

# **Spring Cloud Stream RabbitMQ Binder Reference Guide**

1.1.0.M1

Copyright © 2013-2016Pivotal Software, Inc.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

# Table of Contents

I. Reference Guide .....	1
1. Usage .....	2
2. RabbitMQ Binder Overview .....	3
3. Configuration Options .....	4
3.1. RabbitMQ Binder Properties .....	4
3.2. RabbitMQ Consumer Properties .....	4
3.3. Rabbit Producer Properties .....	5
II. Appendices .....	7
A. Building .....	8
A.1. Basic Compile and Test .....	8
A.2. Documentation .....	8
A.3. Working with the code .....	8
Importing into eclipse with m2eclipse .....	8
Importing into eclipse without m2eclipse .....	9
A.4. Sign the Contributor License Agreement .....	9
A.5. Code Conventions and Housekeeping .....	9

---

# 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:

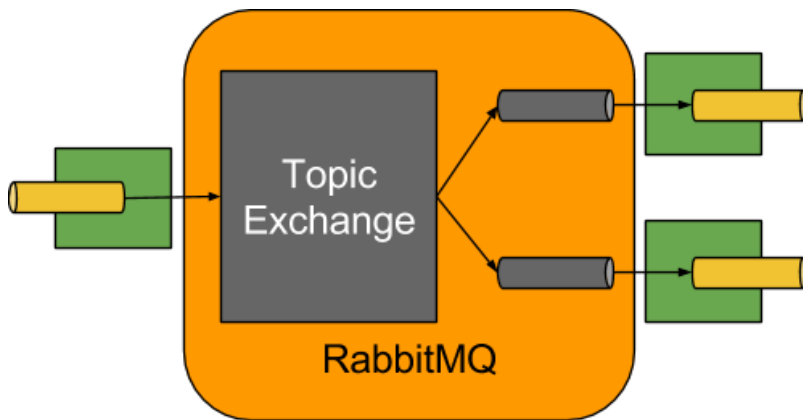
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

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 has 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.

## 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.

Default: `true`.

`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 bus will republish failed messages to the DLQ with additional headers, including the exception message and stack trace from the cause of the final failure.

`transacted`

Whether to use transacted channels.

Default: `false`.

`txSize`

The number of deliveries between acks.

Default: 1.

## 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).



---

## **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 middleware 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 [m2eclipse](#) eclipse plugin for maven support. Other IDEs and tools should also work without issue.

### Importing into eclipse with m2eclipse

We recommend the [m2eclipse](#) 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).