Spring Cloud Stream Reference Guide

1.0.0.RC1

Table of Contents

Part I. Reference Guide

Part II. Spring Cloud Stream Reference Manual

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.

1. Introducing Spring Cloud Stream

Spring Cloud Stream is a framework for building message-driven microservices. Spring Cloud Stream builds upon Spring Boot to create DevOps friendly microservice applications and Spring Integration to provide connectivity to message brokers. Spring Cloud Stream provides an opinionated configuration of message brokers, introducing the concepts of persistent pub/sub semantics, consumer groups and partitions across several middleware vendors. This opinionated configuration provides the basis to create stream processing applications.

By adding `@EnableBinding` to your main application, you get immediate connectivity to a message broker and by adding `@StreamListener` to a method, you will receive events for stream processing.

Here's a sample sink application for receiving external messages:

```
@SpringBootApplication
public class StreamApplication {

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

@EnableBinding(Sink.class)
public class TimerSource {

  ...

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

`@EnableBinding` is parameterized by one or more interfaces (in this case a single `Sink` interface), which declares input and/or output channels. The interfaces `Source`, `Sink` and `Processor` are provided but you can define others. Here's the definition of `Source`:

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

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

The `@Input` annotation is used to identify input channels (messages entering the app), and `@Output` is used to identify output channels (messages leaving the app). These annotations are optionally parameterized by a channel name. If the name is not provided then the method name is used instead. An implementation of the interface is created for you and can be used in the application context by autowiring it, e.g. into a test case:

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

  @Autowired
  private Sink sink;

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

```
  }
}
```

2. Spring Cloud Stream Main Concepts

Spring Cloud Stream provides a number of abstractions and primitives that simplify writing message-driven microservices. In this section we will provide an overview of:

- Spring Cloud Stream application model together with the Binder abstraction
- Persistent publish-subscribe and consumer group support
- Partitioning
- Pluggable Binder API

2.1 Application structure

A Spring Cloud Stream application consists of a middleware-neutral core that communicates with the outside world through input and output channels. The channels are managed and injected into it by the framework, and a `Binder` connects them to the external brokers. Different `Binder` implementations exist for different types of middleware, such as Kafka, Rabbit MQ, Redis or Gemfire, and an extensible API allows you to write your own `Binder`. There is also TestSupportBinder that leaves the channel as-is so a test author can interact with the channels directly and easily assert on what is received.

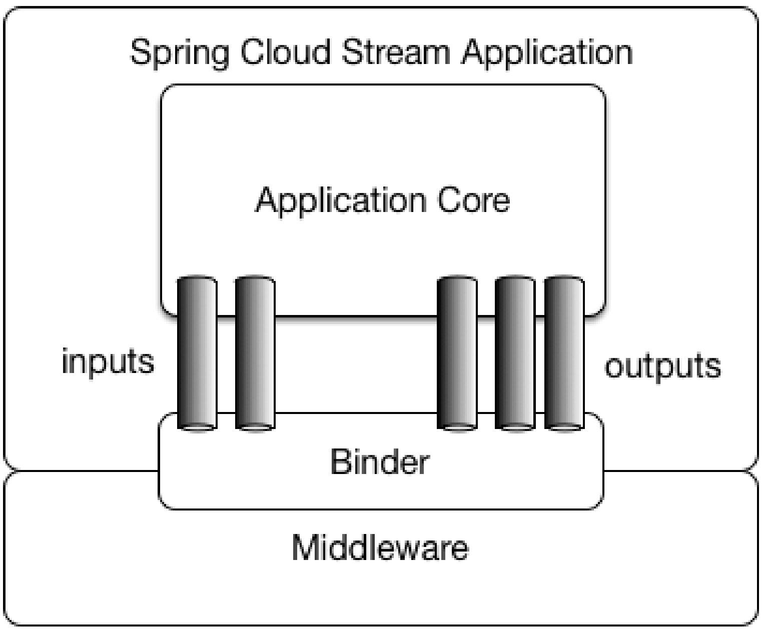

Figure 2.1. Spring Cloud Stream Application

Spring Cloud Stream uses Spring Boot for configuration, and the `Binder` makes it possible for Spring Cloud Stream applications to be flexible in terms of how it connects to the middleware. For example, deployers can dynamically choose the destinations that these channels connect to at runtime (e.g. Kafka topics or Rabbit MQ exchanges). This can be done through external configuration properties in any form that is supported by Spring Boot (application arguments, environment variables, `application.yml` files, etc). Taking the sink example from the previous section, providing the `spring.cloud.stream.bindings.input.destination=raw-sensor-data` property to the application will cause it to read from the `raw-sensor-data` Kafka topic, or from a queue bound to the `raw-sensor-data` exchange in Rabbit MQ. See Section 4.2, “Binding properties”

for more information on the available binder properties you can configure. You are also able to configure middleware specific properties, see ??? for more information.

Spring Cloud Stream will automatically detect and use a binder that is found on the classpath, so you can easily use different types of middleware with the same code, just by including a different binder at build time. For more complex use cases, Spring Cloud Stream also provides the ability of packaging multiple binders within the same application and choosing what type of binder should be used at runtime, and even if multiple binders should be used at runtime for different channels.

Fat JAR

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

2.2 Persistent publish subscribe and consumer groups

Communication between different applications follows a publish-subscribe pattern, with data being broadcast through shared topics. This can be seen in the following picture, which shows a typical deployment for a set of interacting Spring Cloud Stream applications.

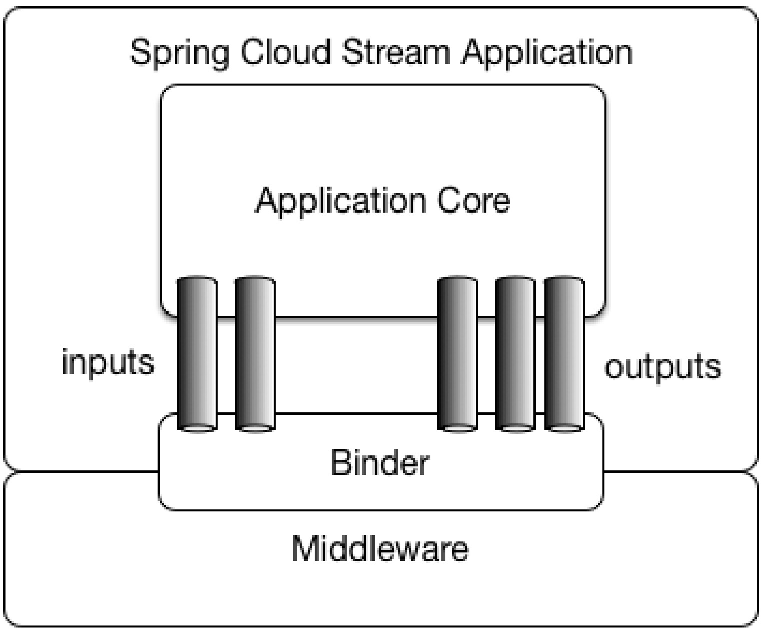

Figure 2.2. Spring Cloud Stream Application topologies

Data reported by sensors to an HTTP endpoint is sent to a common destination named `raw-sensor-data`, from where it is independently processed by a microservice that computes time windowed averages, as well as by a microservice that ingests the raw data into HDFS. In order to do so, both applications will declare the topic as their input at runtime. The publish-subscribe communication model reduces the complexity of both the producer and the consumer, and allows adding new applications to the topology without disrupting the existing flow. For example, downstream from the average calculator we can have a component that calculates the highest temperature values in order to display and monitor them. Later on, we can add an application that interprets the very same flow of averages for fault detection. The fact that all the communication is done through shared topics rather than point to point queues reduces the 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. It also makes it easy for users to work with it across different platform by using the native support of the middleware.

Consumer Groups

While the publish subscribe model ensures that it is easy to connect multiple application by sharing a topic, it is equally important to be able to scale up by creating multiple instances of a given application. When doing so, the different instances would find themselves in a competing consumer relationship with each other: only one of the instances is expected to handle the message. Spring Cloud Stream models this behavior through the concept of a consumer group, which is similar to (and inspired by) the notion of consumer groups in Kafka. Each consumer binding can specify a group name such as `spring.cloud.stream.bindings.input.group=hdfsWrite` or `spring.cloud.stream.bindings.input.group=average`, as shown in the picture. All groups that subscribe to a given destination will receive a copy of the published data, but only one member of the group will receive a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous, independent, single-member consumer group that will be in a publish-subscribe relationship with all the other consumer groups.

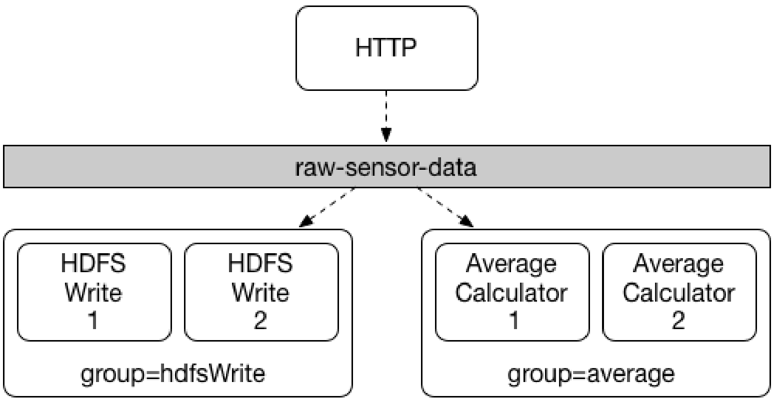

Figure 2.3. Spring Cloud Stream Consumer Groups

Durability

Consistent with the opinionated application model of Spring Cloud Stream, consumer group subscriptions are durable. This is to say that the binder implementation will ensure that group subscriptions are persistent and, once at least one subscription for a group has been created, that group will receive messages, even if they are sent while all the applications of the group were stopped. Anonymous subscriptions are non-durable by nature. For some binder implementations (e.g. Rabbit) 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, a consumer group must be specified for each of its input bindings, in order to prevent its instances from receiving duplicate messages (unless that behavior is desired, which is a less common use case).

2.3 Partitioning

Spring Cloud Stream provides support for partitioning data between multiple instances of a given application. In a partitioned scenario, one or more producer application instances will send data to multiple consumer application instances, ensuring that data with common characteristics is processed by the same consumer instance. The physical communication medium (e.g. the broker topic) is viewed as structured into multiple partitions. This happens regardless of whether the broker type is naturally partitioned (e.g. Kafka) or not (e.g. Rabbit), Spring Cloud Stream provides a common abstraction for implementing partitioned processing use cases in a uniform fashion.

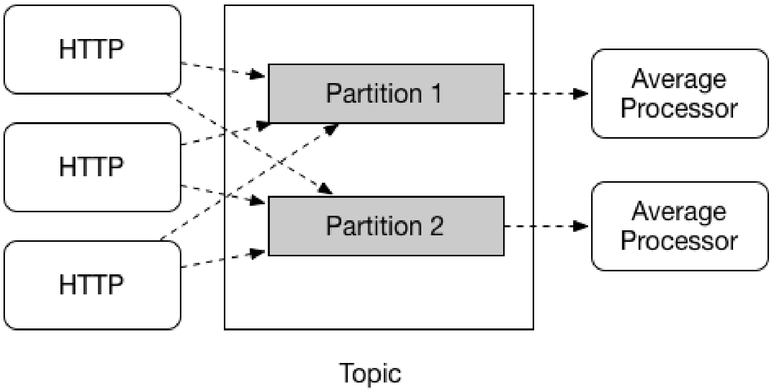

Figure 2.4. Spring Cloud Stream Partitioning

Partitioning is a critical concept in stateful processing, where ensuring that all the related data is processed together is critical for either performance or consistency. For example, in the time-windowed average calculation example, it is important that measurements from the same sensor land in the same application instance.

Setting up a partitioned processing scenario requires configuring both the data producing and the data consuming end.

3. Programming model

This section will describe the programming model of Spring Cloud Stream, which consists from a number of predefined annotations that can be used to declare bound inputs and output channels, as well as how to listen to them.

3.1 Declaring and binding channels

Triggering binding via `@EnableBinding`

A Spring application becomes a Spring Cloud Stream application when the `@EnableBinding` annotation is applied to one of its configuration classes. `@EnableBinding` itself is meta-annotated with `@Configuration`, and triggers the configuration of Spring Cloud Stream infrastructure as follows:

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

`@EnableBinding` can be parameterized with one or more interface classes, containing methods that represent bindable components (typically message channels).

> **Note**
>
> As of version 1.0, the only supported bindable component is the Spring Messaging `MessageChannel` and its extensions `SubscribableChannel` and `PollableChannel`. It is intended for future versions to extend support to other types of components, using the same mechanism. In this documentation, we will continue to refer to channels.

`@Input` and `@Output`

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

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

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

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

   ...
}
```

Customizing channel names

Both @Input and @Output allow specifying a customized name for the channel, as follows:

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

In this case, the name of the bound channel being created will be `inboundOrders`.

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

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

`Source` can be used for applications that have a single outbound channel.

```
public interface Source {

 String OUTPUT = "output";

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

}
```

`Sink` can be used for applications that have a single inbound channel.

```
public interface Sink {

 String INPUT = "input";

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

}
```

`Processor` can be used for applications that have both an inbound and an outbound channel.

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

There is no special handling for either of these interfaces in Spring Cloud Stream, besides of the fact that they are provided out of the box.

Accessing bound channels

Injecting the bound interfaces

For each of the bound interfaces, Spring Cloud Stream will generate a bean that implements it, and for which invoking an `@Input` or `@Output` annotated method will return the bound channel. For example, the bean in the following example will send a message on the output channel every time its `hello` method is invoked, using the injected `Source` bean, and invoking `output()` to retrieve the target channel.

```
@Component
public class SendingBean {
```

```
    private Source source;

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

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

Injecting channels directly

Bound channels can be also injected directly. For example:

```
@Component
public class SendingBean {

    private MessageChannel output;

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

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

Note that if the name of the channel is customized on the declaring annotation, that name should be used instead of the method name. Considering this declaration:

```
public interface CustomSource {
    ...
    @Output("customOutput")
    MessageChannel output();
}
```

The channel will be injected as follows:

```
@Component
public class SendingBean {

    @Autowired
    private MessageChannel output;

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

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

Programming model

Spring Cloud Stream allows you to write applications by either using Spring Integration annotations or Spring Cloud Stream's `@StreamListener` annotation which is modeled after other Spring Messaging annotations (e.g. `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.) but add content type management and type coercion features.

Native Spring Integration support

Due to the fact that Spring Cloud Stream is Spring Integration based, it completely inherits its foundation and infrastructure, as well as the component. For example, the output channel of a `Source` can be attached to a `MessageSource`, as follows:

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

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

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

Or, the channels of a processor can be used in a transformer, as follows:

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

@StreamListener for automatic content type handling

Complementary to the Spring Integration support, Spring Cloud Stream provides a `@StreamListener` annotation of its own modeled by the other similar Spring Messaging annotations (e.g. `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.). It provides a simpler model for handling inbound messages, especially for dealing with use cases that involve content type management and type coercion. Spring Cloud Stream provides an extensible `MessageConverter` mechanism for handling data conversion by bound channels and, in this case, for dispatching to `@StreamListener` annotated methods.

For example, an application that processes external `Vote` events can be declared as follows:

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

  @Autowired
  VotingService votingService;

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

The distinction between this approach and a Spring Integration `@ServiceActivator` becomes relevant if one considers an inbound `Message` with a `String` payload and a `contentType` header of `application/json`. For `@StreamListener`, the `MessageConverter` mechanism will use the `contentType` header to parse the `String` into a `Vote` object.

Just as with the other Spring Messaging methods, method arguments can be annotated with `@Payload`, `@Headers` and `@Header`. For methods that return data, `@SendTo` must be used for specifying the output binding destination for data returned by the methods as follows:

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

  @Autowired
  VotingService votingService;

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

> **Note**
>
> Content type headers can be set by external applications in the case of Rabbit MQ, and they are supported as part of an extended internal protocol by Spring Cloud Stream for any type of transport (even the ones that do not support headers normally, like Kafka).

3.2 Binder SPI

As described above, Spring Cloud Stream provides a binder abstraction for connecting to physical destinations. This section will provide more information about the main concepts behind the Binder SPI, its main components, as well as details specific to different implementations.

Producers and Consumers

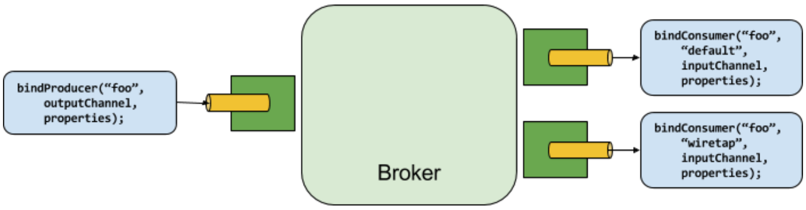

Figure 3.1. Producers and Consumers

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

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

Kafka Binder

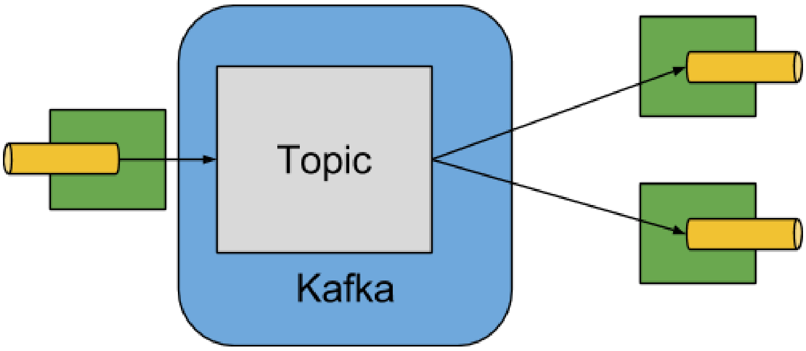

Figure 3.2. Kafka Binder

The Kafka Binder implementation maps the destination to a Kafka topic, and the consumer group maps directly to the same Kafka concept. Spring Cloud Stream does not use the high level consumer, but implements a similar concept for the simple consumer.

RabbitMQ Binder

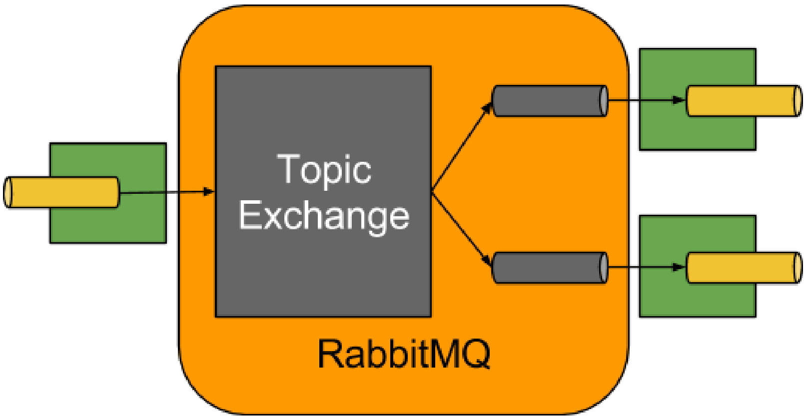

Figure 3.3. RabbitMQ Binder

The RabbitMQ Binder implementation maps the destination to a `TopicExchange`, and for each consumer group, a `Queue` will be bound to that `TopicExchange`. Each consumer instance that binds will trigger creation of a corresponding RabbitMQ `Consumer` instance for its group's `Queue`.

4. Configuration options

Spring Cloud Stream supports general configuration options, as well as configuration for bindings and binders. Some binders allow additional properties for the bindings, supporting middleware-specific features.

All configuration options can be provided to Spring Cloud Stream applications via all the mechanisms supported by Spring Boot: application arguments, environment variables, YML files etc.

4.1 Spring Cloud Stream Properties

spring.cloud.stream.instanceCount
: The number of deployed instances of the same application. Must be set for partitioning and with Kafka. Default value is `1`.

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

spring.cloud.stream.dynamicDestinations
: A list of destinations that can be bound dynamically, for example in a dynamic routing scenario. Only listed destinations can be bound if set. Default empty, allowing any destination to be bound.

spring.cloud.stream.defaultBinder
: The default binder to use, if there are multiple binders configured. See multiple binders.

4.2 Binding properties

Binding properties are supplied using the format `spring.cloud.stream.bindings.<channelName>.<property>=<value>`.`<channelName>` represents the name of the channel being configured, e.g. `output` for a `Source`. In what follows, we will indicate where the `spring.cloud.stream.bindings.<channelName>.` prefix is omitted and focus just on the property name, with the understanding that the prefix will be included at runtime.

Properties for the use of Spring Cloud Stream

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

destination
: The target destination of channel on the bound middleware, e.g. Rabbit MQ exchange or Kafka topic. If not set, the channel name will be used instead.

group
: The consumer group of the channel. This property applies only to inbound bindings. By default it is null, and indicates an anonymous consumer. See consumer groups.

contentType
: The content type of the channel. By default it is `null` and no type coercion is performed. See ???.

binder
: The binder used by this binding. By default, it is set to `null` and will use the default binder, if one exists. See Section 6.2, “Multiple Binders on the Classpath” for details.

Consumer properties

The following binding properties are available for input bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.consumer`:

concurrency
: The concurrency of the inbound consumer. By default, set to `1`.

partitioned
: Must be set to `true` if the consumer is receiving data from a partitioned producer. By default it is set to `false`.

maxAttempts
: The number of attempts of re-processing an inbound message. Default '3'. (Ignored by Kafka, currently).

backOffInitialInterval
: The backoff initial interval on retry. Default `1000`.(Ignored by Kafka, currently).

backOffMaxInterval
: The maximum backoff interval. Default `10000`.(Ignored by Kafka, currently).

backOffMultiplier
: The backoff multiplier. Default `2.0`.

Producer properties

The following binding properties are available for output bindings only and must be prefixed with `spring.cloud.stream.bindings.<channelName>.producer`:

partitionKeyExpression
: A SpEL expression for partitioning outbound data. Default: `null`. If either this property is set or `partitionKeyExtractorClass` is present, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value larger than `1` to be effective. The two options are mutually exclusive. See Section 2.3, “Partitioning”.

partitionKeyExtractorClass
: A `PartitionKeyExtractorStrategy` implementation. Default: `null`. If either this property is set or `partitionKeyExpression` is present, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value larger than `1` to be effective. The two options are mutually exclusive. See Section 2.3, “Partitioning”.

partitionSelectorClass
: A `PartitionSelectorStrategy` implementation. Default `null`. Mutually exclusive with `partitionSelectorExpression`. If none is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

partitionSelectorExpression
: A SpEL expression for customizing partition selection. Default `null`. Mutually exclusive with `partitionSelectorClass`. If none is set, the partition will be selected as the `hashCode(key) % partitionCount`, where `key` is computed via either `partitionKeyExpression` or `partitionKeyExtractorClass`.

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

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

5. Binder-specific configuration

This captures the binder, consumer and producer properties that are specific for several binder implementations.

5.1 Rabbit-specific settings

Rabbit MQ Binder properties

The binder supports the all Spring Boot properties for Rabbit MQ configuration.

In addition to that, it also supports the following properties:

spring.cloud.stream.rabbit.binder.addresses
: A comma-separated list of RabbitMQ server addresses (used only for clustering and in conjunction with `nodes`). Default empty. spring.cloud.stream.rabbit.binder.adminAddresses. Default empty. A comma-separated list of RabbitMQ management plugin URLs - only used when nodes contains more than one entry. Entries in this list must correspond to the corresponding entry in addresses. 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. Entries in this list must correspond to the corresponding entry in addresses. Default empty.

spring.cloud.stream.rabbit.rabbit.username
: The user name. Default `null`.

spring.cloud.stream.rabbit.binder.password
: The password. Default `null`.

spring.cloud.stream.rabbit.binder.vhost
: The virtual host. Default `null`.

spring.cloud.stream.rabbit.binder.useSSL
: True if Rabbit MQ should use SSL.

spring.cloud.stream.rabbit.binder.sslPropertiesLocation
: The location of the SSL properties file, when certificate exchange is used.

spring.cloud.stream.rabbit.binder.compressionLevel
: Compression level for compressed bindings. Defaults to `1` (BEST_LEVEL). See `java.util.zip.Deflater`.

Rabbit MQ 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`.

durableSubscription
Whether subscription should be durable. Only effective if `group` is also set. Default `true`. maxConcurrency: Default `1`. prefetch: Prefetch count. Default `1`.

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

requeueRejected
Whether delivery failures should be requeued. Default `true`.

requestHeaderPatterns
The request headers to be transported. Default `[STANDARD_REQUEST_HEADERS,'*']`.

replyHeaderPatterns
The reply headers to be transported. Default `[STANDARD_REQUEST_HEADERS,'*']`

republishToDlq
By default, failed messages after retries are exhausted are rejected. If a dead-letter queue (DLQ) is configured, rabbitmq will route the failed message (unchanged) to the DLQ. Setting this property to true instructs the bus to republish failed messages to the DLQ, with additional headers, including the exception message and stack trace from the cause of the final failure.

transacted
Whether to use transacted channels. Default `false`.

txSize
The number of deliveries between acks. Default `1`.

Rabbit Producer Properties

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

autoBindDlq
Whether to automatically declare the DLQ and bind it to the binder DLX. Default `false`.

batchingEnabled
True to enable message batching by producers. Default `false`.

batchSize
The number of message to buffer when batching is enabled. Default `100`.

batchBufferLimit
Default `10000`.

batchTimeout
Default `5000`.

compress
Whether data should be compressed when sent. Default `false`.

deliveryMode
Delivery mode. Default `PERSISTENT`.

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

requestHeaderPatterns
: The request headers to be transported. Default `[STANDARD_REQUEST_HEADERS,'*']`.

replyHeaderPatterns
: The reply headers to be transported. Default `[STANDARD_REQUEST_HEADERS,'*']`

5.2 Kafka-specific settings

Kafka binder properties

spring.cloud.stream.kafka.binder.brokers
: A list of brokers that the Kafka binder will connect to. Default `localhost`.

spring.cloud.stream.kafka.binder.defaultBrokerPort
: The list of brokers allows to specify hosts with or without port information, i.e. `host1,host2:port2`. This configuration sets the default port when no port is configured in the broker list. Default `9092`.

spring.cloud.stream.kafka.binder.zkNodes
: A list of Zookeeper nodes for the Kafka binder to connect to. Default `localhost`.

spring.cloud.stream.kafka.binder.defaultZkPort
: The list of Zookeeper nodes allows to specify hosts with or without port information, i.e. `host1,host2:port2`. This configuration sets the default port when no port is configured in the node list. Default `2181`.

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

spring.cloud.stream.kafka.binder.offsetUpdateTimeWindow
: The frequency in milliseconds with which offsets are saved. Ignored if `0`. Default `10000`.

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

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

Kafka Consumer Properties

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

autoCommitOffset
: True to autocommit offsets when a message has been processed. If set to false, an `Acknowledgment` header will be available in the message headers for late acknowledgment. Default `true`.

mode
: When set to `raw`, will disable header parsing on input. Useful when inbound data is coming from outside Spring Cloud Stream applications. Default `embeddedHeaders`.

resetOffsets
: True to reset offsets on the consumer to the value provided by `startOffset`. Default `false`.

startOffset
: The starting offset for new groups or when `resetOffsets` is `true`. Allowed values: `earliest`,`latest`. Defaults to null (equivalent to earliest).

minPartitionCount
: The minimum number of partitions expected by the consumer if it creates the consumed topic automatically. Defaults to `1`.

Kafka Producer Properties

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

bufferSize
: This is an upper limit of how much data the Kafka Producer will attempt to batch before sending – specified in bytes. Default `16384`.

sync
: Whether the producer is synchronous. Defaults to `false`.

batchTimeout
: How long will the producer wait before sending in order to allow more messages to get accumulated in the same batch. Normally the producer will not wait at all, and simply send 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`.

mode
: When set to `raw`, disable header propagation on output. Useful when producing data for non-Spring Cloud Stream applications. Default `embeddedHeaders`.

6. 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. Spring Cloud Stream provides out of the box binders for Kafka, RabbitMQ and Redis.

6.1 Classpath Detection

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

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

6.2 Multiple Binders on the Classpath

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

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

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

Selecting the binder can be done globally by either using the `spring.cloud.stream.defaultBinder` property, e.g. `spring.cloud.stream.defaultBinder=rabbit`, or by individually configuring them on each channel binding.

For instance, a processor app that reads from Kafka and writes to Rabbit can specify the following configuration: `spring.cloud.stream.bindings.input.binder=kafka`,`spring.cloud.stream.bindings.output.bind`

6.3 Connecting to Multiple Systems

By default, binders share the Spring Boot auto-configuration of the application and create one instance of each binder found on the classpath. In scenarios where an application should connect to more than one broker of the same type, Spring Cloud Stream allows you to specify multiple binder configurations, with different environment settings. Please note that turning on explicit binder configuration will disable the default binder configuration process altogether, so all the binders in use must be included in the configuration.

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

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

7. Content Type and Transformation

Spring Cloud Stream allows to propagate information about the content type of the messages it produces by attaching by default a `contentType` header to outbound messages. For middleware that does not directly support headers, Spring Cloud Stream provides its own mechanism of wrapping outbound messages in an envelope of its own, automatically. For middleware that does support headers, Spring Cloud Stream applications may receive messages with a given content type from non-Spring Cloud Stream applications.

Spring Cloud Stream can handle messages based on this information in two ways:

- through its `contentType` settings on inbound and outbound channels;
- through its argument mapping done for `@StreamListener`-annotated methods.

7.1 Type converting message channels

7.2 @StreamListener and conversion

8. Inter-app Communication

8.1 Connecting multiple application instances

While Spring Cloud Stream makes it easy for individual boot apps to connect to messaging systems, the typical scenario for Spring Cloud Stream is the creation of multi-app pipelines, where microservice apps are sending data to each other. This can be achieved by correlating the input and output destinations of adjacent apps, as in the following example.

Supposing that the design calls for the `time-source` app to send data to the `log-sink` app, we will use a common destination named `ticktock` for bindings within both apps. `time-source` will set `spring.cloud.stream.bindings.output.destination=ticktock`, and `log-sink` will set `spring.cloud.stream.bindings.input.destination=ticktock`.

8.2 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. This is done through the `spring.cloud.stream.instanceCount` and `spring.cloud.stream.instanceIndex` properties. For example, if there are 3 instances of the HDFS sink application, all three will have `spring.cloud.stream.instanceCount` set to 3, and the applications will have `spring.cloud.stream.instanceIndex` set to 0, 1 and 2, respectively. When Spring Cloud Stream applications are deployed via Spring Cloud Data Flow, these properties are configured automatically, but 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.

Setting up the two properties correctly on scale up scenarios is important for addressing partitioning behavior in general (see below), and they are always required by certain types of binders (e.g. the Kafka binder) in order to ensure that data is split correctly across multiple consumer instances.

8.3 Partitioning

Configuring Output Bindings for Partitioning

An output binding is configured to send partitioned data, by setting one and only one of its `partitionKeyExpression` or `partitionKeyExtractorClass` properties, as well as its `partitionCount` property. For example, setting `spring.cloud.stream.bindings.output.partitionKeyExpression=payload.id`,`spring.cloud.strea` is a valid and typical configuration.

Based on this configuration, the data will be sent to the target partition using the following logic. A partition key's value is calculated for each message sent to a partitioned output channel based on the `partitionKeyExpression`. The `partitionKeyExpression` is a SpEL expression 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 setting the property `partitionKeyExtractorClass`. This class must implement the interface `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy`. While, in general, the SpEL expression should suffice, more complex cases may use the custom implementation strategy.

Once the message key is calculated, the partition selection process will determine the target partition as a value between `0` and `partitionCount - 1`. The default calculation, applicable in most scenarios is based on the formula `key.hashCode() % partitionCount`. This can be customized on the binding, either by setting a SpEL expression to be evaluated against the key via the `partitionSelectorExpression` property, or by setting a `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` implementation via the `partitionSelectorClass` property.

Additional properties can be configured for more advanced scenarios, as described in the following section.

Configuring Input Bindings for Partitioning

An input binding is configured to receive partitioned data by setting its `partitioned` property, as well as the instance index and instance count properties on the app itself, as follows: `spring.cloud.stream.bindings.input.partitioned=true`,`spring.cloud.stream.instanceIndex=3`,
The instance count value represents the total number of app instances between which the data needs to be partitioned, whereas instance index must be a unique value across the multiple instances, between `0` and `instanceCount - 1`. The instance index helps each app instance to identify the unique partition (or in the case of Kafka, the partition set) from which it receives data. It is important that both values are set correctly in order to ensure that all the data is consumed, and that the app instances receive mutually exclusive datasets.

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

9. Health Indicator

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

10. Samples

For Spring Cloud Stream samples, please refer: github.com/spring-cloud/spring-cloud-stream-samples

11. Getting Started

To get started creating Spring Cloud Stream applications, head over to start.spring.io and create a new project named `GreetingSource`. Select the Spring Boot Version to be 1.3.4 (SNAPSHOT as of the time of this release) and tick the checkbox for `Stream Kafka` as we will be using Kafka for messaging. Next create a new class `GreetingSource` in the same package as the class `GreetingSourceApplication` with the following code:

```
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.integration.annotation.InboundChannelAdapter;

@EnableBinding(Source.class)
public class GreetingSource {

    @InboundChannelAdapter(Source.OUTPUT)
    public String greet() {
        return "hello world " + System.currentTimeMillis();
    }
}
```

The annotation `@EnableBinding` is what triggers the creation of Spring Integration infrastructure components. Specifically, it will create a Kafka Connection Factory, Kafka Outbound Channel Adapter, and the Message Channel defined inside the Source interface.

```
public interface Source {

  String OUTPUT = "output";

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

}
```

Furthermore, the auto configuration creates a default poller so that the greet method will be invoked once a second. The standard Spring Integration InboundChannelAdapter annotation sends a message to the source’s output channel using the return value as the payload of the message.

To test drive this setup run a Kafka Message Broker. An easy way to do this is using a docker image.

```
# on mac
docker run -p 2181:2181 -p 9092:9092 --env ADVERTISED_HOST=`docker-machine ip \`docker-machine active\``
 --env ADVERTISED_PORT=9092 spotify/kafka

# on linux
docker run -p 2181:2181 -p 9092:9092 --env ADVERTISED_HOST=localhost --env ADVERTISED_PORT=9092 spotify/
kafka
```

Build the application using `./mvnw clean package`

The consumer application is coded in a similar manner, go back to start.spring.io and create a new project named `LoggerSink`. Then create a new class `LoggingSink` in the same package as the class `LoggingSinkApplication` with the following code

```
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
```

```
    public void log(String message) {
        System.out.println(message);
    }
}
```

Build the application using `./mvnw clean package`

To connect the Source application to the Sink application, each application needs to share the same destination name. Starting up both applications as shown below you will see the consumer application printing ‘hello world’ and the timestamp to the console.

```
cd GreetingSource
java -jar target/GreetingSource-0.0.1-SNAPSHOT.jar --
spring.cloud.stream.bindings.output.destination=mydest

cd LoggingSink
java -jar target/LoggingSink-0.0.1-SNAPSHOT.jar --server.port=8090 --
spring.cloud.stream.bindings.input.destination=mydest
```

The different server port is avoid collisions of the http port used to service the boot actuator endpoints.

The output of the logging sink will look something like

```
[           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
[           main] com.example.LoggingSinkApplication       : Started LoggingSinkApplication in 6.828
 seconds (JVM running for 7.371)
hello world 1458595076731
hello world 1458595077732
hello world 1458595078733
hello world 1458595079734
hello world 1458595080735


= Appendices
[appendix]
[[building]]
== Building

:jdkversion: 1.7

=== Basic Compile and Test

To build the source you will need to install JDK {jdkversion}.

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
http://compose.docker.io/[Docker Compose] to run the middeware servers
in Docker containers. See the README in the
https://github.com/spring-cloud-samples/scripts[scripts demo
repository] for specific instructions about the common cases of mongo,
rabbit and redis.

=== Documentation

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

=== Working with the code
If you don't have an IDE preference we would recommend that you use
http://www.springsource.com/developer/sts[Spring Tools Suite] or
http://eclipse.org[Eclipse] when working with the code. We use the
http://eclipse.org/m2e/[m2eclipe] eclipse plugin for maven support. Other IDEs and tools
should also work without issue.

==== Importing into eclipse with m2eclipse
We recommend the http://eclipse.org/m2e/[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 https://github.com/spring-cloud/spring-
cloud-build/blob/master/.settings.xml[`.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:

[indent=0]
```

```
$ ./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.

=== Sign the Contributor License Agreement
Before we accept a non-trivial patch or pull request we will need you to sign the
https://support.springsource.com/spring_committer_signup[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.

=== 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
  https://github.com/spring-cloud/build/tree/master/eclipse-coding-conventions.xml[Spring
  Cloud Build] project. If using IntelliJ, you can use the
  http://plugins.jetbrains.com/plugin/6546[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 http://tbaggery.com/2008/04/19/a-note-about-git-commit-
messages.html[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).

// ==========================================================================================
```