Spring Cloud Stream RabbitMQ Binder Reference Guide

1.1.1.BUILD-SNAPSHOT

Table of Contents

Part I. Reference Guide

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

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>
```

2. RabbitMQ Binder Overview

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

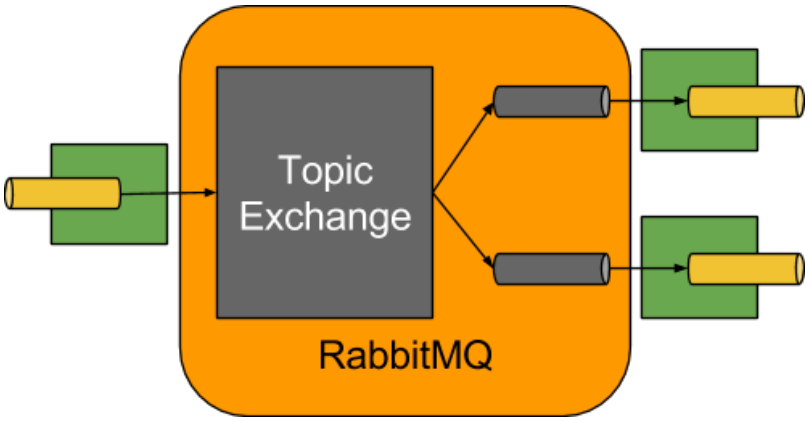

Figure 2.1. RabbitMQ Binder

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 Section 3.1, "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 Chapter 4, *Dead-Letter Queue Processing*.

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.

3.1 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).

3.2 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`.

3.3 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`.

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).

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`.

4.1 Non-Partioned 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);
    }

}
```

4.2 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 II. Appendices

Appendix A. Building

A.1 Basic Compile and Test

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

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

The main build command is

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

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

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

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

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

A.2 Documentation

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

A.3 Working with the code

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

Importing into eclipse with m2eclipse

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

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

navigate to the Spring Cloud project you imported selecting the `.settings.xml` file in that project. Click Apply and then OK to save the preference changes.

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

Importing into eclipse without m2eclipse

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

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

The generated eclipse projects can be imported by selecting `import existing projects` from the `file` menu.

Appendix B. 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.

B.1 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.

B.2 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).