Spring Cloud Stream Reference Guide

Brooklyn.SR2

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

Table of Contents

Part I. Spring Cloud Stream Core

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

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

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

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

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

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

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

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

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

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

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

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

  @Autowired
  private Sink sink;

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

```
}
```

2. Main Concepts

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

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

2.1 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.

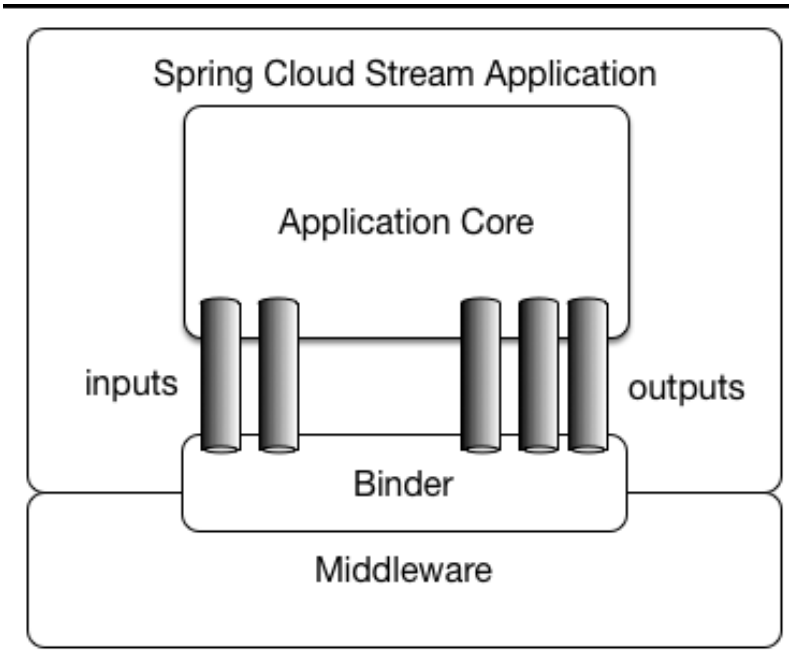

Figure 2.1. Spring Cloud Stream Application

Fat JAR

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

2.2 The Binder Abstraction

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

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

Spring Cloud Stream uses Spring Boot for configuration, and the Binder abstraction makes it possible for a Spring Cloud Stream application to be flexible in how it connects to middleware. For example, deployers can dynamically choose, at runtime, the destinations (e.g., the Kafka topics or RabbitMQ exchanges) to which channels connect. Such configuration can be provided through external configuration properties and in any form supported by Spring Boot (including application arguments, environment variables, and `application.yml` or `application.properties` files). In the sink example from the Chapter 1, *Introducing Spring Cloud Stream* section, setting the application property `spring.cloud.stream.bindings.input.destination` to `raw-sensor-data` will cause it to read from the `raw-sensor-data` Kafka topic, or from a queue bound to the `raw-sensor-data` RabbitMQ exchange.

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

2.3 Persistent Publish-Subscribe Support

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

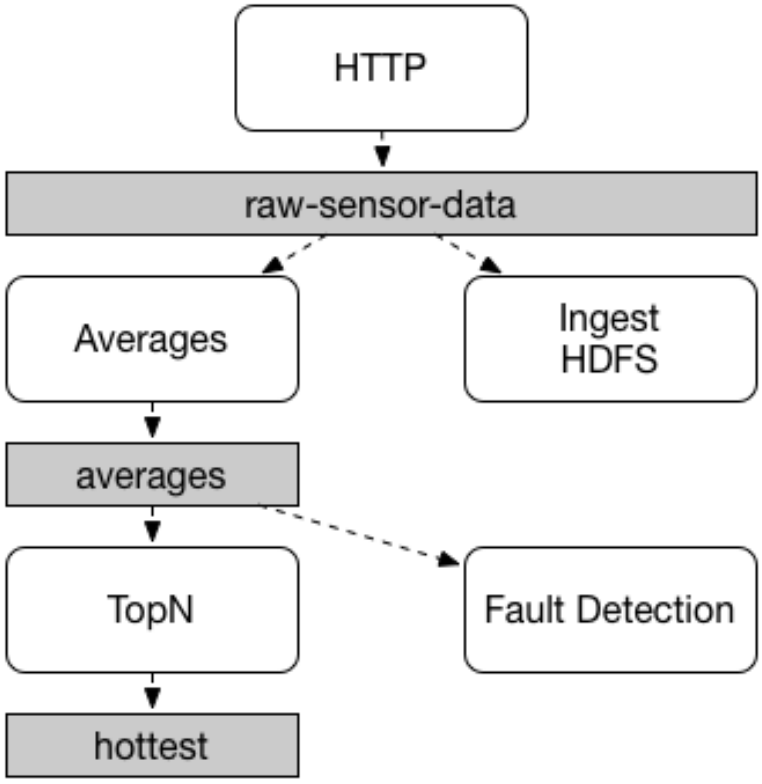

Figure 2.2. Spring Cloud Stream Publish-Subscribe

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

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

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

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

2.4 Consumer Groups

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

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

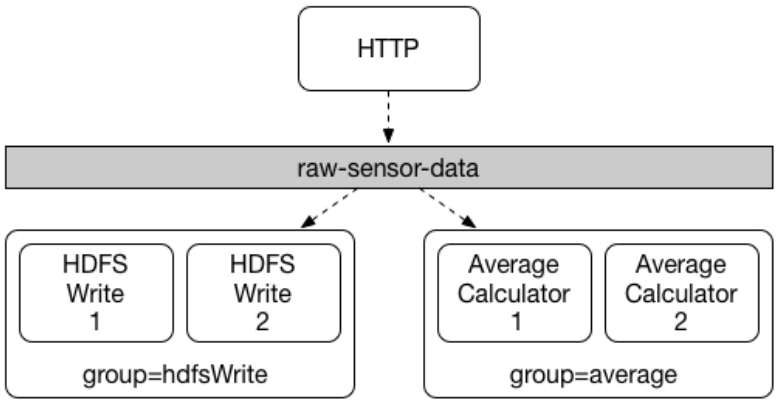

Figure 2.3. Spring Cloud Stream Consumer Groups

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

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 once at least one subscription for a group has been created, the group will receive messages, even if they are sent while all applications in the group are stopped.

Note

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

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

2.5 Partitioning Support

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

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

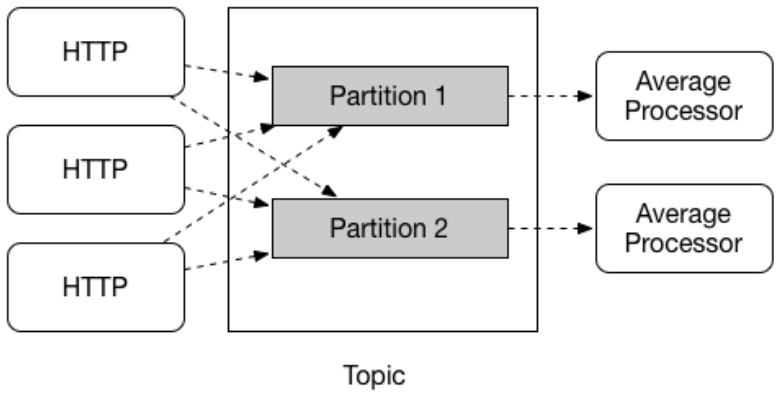

Figure 2.4. Spring Cloud Stream Partitioning

Partitioning is a critical concept in stateful processing, where it is critiical, 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.

3. Programming Model

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

3.1 Declaring and Binding Channels

Triggering Binding Via `@EnableBinding`

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

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

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

Note

In Spring Cloud Stream 1.0, the only supported bindable components are the Spring Messaging `MessageChannel` and its extensions `SubscribableChannel` and `PollableChannel`. Future versions should extend this 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 in an interface as `@Input` and `@Output` methods:

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
    MessageChannel coldDrinks();
}
```

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

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

   ...
}
```

Customizing Channel Names

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

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

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

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

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

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

```
public interface Source {

  String OUTPUT = "output";

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

}
```

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

```
public interface Sink {

  String INPUT = "input";

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

}
```

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

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

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

Accessing Bound Channels

Injecting the Bound Interfaces

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

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

```
@Component
public class SendingBean {

    private Source source;

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

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

Injecting Channels Directly

Bound channels can be also injected directly:

```
@Component
public class SendingBean {

    private MessageChannel output;

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

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

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

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

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

```
@Component
public class SendingBean {

    @Autowired @Qualifier("customOutput")
    private MessageChannel output;

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

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

Producing and Consuming Messages

You can write a Spring Cloud Stream application using either Spring Integration annotations or Spring Cloud Stream's `@StreamListener` annotation. The `@StreamListener` annotation is

modeled after other Spring Messaging annotations (such as `@MessageMapping`, `@JmsListener`, `@RabbitListener`, etc.) but adds content type management and type coercion features.

Native Spring Integration Support

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

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

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

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

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

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

Using @StreamListener for Automatic Content Type Handling

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

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

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

  @Autowired
  VotingService votingService;

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

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

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

Note

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

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

  @Autowired
  VotingService votingService;

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

Reactive Programming Support

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

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

Spring Cloud Stream supports the following reactive APIs:

- Reactor
- RxJava 1.x

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

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

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

Note

Reactive programming support requires Java 1.8.

Note

As of Spring Cloud Stream 1.1.1 and later (starting with release train Brooklyn.SR2), reactive programming support requires the use of Reactor 3.0.4.RELEASE and higher.

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

Note

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

Reactor-based handlers

A Reactor based handler can have the following argument types:

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

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

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

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

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

The same processor using output arguments looks like this:

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

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

RxJava 1.x support

RxJava 1.x handlers follow the same rules as Reactor-based one, but will use `Observable` and `ObservableSender` arguments and return types.

So the first example above will become:

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

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

The second example above will become:

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

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

Aggregation

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

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

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

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

- if the sequence starts with a source and ends with a sink, all communication between the applications is direct and no channels will be bound

- if the sequence starts with a processor, then its input channel will become the `input` channel of the aggregate and will be bound accordingly
- if the sequence ends with a processor, then its output channel will become the `output` channel of the aggregate and will be bound accordingly

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

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

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

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

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

 @Transformer
 public String loggerSink(String payload) {
  return payload.toUpperCase();
 }
}
```

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

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

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

```
@SpringBootApplication
public class SampleAggregateApplication {

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

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

Configuring aggregate application

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

The namespace can be set for applications as follows:

```
@SpringBootApplication
public class SampleAggregateApplication {

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

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

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

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

4. 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.

4.1 Producers and Consumers

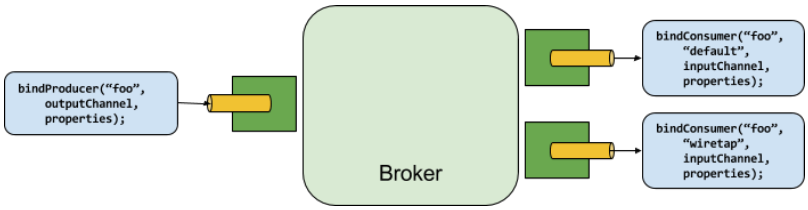

Figure 4.1. Producers and Consumers

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

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

4.2 Binder SPI

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

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

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

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

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

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

A typical binder implementation consists of the following

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

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

4.3 Binder Detection

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

Classpath Detection

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

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

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

4.4 Multiple Binders on the Classpath

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

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

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

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

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

4.5 Connecting to Multiple Systems

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

Note

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

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

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

4.6 Binder configuration properties

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

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

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

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

  Default `true`.

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

  Default `empty`.

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

  Default `true`.

5. Configuration Options

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

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

5.1 Spring Cloud Stream Properties

spring.cloud.stream.instanceCount
: The number of deployed instances of an application. Must be set for partitioning and if using Kafka.

  Default: `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 application's instance index.

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

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

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

  Default: empty.

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

  Default: false.

5.2 Binding Properties

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

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

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

Properties for Use of Spring Cloud Stream

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

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

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

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

  Default: null (indicating an anonymous consumer).

contentType
: The content type of the channel.

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

binder
: The binder used by this binding. See Section 4.4, “Multiple Binders on the Classpath” for details.

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

Consumer properties

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

Default values can be set by using the prefix `spring.cloud.stream.default.consumer`, e.g. `spring.cloud.stream.default.consumer.headerMode=raw`.

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 `raw`, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when inbound data is coming from outside Spring Cloud Stream applications.

Default: `embeddedHeaders`.

maxAttempts
: The number of attempts of re-processing an inbound message.

  Default: `3`.

backOffInitialInterval
: The backoff initial interval on retry.

  Default: `1000`.

backOffMaxInterval
: The maximum backoff interval.

  Default: `10000`.

backOffMultiplier
: The backoff multiplier.

  Default: `2.0`.

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

  Default: `-1`.

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

  Default: `-1`.

Producer Properties

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

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

partitionKeyExpression
: A SpEL expression that determines how to partition outbound data. If set, or if `partitionKeyExtractorClass` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Section 2.5, “Partitioning Support”.

  Default: null.

partitionKeyExtractorClass
: A `PartitionKeyExtractorStrategy` implementation. If set, or if `partitionKeyExpression` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to

a value greater than 1 to be effective. The two options are mutually exclusive. See Section 2.5, “Partitioning Support”.

Default: null.

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

  Default: null.

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

  Default: null.

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

  Default: `1`.

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

headerMode
: When set to `raw`, disables header embedding on output. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when producing data for non-Spring Cloud Stream applications.

  Default: `embeddedHeaders`.

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

  Default: `false`.

6. Content Type and Transformation

To allow you to propagate information about the content type of produced messages, Spring Cloud Stream attaches, by default, a `contentType` header to outbound messages. For middleware that does not directly support headers, Spring Cloud Stream provides its own mechanism of automatically wrapping outbound messages in an envelope of its own. 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 performed for methods annotated with `@StreamListener`

Spring Cloud Stream allows you to declaratively configure type conversion for inputs and outputs using the `spring.cloud.stream.bindings.<channelName>.content-type` property of a binding. Note that general type conversion may also be accomplished easily by using a transformer inside your application. Currently, Spring Cloud Stream natively supports the following type conversions commonly used in streams:

- **JSON** to/from **POJO**
- **JSON** to/from org.springframework.tuple.Tuple
- **Object** to/from **byte[]** : Either the raw bytes serialized for remote transport, bytes emitted by an application, or converted to bytes using Java serialization(requires the object to be Serializable)
- **String** to/from **byte[]**
- **Object** to **plain text** (invokes the object's *toString()* method)

Where *JSON* represents either a byte array or String payload containing JSON. Currently, Objects may be converted from a JSON byte array or String. Converting to JSON always produces a String.

6.1 MIME types

`content-type` values are parsed as media types, e.g., `application/json` or `text/plain;charset=UTF-8`. MIME types are especially useful for indicating how to convert to String or byte[] content. Spring Cloud Stream also uses MIME type format to represent Java types, using the general type `application/x-java-object` with a `type` parameter. For example, `application/x-java-object;type=java.util.Map` or `application/x-java-object;type=com.bar.Foo` can be set as the `content-type` property of an input binding. In addition, Spring Cloud Stream provides custom MIME types, notably, `application/x-spring-tuple` to specify a Tuple.

6.2 MIME types and Java types

The type conversions Spring Cloud Stream provides out of the box are summarized in the following table:

Source Payload	Target Payload	content-type header	content-type	Comments
POJO	JSON String	ignored	application/json	

Source Payload	Target Payload	content-type header	content-type	Comments
Tuple	JSON String	ignored	application/json	JSON is tailored for Tuple
POJO	String (toString())	ignored	text/plain, java.lang.String	
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object	
JSON byte[] or String	POJO	application/json (or none)	application/x-java-object	
byte[] or String	Serializable	application/x-java-serialized-object	application/x-java-object	
JSON byte[] or String	Tuple	application/json (or none)	application/x-spring-tuple	
byte[]	String	any	text/plain, java.lang.String	will apply any Charset specified in the content-type header
String	byte[]	any	application/octet-stream	will apply any Charset specified in the content-type header

Conversion applies to payloads that require type conversion. For example, if a module produces an XML string with outputType=application/json, the payload will not be converted from XML to JSON. This is because the payload at the module's output channel is already a String so no conversion will be applied at runtime.

While conversion is supported for both input and output channels, it is especially recommended to be used for the conversion of outbound messages. For the conversion of inbound messages, especially when the target is a POJO, the `@StreamListener` support will perform the conversion automatically.

6.3 Customizing message conversion

Besides the conversions that it supports out of the box, Spring Cloud Stream also supports registering your own message conversion implementations. This allows you to send and receive data in a variety of custom formats, including binary, and associate them with specific `contentTypes`. Spring Cloud Stream registers all the beans of type `org.springframework.messaging.converter.MessageConverter` as custom message converters along with the out of the box message converters.

If your message converter needs to work with a specific `content-type` and target class (for both input and output), then the message converter needs to extend `org.springframework.messaging.converter.AbstractMessageConverter`. For conversion when using `@StreamListener`, a message converter that implements `org.springframework.messaging.converter.MessageConverter` would suffice.

Here is an example of creating a message converter bean (with the content-type `application/bar`) inside a Spring Cloud Stream application:

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

  ...

  @Bean
  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 == clazz);
  }

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

6.4 Schema-based 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 is Apache Avro, with more formats to be added in future versions.

Apache Avro Message Converters

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

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

Converters with schema support

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

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

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

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

  ...

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

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

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

  ...

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

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

6.5 Schema Registry Support

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

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

Schema Registry Server

Spring Cloud Stream provides a schema registry server implementation. In order to use it, you can simply add the `spring-cloud-stream-schema-server` artifact to your project and use the `@EnableSchemaRegistryServer` annotation, adding the schema registry server REST controller to your application. This annotation is intended to be used with Spring Boot web applications, and the listening port of the server is controlled by the `server.port` setting. The `spring.cloud.stream.schema.server.path` setting can be used to control the root

path of the schema server (especially when it is embedded in other applications). The `spring.cloud.stream.schema.server.allowSchemaDeletion` boolean setting enables the deletion of schema. By default this is disabled.

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

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

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

Schema Registry Server API

The Schema Registry Server API consists of the following operations:

`POST /`

Register a new schema.

Accepts JSON payload with the following fields:

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

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

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

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

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

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

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

- `definition` the schema definition.

`GET /schemas/{id}`

Retrieve an existing schema by its id.

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

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

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

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

`DELETE /schemas/{id}`

Delete an existing schema by its id.

`DELETE /{subject}`

Delete existing schemas by their subject.

Note

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

Schema Registry Client

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

```
public interface SchemaRegistryClient {

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

 String fetch(SchemaReference schemaReference);

 String fetch(Integer id);

}
```

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

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

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

Avro Schema Registry Client Message Converters

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

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

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

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

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

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

6.6 `@StreamListener` and Message Conversion

The `@StreamListener` annotation provides a convenient way for converting incoming messages without the need to specify the content type of an input channel. During the dispatching process to methods annotated with `@StreamListener`, a conversion will be applied automatically if the argument requires it.

For example, let's consider a message with the String content `{"greeting":"Hello, world"}` and a `content-type` header of `application/json` is received on the input channel. Let us consider the following application that receives it:

```
public class GreetingMessage {

  String greeting;

  public String getGreeting() {
    return greeting;
  }

  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
```

```
}

@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class GreetingSink {

  @StreamListener(Sink.INPUT)
  public void receive(Greeting greeting) {
   // handle Greeting
  }
 }
```

The argument of the method will be populated automatically with the POJO containing the unmarshalled form of the JSON String.

7. Inter-Application Communication

7.1 Connecting Multiple Application Instances

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

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

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

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

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

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

7.2 Instance Index and Instance Count

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

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

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

7.3 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, the following is a valid and typical configuration:

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

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

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

Tip

If a SpEL expression is not sufficient for your needs, you can instead calculate the partition key value by setting the property `partitionKeyExtractorClass` to a class which implements the `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` interface. While the SpEL expression should usually 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 (with the channel name `input`) is configured to receive partitioned data by setting its `partitioned` property, as well as the `instanceIndex` and `instanceCount` properties on the application itself, as in the following example:

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

The `instanceCount` value represents the total number of application instances between which the data need to be partitioned, and the `instanceIndex` must be a unique value across the multiple instances, between `0` and `instanceCount - 1`. The instance index helps each application instance to identify the unique partition (or, in the case of Kafka, the partition set) from which it receives data. It is important to set both values correctly in order to ensure that all of the data is consumed and that the application instances receive mutually exclusive datasets.

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

8. 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`. This is useful especially for unit testing your microservices.

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

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

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

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = ExampleTest.MyProcessor.class)
@IntegrationTest({"server.port=-1"})
@DirtiesContext
public class ExampleTest {

  @Autowired
  private Processor processor;

  @Autowired
  private BinderFactory<MessageChannel> binderFactory;

  @Autowired
  private MessageCollector messageCollector;

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


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

    @Autowired
    private Processor channels;

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

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

9. Health Indicator

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

10. Samples

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

11. Getting Started

To get started with creating Spring Cloud Stream applications, visit the Spring Initializr and create a new Maven project named "GreetingSource". Select Spring Boot {supported-spring-boot-version} in the dropdown. In the *Search for dependencies* text box type `Stream Rabbit` or `Stream Kafka` depending on what binder you want to use.

Next, create a new class, `GreetingSource`, in the same package as the `GreetingSourceApplication` class. Give it 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 `@EnableBinding` annotation is what triggers the creation of Spring Integration infrastructure components. Specifically, it will create a Kafka connection factory, a Kafka outbound channel adapter, and the message channel defined inside the Source interface:

```
public interface Source {

  String OUTPUT = "output";

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

}
```

The auto-configuration also creates a default poller, so that the `greet()` method will be invoked once per 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 to use a Docker image:

```
# On OS X
$ 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:

```
./mvnw clean package
```

The consumer application is coded in a similar manner. Go back to Initializr and create another project, named LoggingSink. Then create a new class, `LoggingSink`, in the same package as the class `LoggingSinkApplication` and 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:

```
./mvnw clean package
```

To connect the GreetingSource application to the LoggingSink application, each application must share the same destination name. Starting up both applications as shown below, you will see the consumer application printing "hello world" and a 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 prevents collisions of the HTTP port used to service the Spring Boot Actuator endpoints in the two applications.)

The output of the LoggingSink application will look something like the following:

```
[           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
```

Part II. Binder Implementations

12. Apache Kafka Binder

12.1 Usage

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

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

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

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

12.2 Apache Kafka Binder Overview

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

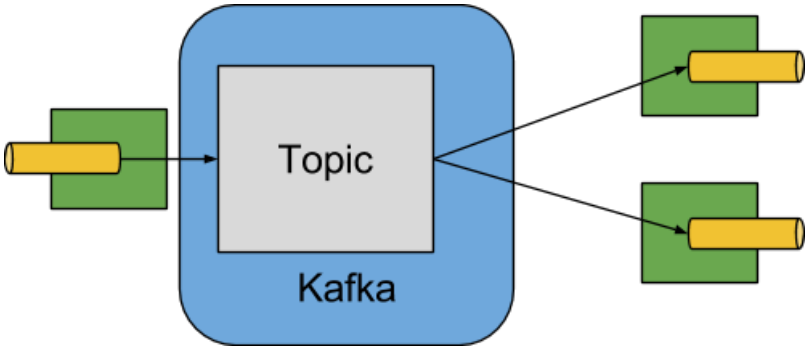

Figure 12.1. Kafka Binder

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.

12.3 Configuration Options

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

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

Kafka Binder Properties

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

  Default: `localhost`.

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

Default: `9092`.

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

Default: `localhost`.

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

Default: `2181`.

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

Default: Empty map.

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

Default: empty.

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

Default: `10000`.

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

Default: `0`.

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

Default: `1`.

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

Default: `1`.

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

Default: `1`.

spring.cloud.stream.kafka.binder.autoCreateTopics
If set to `true`, the binder will create new topics automatically. If set to `false`, the binder will rely on the topics being already configured. In the latter case, if the topics do not exist, the binder will fail

to start. Of note, this setting is independent of the `auto.topic.create.enable` setting of the broker and it does not influence it: if the server is set to auto-create topics, they may be created as part of the metadata retrieval request, with default broker settings.

Default: `true`.

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

Default: `false`.

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

Default: `2097152`.

Kafka Consumer Properties

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

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

Default: `true`.

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

Default: `true`.

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

Default: not set.

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

  Default: `5000`.

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

  Default: `false`.

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

  Default: null (equivalent to `earliest`).

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

  Default: `false`.

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

  Default: Empty map.

Kafka Producer Properties

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

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

  Default: `16384`.

sync
: Whether the producer is synchronous.

  Default: `false`.

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

  Default: `0`.

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

  Default: Empty map.

Note

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

Usage examples

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

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

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

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

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

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

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

Example: security configuration

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

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

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

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

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

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

Using JAAS configuration files

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

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

Using Spring Boot properties

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

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

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

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

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

  Default: `required`.

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

  Default: Empty map.

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

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

This represents the equivalent of the following JAAS file:

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

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

Note

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

Note

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

Using the binder with Apache Kafka 0.10

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

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

Note

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

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

The Apache Kafka Binder uses the administrative utilities which are part of the Apache Kafka server library to create and reconfigure topics. If the inclusion of the Apache Kafka server library and its

dependencies is not necessary at runtime because the application will rely on the topics being configured administratively, the Kafka binder allows for Apache Kafka server dependency to be excluded from the application.

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

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

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

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

13. RabbitMQ Binder

13.1 Usage

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

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

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

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

13.2 RabbitMQ Binder Overview

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

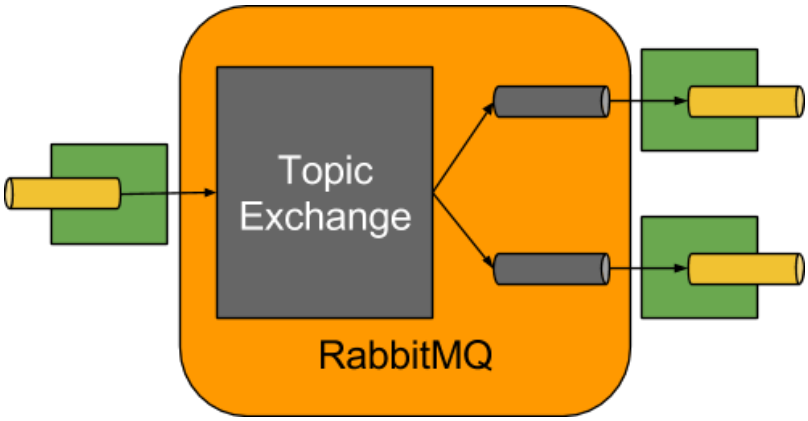

Figure 13.1. RabbitMQ Binder

The RabbitMQ Binder implementation maps each destination to a `TopicExchange`. For each consumer group, a `Queue` will be bound to that `TopicExchange`. Each consumer instance have a corresponding RabbitMQ `Consumer` instance for its group's `Queue`. For partitioned producers/consumers the queues are suffixed with the partition index and use the partition index as routing key.

Using the `autoBindDlq` option, you can optionally configure the binder to create and configure dead-letter queues (DLQs) (and a dead-letter exchange `DLX`). The dead letter queue has the name of the destination, appended with `.dlq`. If retry is enabled (`maxAttempts > 1`) failed messages will be delivered to the DLQ. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (default) so that a failed message will be routed to the DLQ, instead of being requeued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it); this enables additional information to be added to the message in headers, such as the stack trace in the `x-exception-stacktrace` header. This option does not need retry enabled; you can republish a failed message after just one attempt.

Important

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

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

The framework does not provide any standard mechanism to consume dead-letter messages (or to re-route them back to the primary queue). Some options are described in Section 13.4, "Dead-Letter Queue Processing".

13.3 Configuration Options

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

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

RabbitMQ Binder Properties

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

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

spring.cloud.stream.rabbit.binder.adminAddresses
: A comma-separated list of RabbitMQ management plugin URLs. Only used when `nodes` contains more than one entry. Each entry in this list must have a corresponding entry in `spring.rabbitmq.addresses`.

  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`.

  Default: empty.

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

  Default: `1` (BEST_LEVEL).

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`.

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: "".

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

  Default: `5000`.

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

  Default: `false`.

requestHeaderPatterns
: The request headers to be transported.

  Default: `[STANDARD_REQUEST_HEADERS,'*']`.

replyHeaderPatterns
: The reply headers to be transported.

  Default: `[STANDARD_REPLY_HEADERS,'*']`.

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

  Default: false

transacted
: Whether to use transacted channels.

  Default: `false`.

txSize
: The number of deliveries between acks.

Default: `1`.

Rabbit Producer Properties

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

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

  Default: `false`.

batchingEnabled
: Whether to enable message batching by producers.

  Default: `false`.

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

  Default: `100`.

batchBufferLimit
: Default: `10000`.

batchTimeout
: Default: `5000`.

compress
: Whether data should be compressed when sent.

  Default: `false`.

transacted
: Whether to use transacted channels.

  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_REPLY_HEADERS,'*']`.

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, that do not normally support headers).

13.4 Dead-Letter Queue Processing

Because it can't be anticipated how users would want to dispose of dead-lettered messages, the framework does not provide any standard mechanism to handle them. If the reason for the dead-lettering is transient, you may wish to route the messages back to the original queue. However, if the problem is a permanent issue, that could cause an infinite loop. The following `spring-boot` application is an example of how to route those messages back to the original queue, but moves them to a third "parking lot" queue after three attempts. The second example utilizes the RabbitMQ Delayed Message Exchange to introduce a delay to the requeued message. In this example, the delay increases for each attempt. These examples use a `@RabbitListener` to receive messages from the DLQ, you could also use `RabbitTemplate.receive()` in a batch process.

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

Non-Partitioned Destinations

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

```
@SpringBootApplication
public class ReRouteDlqApplication {

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

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

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

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

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

    @Autowired
    private RabbitTemplate rabbitTemplate;

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

    @Bean
```

```
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

```
@SpringBootApplication
public class ReRouteDlqApplication {

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

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

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

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

    private static final String DELAY_EXCHANGE = "dlqReRouter";

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

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader + 1);
            headers.put("x-delay", 5000 * retriesHeader);
            this.rabbitTemplate.send(DELAY_EXCHANGE, ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT, failedMessage);
        }
    }

    @Bean
    public DirectExchange delayExchange() {
        DirectExchange exchange = new DirectExchange(DELAY_EXCHANGE);
        exchange.setDelayed(true);
        return exchange;
    }

    @Bean
    public Binding bindOriginalToDelay() {
        return BindingBuilder.bind(new Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_QUEUE);
    }

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

}
```

Partitioned Destinations

With partitioned destinations, there is one DLQ for all partitions and we determine the original queue from the headers.

republishToDlq=false

When `republishToDlq` is `false`, RabbitMQ publishes the message to the DLX/DLQ with an `x-death` header containing information about the original destination.

```
@SpringBootApplication
public class ReRouteDlqApplication {

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

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

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

 private static final String X_DEATH_HEADER = "x-death";

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

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

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @SuppressWarnings("unchecked")
 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   List<Map<String, ?>> xDeath = (List<Map<String, ?>>) headers.get(X_DEATH_HEADER);
   String exchange = (String) xDeath.get(0).get("exchange");
   List<String> routingKeys = (List<String>) xDeath.get(0).get("routing-keys");
   this.rabbitTemplate.send(exchange, routingKeys.get(0), failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

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

}
```

republishToDlq=true

When `republishToDlq` is `true`, the republishing recoverer adds the original exchange and routing key to headers.

```
@SpringBootApplication
public class ReRouteDlqApplication {
```

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

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

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

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

 private static final String X_ORIGINAL_EXCHANGE_HEADER = RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

 private static final String X_ORIGINAL_ROUTING_KEY_HEADER =
 RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

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

 @Autowired
 private RabbitTemplate rabbitTemplate;

 @RabbitListener(queues = DLQ)
 public void rePublish(Message failedMessage) {
  Map<String, Object> headers = failedMessage.getMessageProperties().getHeaders();
  Integer retriesHeader = (Integer) headers.get(X_RETRIES_HEADER);
  if (retriesHeader == null) {
   retriesHeader = Integer.valueOf(0);
  }
  if (retriesHeader < 3) {
   headers.put(X_RETRIES_HEADER, retriesHeader + 1);
   String exchange = (String) headers.get(X_ORIGINAL_EXCHANGE_HEADER);
   String originalRoutingKey = (String) headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
   this.rabbitTemplate.send(exchange, originalRoutingKey, failedMessage);
  }
  else {
   this.rabbitTemplate.send(PARKING_LOT, failedMessage);
  }
 }

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

}
```

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).