boot properties.

The following properties can be used to configure 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.

The following example shows how to launch a Spring Cloud Stream application with SASL and Kerberos by 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
```

The preceding example 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 ignores the Spring Boot properties.

Note

Be careful when using the `autoCreateTopics` and `autoAddPartitions` with Kerberos. Usually, applications may use principals that do not have administrative rights in Kafka and Zookeeper. Consequently, relying on Spring Cloud Stream to create/modify topics may fail. In secure environments, we strongly recommend creating topics and managing ACLs administratively by 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` instances. 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());
   }
  };
 }

}
```

16.4 Error Channels

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See Section 6.4, “Error Handling” 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.

16.5 Kafka Metrics

Kafka binder module exposes the following metrics:

`spring.cloud.stream.binder.kafka.offset`: This metric indicates how many messages have not been yet consumed from a given binder’s topic by a given consumer group. The metrics provided are based on the Mircometer metrics library. The metric contains the consumer group information, topic and the actual lag in committed offset from the latest offset on the topic. This metric is particularly useful for providing auto-scaling feedback to a PaaS platform.

16.6 Dead-Letter Topic Processing

Because you cannot anticipate 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 sample Spring Boot application within this topic is an example of how to route those messages back to the original topic, but it moves them to a

“parking lot” topic after three attempts. The application is 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 a couple of strategies to consider:

- Consider running the rerouting only when the main application is not running. Otherwise, the retries for transient errors are 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.

The following code listings show the sample application:

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();

    }

}
```

16.7 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 example shows 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 supports up to 12 consumer instances (6 if their `concurrency` is 2, 4 if their concurrency is 3, and so on). It is generally best to “over-provision” the partitions to allow for future increases in consumers or concurrency.

Note

The preceding 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 allocates partitions across the instances.

The following Spring Boot application listens to a Kafka stream and prints (to the console) the partition ID to which each message goes:

```
@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 rebalances the partition allocations. If the instance count (or `instance count * concurrency`) exceeds the number of partitions, some consumers are idle.

17. Apache Kafka Streams Binder

17.1 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>
```

17.2 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.

Kafka Streams binder provides binding capabilities for the three major types in Kafka Streams - KStream, KTable and GlobalKTable.

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.

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

As noted early-on, Kafka Streams support in Spring Cloud Stream is 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.

17.3 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.streams.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
: Convenient way to set the application.id for the Kafka Streams application globally at the binder level. If the application contains multiple `StreamListener` methods, then application.id should be set at the binding level per input binding.

  Default: `none`

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. For convenience, if there multiple output bindings and they all require a common value, that can be configured by using the prefix `spring.cloud.stream.kafka.streams.default.producer.`.

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. For convenience, if there multiple input bindings and they all require a common value, that can be configured by using the prefix `spring.cloud.stream.kafka.streams.default.consumer.`.

applicationId
: Setting application.id per input binding.

  Default: `none`

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.kafka.streams.timeWindow.advanceBy
: Value is given in milliseconds.

  Default: none.

17.4 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
```

The above example shows the use of KTable as an input binding. The binder also supports input bindings for GlobalKTable. GlobalKTable binding is useful when you have to ensure that all instances of your application has access to the data updates from the topic. KTable and GlobalKTable bindings are only available on the input. Binder supports both input and output bindings for KStream.

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();
}
```

17.5 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
```

17.6 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.kafka.streams.bindings.output1.producer.valueSerde=IntegerSerde
spring.cloud.stream.kafka.streams.bindings.output2.producer.valueSerde=StringSerde
spring.cloud.stream.kafka.streams.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.

17.7 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());
           }
       }

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

17.8 State Store

State store is created automatically by Kafka Streams when the DSL is used. When processor API is used, you need to register a state store manually. In order to do so, you can use `KafkaStreamsStateStore` annotation. You can specify the name and type of the store, flags to control log and disabling cache, etc. Once the store is created by the binder during the bootstrapping phase, you can access this state store through the processor API. Below are some primitives for doing this.

Creating a state store:

```
@KafkaStreamsStateStore(name="mystate", type= KafkaStreamsStateStoreProperties.StoreType.WINDOW,
 lengthMs=300000)
public void process(KStream<Object, Product> input) {
    ...
}
```

Accessing the state store:

```
Processor<Object, Product>() {

    WindowStore<Object, String> state;

    @Override
    public void init(ProcessorContext processorContext) {
        state = (WindowStore)processorContext.getStateStore("mystate");
    }
    ...
}
```

17.9 Interactive Queries

As part of the public Kafka Streams binder API, we expose a class called `InteractiveQueryService`. 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.

```
@Autowired
private InteractiveQueryService interactiveQueryService;
```

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 =
      interactiveQueryService.getQueryableStoreType("my-store", QueryableStoreTypes.keyValueStore());
```

If there are multiple instances of the kafka streams application running, then before you can query them interactively, you need to identify which application instance hosts the key. `InteractiveQueryService` API provides methods for identifying the host information.

In order for this to work, you must configure the property `application.server` as below:

```
spring.cloud.stream.kafka.streams.binder.configuration.application.server: <server>:<port>
```

Here are some code snippets:

```
org.apache.kafka.streams.state.HostInfo hostInfo = interactiveQueryService.getHostInfo("store-name",
      key, keySerializer);

if (interactiveQueryService.getCurrentHostInfo().equals(hostInfo)) {

    //query from the store that is locally available
}
else {
    //query from the remote host
}
```

17.10 Accessing the underlying KafkaStreams object

`StreamBuilderFactoryBean` from spring-kafka that is responsible for constructing the `KafkaStreams` object can be accessed programmatically. Each `StreamBuilderFactoryBean` is registered as `stream-builder` and appended with the `StreamListener` method name. If your `StreamListener` method is named as `process` for example, the stream builder bean is named as `stream-builder-process`. Since this is a factory bean, it should be accessed by prepending an ampersand (&) when accessing it programmatically. Following is an example and it assumes the `StreamListener` method is named as `process`

```
StreamsBuilderFactoryBean streamsBuilderFactoryBean = context.getBean("&stream-builder-process",
 StreamsBuilderFactoryBean.class);
   KafkaStreams kafkaStreams = streamsBuilderFactoryBean.getKafkaStreams();
```

17.11 State Cleanup

By default, the `Kafkastreams.cleanup()` method is called when the binding is stopped. See the Spring Kafka documentation. To modify this behavior simply add a single `CleanupConfig` `@Bean` (configured to clean up on start, stop, or neither) to the application context; the bean will be detected and wired into the factory bean.

18. RabbitMQ Binder

18.1 Usage

To use the RabbitMQ binder, you can add it to your Spring Cloud Stream application, by using the following Maven coordinates:

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

Alternatively, you can use the Spring Cloud Stream RabbitMQ Starter, as follows:

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

18.2 RabbitMQ Binder Overview

The following simplified diagram shows how the RabbitMQ binder operates:

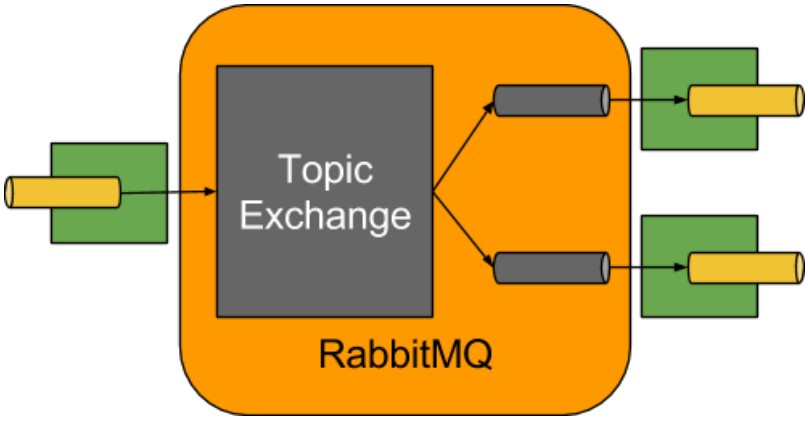


Figure 18.1. RabbitMQ Binder

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

By 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 are delivered to the DLQ after retries are exhausted. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (the default) so that failed messages are routed to the DLQ, instead of being re-queued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it). This feature lets additional information (such as the stack trace in the `x-exception-stacktrace` header) be added to the message in headers. 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 the `republishDeliveryMode` property.

Important

Setting `requeueRejected` to `true` (with `republishToDlq=false`) causes the message to be re-queued and redelivered continually, which is likely not what you want unless the reason for the failure is transient. In general, you should enable retry within the binder by setting `maxAttempts` to greater than one or by setting `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 18.6, “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. You can exclude the class by using the `@SpringBootApplication` annotation.

Starting with version 2.0, the `RabbitMessageChannelBinder` sets the `RabbitTemplate.userPublisherConnection` property to `true` so that the non-transactional producers avoid deadlocks on consumers, which can happen if cached connections are blocked because of a memory alarm on the broker.

Note

Currently, a `multiplex` consumer (a single consumer listening to multiple queues) is only supported for message-driven conssumers; polled consumers can only retrieve messages from a single queue.

18.3 Configuration Options

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

For general binding configuration options and properties, see the Spring Cloud Stream core documentation.

RabbitMQ Binder Properties

By default, the RabbitMQ binder uses Spring Boot’s `ConnectionFactory`. Conseuqently, it supports all Spring Boot configuration options for RabbitMQ. (For reference, see 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 use 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 use 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
The 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 is 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>` is appended.

Default: #.

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

Default: `true`.

consumerTagPrefix
Used to create the consumer tag(s); will be appended by #n where n increments for each consumer created. Example: `${spring.application.name}-`

`${spring.cloud.stream.bindings.input.group}-${spring.cloud.stream.instance-index}`.

Default: none - the broker will generate random consumer tags.

deadLetterQueueName
The name of the DLQ

Default: `prefix+destination.dlq`

deadLetterExchange
A DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: 'prefix+DLX'

deadLetterExchangeType
The type of the DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: 'direct'

deadLetterRoutingKey
A dead letter routing key to assign to the queue. Relevant only if `autoBindDlq` is `true`.

Default: `destination`

declareDlx
Whether to declare the dead letter exchange for the destination. Relevant only if `autoBindDlq` is `true`. Set to `false` if you have a pre-configured DLX.

Default: `true`.

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`

dlqExpires
How long before an unused dead letter queue is deleted (in milliseconds).

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`

dlqOverflowBehavior
: Action to take when `dlqMaxLength` or `dlqMaxLengthBytes` is exceeded; currently `drop-head` or `reject-publish` but refer to the RabbitMQ documentation.

  Default: `none`

dlqTtl
: Default time to live to apply to the dead letter queue when declared (in milliseconds).

  Default: `no limit`

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

  Default: `true`.

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

  Default: `true`.

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

  Default: `true`.

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

  Default: `topic`.

exclusive
: Whether to 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 attempts to consume.

  Default: `false`.

expires
: How long before an unused queue is deleted (in milliseconds).

  Default: `no expiration`

failedDeclarationRetryInterval
: The interval (in milliseconds) 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
: The maximum number of messages in the queue.

  Default: `no limit`

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

  Default: `no limit`

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

  Default: `none`

missingQueuesFatal
: When the queue cannot be found, whether to 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`

overflowBehavior
: Action to take when `maxLength` or `maxLengthBytes` is exceeded; currently `drop-head` or `reject-publish` but refer to the RabbitMQ documentation.

  Default: `none`

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. Relevant only when `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 re-queued when retry is disabled or `republishToDlq` is `false`.

  Default: `false`.

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

  Default: `DeliveryMode.PERSISTENT`

republishToDlq
: By default, messages that fail after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, RabbitMQ routes the failed message (unchanged) to the DLQ. If set to `true`, the binder republishs 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 (in milliseconds).

  Default: `no limit`

txSize
: The number of deliveries between acks.

  Default: `1`.

Advanced Listener Container Configuration

To set listener container properties that are not exposed as binder or binding properties, add a single bean of type `ListenerContainerCustomizer` to the application context. The binder and binding properties will be set and then the customizer will be called. The customizer (`configure()` method) is provided with the queue name as well as the consumer group as arguments.

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 (described in the next three entries in this list): 'batchSize', `batchBufferLimit`, and `batchTimeout`. See 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 it to `false` if you have set up your own infrastructure and have previously created and 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
: The 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. Relevant only when `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: 'prefix+DLX'

deadLetterExchangeType
: The type of the DLX to assign to the queue. Relevant only if `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: 'direct'

deadLetterRoutingKey
: A dead letter routing key to assign to the queue. Relevant only when `autoBindDlq` is `true`. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `destination`

declareDlx
: Whether to declare the dead letter exchange for the destination. Relevant only if `autoBindDlq` is `true`. Set to `false` if you have a pre-configured DLX. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `true`.

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). It 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
: The delivery mode.

  Default: `PERSISTENT`.

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

  Default: `none`

dlqDeadLetterRoutingKey
: When a DLQ is declared, a dead letter routing key to assign to that queue. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `none`

dlqExpires
: How long (in milliseconds) before an unused dead letter queue is deleted. Applies only when `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. Applies only when `requiredGroups` are provided and then only to those groups.

dlqMaxLength
: Maximum number of messages in the dead letter queue. Applies only 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. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

dlqMaxPriority
: Maximum priority of messages in the dead letter queue (0-255) Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `none`

dlqTtl
: Default time (in milliseconds) to live to apply to the dead letter queue when declared. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `no limit`

exchangeAutoDelete
: If `declareExchange` is `true`, whether the exchange should be auto-delete (it is 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 and `direct` or `topic` for partitioned destinations.

  Default: `topic`.

expires
: How long (in milliseconds) before an unused queue is deleted. Applies only when `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. Applies only when `requiredGroups` are provided and then only to those groups.

  Default: `false`.

maxLength
: Maximum number of messages in the queue. Applies only when `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. Applies only when `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, such as `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 (in milliseconds) to live to apply to the queue when declared. Applies only when `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.

18.4 Retry With the RabbitMQ Binder

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. However, 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. You can use the following example configuration to enable this feature:

- Set `autoBindDlq` to `true`. The binder create a DLQ. Optionally, you can 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 are routed to the original queue, because the default `deadLetterRoutingKey` is the queue name (`destination.group`). Setting to the default exchange is achieved by setting the property with no value, as shown in the next example.

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

The loop 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 lets you determine how many cycles have occurred.

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

Putting it All Together

The following configuration creates an exchange `myDestination` with queue `myDestination.consumerGroup` bound to a topic exchange with a wildcard routing key #:

```
---
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 a DLQ bound to a direct exchange (DLX) with a routing key of `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 by using the queue name as the routing key, as shown in the following example:

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`.

18.5 Error Channels

Starting with version 1.3, the binder unconditionally sends exceptions to an error channel for each consumer destination and can also be configured to send async producer send failures to an error channel. See “???” for more information.

RabbitMQ has two types of send failures:

- Returned messages,
- Negatively acknowledged Publisher Confirms.

The latter is rare. According to 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 RabbitMQ binder only sends messages to the channels if the connection factory is appropriately configured, as follows:

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

When using Spring Boot configuration for the connection factory, set the following properties:

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

The payload of the `ErrorMessage` for a returned message is a `ReturnedAmqpMessageException` with the following 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 (for example, 312 - No route).
- `replyText`: A text value indicating the reason for the failure (for example, `NO_ROUTE`).
- `exchange`: The exchange to which the message was published.
- `routingKey`: The routing key used when the message was published.

For negatively acknowledged confirmations, the payload is a `NackedAmqpMessageException` with the following 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.

18.6 Dead-Letter Queue Processing

Because you cannot anticipate 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 shows 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 uses the RabbitMQ Delayed Message Exchange

to introduce a delay to the re-queued 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 for 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. 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, as shown in the following example:

```
@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, as shown in the following example:

```
@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);
 }

}
```

18.7 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 Java and YAML examples show how to configure the producer:

Producer.

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

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

    private static final String[] data = new String[] {
            "abc1", "def1", "qux1",
            "abc2", "def2", "qux2",
            "abc3", "def3", "qux3",
            "abc4", "def4", "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

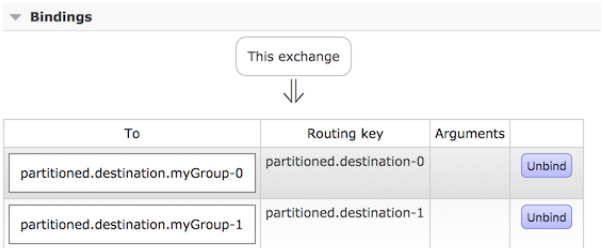
The configuration in the prececing example uses the default partitioning (`key.hashCode() % partitionCount`). This may or may not provide a suitably balanced algorithm, depending on the key values. You can override this default by using the `partitionSelectorExpression` or `partitionSelectorClass` properties.

The `required-groups` property is required only if you need the consumer queues to be provisioned when the producer is deployed. Otherwise, any messages sent to a partition are lost until the corresponding consumer is deployed.

The following configuration provisions a topic exchange:

The following queues are bound to that exchange:

The following bindings associate the queues to the exchange:



The following Java and YAML examples continue the previous examples and show how to configure the consumer:

Consumer.

```
@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.

```
    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 is consumed. Platforms such as Cloud Foundry can have only 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).