

Reactor RabbitMQ Reference Guide

Arnaud Cogoluègnes, Pawel Mackowski

1.4.0.BUILD-SNAPSHOT

Table of Contents

Introduction	1
1. Overview	2
1.1. RabbitMQ	2
1.2. Project Reactor	2
1.3. Reactive API for RabbitMQ	2
2. Motivation	3
2.1. Functional interface for RabbitMQ	3
2.2. Non-blocking Back-pressure	3
2.3. End-to-end Reactive Pipeline	3
2.4. Comparisons with other RabbitMQ Java libraries	3
2.4.1. RabbitMQ Java Client	3
2.4.2. Spring AMQP	3
3. Getting Started	5
3.1. Requirements	5
3.2. Quick Start	5
3.2.1. Start RabbitMQ	5
3.2.2. Run Reactor RabbitMQ Samples	5
Sample Sender	5
Sample Receiver	6
Sample Spring Boot Application	7
3.2.3. Building Reactor RabbitMQ Applications	8
4. Additional Resources	10
4.1. Getting help	10
4.2. Resources	10
5. New & Noteworthy	11
5.1. What's new in Reactor RabbitMQ 1.4	11
5.2. What's new in Reactor RabbitMQ 1.3	11
5.3. What's new in Reactor RabbitMQ 1.2	11
5.4. What's new in Reactor RabbitMQ 1.1	11
5.5. What's new in Reactor RabbitMQ 1.0	11
Reference Documentation	12
6. Reactor RabbitMQ API	13
6.1. Overview	13
6.2. Reactive RabbitMQ Sender	13
6.2.1. Managing resources (exchanges, queues, and bindings)	14
6.2.2. Reliable publishing with publisher confirms	15
6.2.3. Threading model	17
6.2.4. Closing the Sender	17

6.2.5. Error handling during publishing	18
6.2.6. Request/reply	18
6.3. Reactive RabbitMQ Receiver	20
6.3.1. Consuming options	21
6.3.2. Acknowledgment	21
6.3.3. Closing the Receiver	21
6.3.4. Connection failure	22
6.4. Advanced features	23
6.4.1. Customizing connection creation	23
Creating a connection with a supplier	23
Creating a connection with a custom Mono	24
6.4.2. Sharing the same connection between Sender and Receiver	25
6.4.3. Creating channels with a custom Mono in Sender	25
6.4.4. Threading considerations for resource management	25
6.4.5. Channel pooling in Sender	26
6.4.6. Retry configuration on connection opening	27

Introduction

Chapter 1. Overview

1.1. RabbitMQ

With more than 35,000 production deployments world-wide at small startups and large enterprises, [RabbitMQ](#) is the most popular open source message broker.

RabbitMQ is lightweight and easy to deploy on premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

1.2. Project Reactor

[Reactor](#) is a highly optimized reactive library for building efficient, non-blocking applications on the JVM based on the [Reactive Streams Specification](#). Reactor based applications can sustain very high throughput message rates and operate with a very low memory footprint, making it suitable for building efficient event-driven applications using the microservices architecture.

Reactor implements two publishers [Flux<T>](#) and [Mono<T>](#), both of which support non-blocking back-pressure. This enables exchange of data between threads with well-defined memory usage, avoiding unnecessary intermediate buffering or blocking.

1.3. Reactive API for RabbitMQ

[Reactor RabbitMQ](#) is a reactive API for RabbitMQ based on Reactor and [RabbitMQ Java Client](#). Reactor RabbitMQ API enables messages to be published to RabbitMQ and consumed from RabbitMQ using functional APIs with non-blocking back-pressure and very low overheads. This enables applications using Reactor to use RabbitMQ as a message bus or streaming platform and integrate with other systems to provide an end-to-end reactive pipeline.

Chapter 2. Motivation

2.1. Functional interface for RabbitMQ

Reactor RabbitMQ is a functional Java API for RabbitMQ. For applications that are written in functional style, this API enables RabbitMQ interactions to be integrated easily without requiring non-functional produce or consume APIs to be incorporated into the application logic.

2.2. Non-blocking Back-pressure

The Reactor RabbitMQ API benefits from non-blocking back-pressure provided by Reactor. For example, in a pipeline, where messages received from an external source (e.g. an HTTP proxy) are published to RabbitMQ, back-pressure can be applied easily to the whole pipeline, limiting the number of messages in-flight and controlling memory usage. Messages flow through the pipeline as they are available, with Reactor taking care of limiting the flow rate to avoid overflow, keeping application logic simple.

2.3. End-to-end Reactive Pipeline

The value proposition for Reactor RabbitMQ is the efficient utilization of resources in applications with multiple external interactions where RabbitMQ is one of the external systems. End-to-end reactive pipelines benefit from non-blocking back-pressure and efficient use of threads, enabling a large number of concurrent requests to be processed efficiently. The optimizations provided by Project Reactor enable development of reactive applications with very low overheads and predictable capacity planning to deliver low-latency, high-throughput pipelines.

2.4. Comparisons with other RabbitMQ Java libraries

Reactor RabbitMQ is not intended to replace any of the existing Java libraries. Instead, it is aimed at providing an alternative API for reactive event-driven applications.

2.4.1. RabbitMQ Java Client

For non-reactive applications, [RabbitMQ Java Client](#) provides the most complete API to manage resources, publish messages to and consume messages from RabbitMQ. Note Reactor RabbitMQ is based on RabbitMQ Java Client.

Applications using RabbitMQ as a message bus using this API may consider switching to Reactor RabbitMQ if the application is implemented in a functional style.

2.4.2. Spring AMQP

[Spring AMQP](#) applies core [Spring Framework](#) concepts to the development of AMQP-based messaging solutions. It provides a "template" as a high-level abstraction for sending and receiving messages. It also provides support for Message-driven POJOs with a "listener container". These libraries facilitate management of AMQP resources while promoting the use of dependency

injection and declarative configuration. Spring AMQP is based on RabbitMQ Java Client.

Chapter 3. Getting Started

3.1. Requirements

You need Java JRE installed (Java 8 or later).

You also need to install RabbitMQ. Follow the [instructions from the website](#). Note you should use RabbitMQ 3.6.x or later.

3.2. Quick Start

This quick start tutorial sets up a single node RabbitMQ and runs the sample reactive sender and consumer.

3.2.1. Start RabbitMQ

Start RabbitMQ on your local machine with all the defaults (e.g. AMQP port is 5672).

3.2.2. Run Reactor RabbitMQ Samples

Download Reactor RabbitMQ from github.com/reactor/reactor-rabbitmq/.

```
> git clone https://github.com/reactor/reactor-rabbitmq
> cd reactor-rabbitmq
```

Sample Sender

The **SampleSender** code is on GitHub.

Run the sample sender:


```
> ./gradlew -q sender
10:20:12.590 INFO r.rabbitmq.samples.SampleSender - Message Message_1 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_2 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_3 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_4 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_5 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_6 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_7 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_8 sent
successfully
10:20:12.596 INFO r.rabbitmq.samples.SampleSender - Message Message_9 sent
successfully
10:20:12.597 INFO r.rabbitmq.samples.SampleSender - Message Message_10 sent
successfully
10:20:12.597 INFO r.rabbitmq.samples.SampleSender - Message Message_11 sent
successfully
10:20:12.597 INFO r.rabbitmq.samples.SampleSender - Message Message_12 sent
successfully
10:20:12.599 INFO r.rabbitmq.samples.SampleSender - Message Message_13 sent
successfully
10:20:12.600 INFO r.rabbitmq.samples.SampleSender - Message Message_14 sent
successfully
10:20:12.600 INFO r.rabbitmq.samples.SampleSender - Message Message_15 sent
successfully
10:20:12.600 INFO r.rabbitmq.samples.SampleSender - Message Message_16 sent
successfully
10:20:12.600 INFO r.rabbitmq.samples.SampleSender - Message Message_17 sent
successfully
10:20:12.600 INFO r.rabbitmq.samples.SampleSender - Message Message_18 sent
successfully
10:20:12.601 INFO r.rabbitmq.samples.SampleSender - Message Message_19 sent
successfully
10:20:12.601 INFO r.rabbitmq.samples.SampleSender - Message Message_20 sent
successfully
```

The `SampleSender` sends 20 messages to the `demo-queue` queue, with publisher confirms enabled. The log line for a given message is printed to the console when the publisher confirmation is received from the broker.

Sample Receiver

The `SampleReceiver` code is on [GitHub](#).

Run the sample receiver:

```
> ./gradlew -q receiver
10:22:43.568 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_1
10:22:43.575 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_2
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_3
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_4
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_5
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_6
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_7
10:22:43.576 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_8
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_9
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_10
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_11
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_12
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_13
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_14
10:22:43.577 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_15
10:22:43.578 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_16
10:22:43.578 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_17
10:22:43.578 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_18
10:22:43.578 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_19
10:22:43.578 INFO r.rabbitmq.samples.SampleReceiver - Received message Message_20
```

The `SampleReceiver` consumes messages from the `demo-queue` queue and logs the message content in the console.

Sample Spring Boot Application

The `SpringBootSample` code is on GitHub.

Run the sample Spring Boot application:

```
> ./gradlew -q springboot
...
11:47:43.837 INFO r.rabbitmq.samples.SpringBootSample - Sending messages...
11:47:43.846 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_1
11:47:43.849 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_2
11:47:43.850 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_3
11:47:43.850 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_4
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_5
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_6
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_7
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_8
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_9
11:47:43.851 INFO r.rabbitmq.samples.SpringBootSample - Received message Message_10
```

The Spring Boot sample publishes messages with a `Sender` and consumes them with a `Receiver`. This application illustrates how to configure Reactor RabbitMQ in a Spring Boot environment.

3.2.3. Building Reactor RabbitMQ Applications

To build your own application using the Reactor RabbitMQ API, you need to include a dependency to Reactor RabbitMQ.

For Gradle:

```
dependencies {  
    compile "io.projectreactor.rabbitmq:reactor-rabbitmq:1.4.0.BUILD-SNAPSHOT"  
}
```

For Maven:

```
<dependency>  
    <groupId>io.projectreactor.rabbitmq</groupId>  
    <artifactId>reactor-rabbitmq</artifactId>  
    <version>1.4.0.BUILD-SNAPSHOT</version>  
</dependency>
```

When using a **milestone** or a **release candidate**, you need to add the Spring IO milestone repository.

For Gradle:

```
repositories {  
    maven { url 'https://repo.spring.io/milestone' }  
    mavenCentral()  
}
```

For Maven:

```
<repositories>  
    <repository>  
        <id>spring-milestones</id>  
        <name>Spring Milestones</name>  
        <url>https://repo.spring.io/milestone</url>  
        <snapshots>  
            <enabled>false</enabled>  
        </snapshots>  
    </repository>  
</repositories>
```

When using a **snapshot**, you need to add the Spring IO snapshots repository.

For Gradle:

```
repositories {  
    maven { url 'https://repo.spring.io/libs-snapshot' }  
    mavenCentral()  
}
```

For Maven:

```
<repositories>  
  <repository>  
    <id>spring-snapshots</id>  
    <name>Spring Snapshots</name>  
    <url>https://repo.spring.io/libs-snapshot</url>  
    <snapshots>  
      <enabled>true</enabled>  
    </snapshots>  
  </repository>  
</repositories>
```

Chapter 4. Additional Resources

4.1. Getting help

If you are having trouble with Reactor RabbitMQ, you can ask for help on [RabbitMQ community mailing list](#).

Report bugs in Reactor RabbitMQ at github.com/reactor/reactor-rabbitmq/issues.

Reactor Rabbitmq is open source and the code and documentation are available at github.com/reactor/reactor-rabbitmq.

4.2. Resources

- [Reactor RabbitMQ on github](#)
- [RabbitMQ](#)
- [Project Reactor](#)
- [Reactor Core](#)
- [Reactive Streams Specification](#)
- [Understanding Reactive types](#)
- [Lite Rx API Hands-on](#)
- [Reactor by Example](#)

Chapter 5. New & Noteworthy

5.1. What's new in Reactor RabbitMQ 1.4

- Add `@NonNullApi` and `@Nullable` annotations
- Add `Sender#sendWithTypedPublishConfirms` to be able to publish `OutboundMessage` instances of a custom class and get them back in the flux of `OutboundMessageResult`

5.2. What's new in Reactor RabbitMQ 1.3

- Use Reactor 3.3.0.RELEASE
- Allow passive exchange and queue declaration
- Emit exception on server-initiated channel closing in publish confirms flow
- Add support for handling returned (undeliverable) messages in `Sender`
- Add hook to configure `Mono<Connection>`
- Support client-generated consumer tags
- Cache connections and channels only on success
- Use Java client 5.7.3

5.3. What's new in Reactor RabbitMQ 1.2

- Limit in-flight records in publisher confirms if requested
- Implement back pressure in publisher confirms support
- Use Reactor 3.2.8.RELEASE

5.4. What's new in Reactor RabbitMQ 1.1

- Let user provide `Mono<Channel>` for sending messages
- Add optional channel pooling for sending messages
- Automatically retry on ack and nack
- Use Reactor 3.2.5.RELEASE
- Use Java client 5.6.0

5.5. What's new in Reactor RabbitMQ 1.0

- Introduction of the `Sender` and `Receiver` API
- Support for request/reply
- Exception handling
- Let user provide `Mono<Channel>` for resource management

- Complete receiving flux on channel termination
- Handle error signal of `connectionMono` subscription to enable proper error handling
- Use Reactor 3.2.3.RELEASE
- Use Java client 5.5.1

Reference Documentation

Chapter 6. Reactor RabbitMQ API

6.1. Overview

This section describes the reactive API for producing and consuming messages using RabbitMQ. There are two main classes in Reactor RabbitMQ:

1. `reactor.rabbitmq.Sender` for publishing messages to RabbitMQ
2. `reactor.rabbitmq.Receiver` for consuming messages from RabbitMQ

Full API for Reactor RabbitMQ is available in the [javadocs](#).

The project uses [Reactor Core](#) to expose a "Reactive Streams" API.

6.2. Reactive RabbitMQ Sender

Outbound messages are sent to RabbitMQ using `reactor.rabbitmq.Sender`. A `Sender` is associated with one RabbitMQ `Connection` that is used to transport messages to the broker. A `Sender` can also manage resources (exchanges, queues, bindings).

A `Sender` is created with an instance of sender configuration options `reactor.rabbitmq.SenderOptions`. The properties of `SenderOptions` contains the `ConnectionFactory` that creates connections to the broker and a Reactor `Scheduler` used by the `Sender`.

```
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.useNio();

SenderOptions senderOptions = new SenderOptions()
    .connectionFactory(connectionFactory)           ①
    .resourceManagementScheduler(Schedulers.elastic()); ②
```

① Specify connection factory

② Specify scheduler for resource management

Note you can control the creation of the `Connection` thanks to the `connectionSupplier(ConnectionFactory)` method:

```
SenderOptions senderOptions = new SenderOptions()
    .connectionFactory(connectionFactory)
    .connectionSupplier(cf -> cf.newConnection(           ①
        new Address[] {new Address("192.168.0.1"), new Address("192.168.0.2")},
        "reactive-sender"))
    .resourceManagementScheduler(Schedulers.elastic());
```

① Specify array of addresses and connection name

In the snippet above the connection can be created from 2 different nodes (useful for failover) and

the connection name is set up.

Once the required options have been configured on the options instance, a new `Sender` instance can be created with the options already configured in `senderOptions`.

```
Sender sender = RabbitFlux.createSender(senderOptions);
```

The `Sender` is now ready to send messages to RabbitMQ. At this point, a `Sender` instance has been created, but no connections to RabbitMQ have been made yet. The underlying `Connection` instance is created lazily when a first call is made to create a resource or to send messages.

Let's now create a sequence of messages to send to RabbitMQ. Each outbound message to be sent to RabbitMQ is represented as a `OutboundMessage`. An `OutboundMessage` contains routing information (exchange to send to and routing key) as well as the message itself (properties and body).

A `Flux<OutboundMessage>` of messages is created for sending to RabbitMQ. For beginners, [Lite Rx API Hands-on](#) provides a hands-on tutorial on using the Reactor classes `Flux` and `Mono`.

```
Flux<OutboundMessage> outboundFlux =  
    Flux.range(1, 10)  
        .map(i -> new OutboundMessage(  
            "amq.direct",  
            "routing.key", ("Message " + i).getBytes()  
        ));
```

The code segment above creates a sequence of messages to send to RabbitMQ. The outbound Flux can now be sent to RabbitMQ using the `Sender` created earlier.

The code segment below sends the messages to RabbitMQ. The final `subscribe()` in the code block requests upstream to send the messages to RabbitMQ.

```
sender.send(outboundFlux)           ①  
    .doOnError(e -> log.error("Send failed", e)) ②  
    .subscribe();                   ③
```

- ① Reactive send operation for the outbound Flux
- ② If the sending fails, log an error
- ③ Subscribe to trigger the actual flow of records from `outboundFlux` to RabbitMQ.

See [SampleSender](#) for a full code listing for a `Sender`.

6.2.1. Managing resources (exchanges, queues, and bindings)

The `Sender` is also able to declare and delete AMQP resources the reactive way. You can learn more about the [AMQP model on RabbitMQ website](#).

`Sender` has a `declare*` method for each type of resource (exchange, binding, and queue) and there's

also a respective `*Specification` class to describe each creation.

```
Mono<AMQP.Exchange.DeclareOk> exchange = sender.declareExchange(
    ExchangeSpecification.exchange("my.exchange")
);
Mono<AMQP.Queue.DeclareOk> queue = sender.declareQueue(
    QueueSpecification.queue("my.queue")
);
Mono<AMQP.Queue.BindOk> binding = sender.bind(
    BindingSpecification.binding().exchange("my.exchange")
    .queue("my.queue").routingKey("a.b")
);
```

Note the `Sender#declare*` methods return their respective AMQP results wrapped into a `Mono`.



For queue creation, note that if a queue specification has a null name, the queue to be created will have a server-generated name and will be non-durable, exclusive, and auto-delete. If you want a queue to have a server-generated name but other parameters, specify an empty name `""` and set the parameters accordingly on the `QueueSpecification` instance. For more information about queues, see the [official documentation](#).

One can also use the `ResourcesSpecification` factory class with a static import to reduce boilerplate code. Combined with `Mono` chaining and `Sender#declare` shortcuts, it allows for condensed syntax:

```
import static reactor.rabbitmq.ResourcesSpecification.*;
...
sender.declare(exchange("my.exchange"))
    .then(sender.declare(queue("my.queue")))
    .then(sender.bind(binding("my.exchange", "a.b", "my.queue")))
    .subscribe(r -> System.out.println("Exchange and queue declared and bound"));
```

`Sender` has `delete*` and `delete` methods as well. Here is an example with the short method forms:

```
import static reactor.rabbitmq.ResourcesSpecification.*;
...
sender.unbind(binding("my.exchange", "a.b", "my.queue"))
    .then(sender.delete(exchange("my.exchange")))
    .then(sender.delete(queue("my.queue")))
    .subscribe(r -> System.out.println("Exchange and queue unbound and deleted"));
```

6.2.2. Reliable publishing with publisher confirms

`Sender` offers also the `sendWithPublishConfirms` method to send messages and receive `publisher confirms` to make sure the broker has taken into account the outbound messages.

```

Flux<OutboundMessage> outboundFlux = Flux.range(1, 10)
    .map(i -> new OutboundMessage(
        "amq.direct",
        "routing.key", "hello".getBytes()
    ));
sender.sendWithPublishConfirms(outboundFlux)
    .subscribe(outboundMessageResult -> {
        if (outboundMessageResult.isAck()) {
            // ①
        }
    });

```

① Outbound message has reached the broker

`Sender#sendWithPublishConfirms` returns a `Flux<OutboundMessageResult>` that can be subscribed to to know that outbound messages have successfully reached the broker.

It is also possible to know about `unroutable messages`, that is messages not routed to any queue because they do not match any routing rule. Tracking of unroutable messages is disabled by default, it can be enabled by using the `Sender#sendWithPublishConfirms(Publisher<OutboundMessage>, SendOptions)` method and set the `trackReturned` flag on `SendOptions`:

```

Flux<OutboundMessage> outboundFlux = Flux.range(1, 10)
    .map(i -> new OutboundMessage(
        "amq.direct",
        "routing.key", "hello".getBytes()
    ));
sender.sendWithPublishConfirms(outboundFlux, new SendOptions().trackReturned(true))
    .subscribe(outboundMessageResult -> {
        // ①
        if (outboundMessageResult.isReturned()) {
            // ②
        }
    });

```

① Track unroutable messages

② Outbound message was not routed to any queue

The `OutboundMessageResult#isReturned` method then tells whether a message has been routed somewhere or not. This method always returns `false` if unroutable messages are not tracked. Note the `OutboundMessageResult#isAck` method returns `true` for unroutable messages, because the broker considered they have been taken care of (i.e. confirmed). So if you are interested in unroutable messages, the returned status should always be checked **before** the confirmed status.

Note it is possible to publish `OutboundMessage` instances of a custom class and get them back in the flux of `OutboundMessageResult`. To do so, use the `sendWithTypedPublishConfirms` method:

```

Flux<CorrelableOutboundMessage<Integer>> outboundFlux = Flux.range(1, 10)
    .map(i -> new CorrelableOutboundMessage<>(
①
        "amq.direct",
        "routing.key", "hello".getBytes(),
        i
②
    ));
sender.sendWithTypedPublishConfirms(outboundFlux)
    .subscribe(confirmation -> {
        CorrelableOutboundMessage<Integer> message = confirmation.getOutboundMessage(
); ③
        Integer confirmedBusinessData = message.getCorrelationMetadata();
④
    });

```

- ① Use a subclass of `OutboundMessage`
- ② Provide some extra information in the outbound message
- ③ Get the original message
- ④ Retrieve the extra information

The previous sample uses the provided `CorrelableOutboundMessage` class, but it could be any subclass of `OutboundMessage`. The `OutboundMessageResult` instances of the confirmation flux are typed with the original message class, so the extra information is available. This allows to perform any useful processing on this extra information when the outbound message is confirmed.

6.2.3. Threading model

Reactor RabbitMQ configure by default the Java Client to use NIO, i.e. there's only one thread that deals with IO. This can be changed by specifying a `ConnectionFactory` in the `SenderOptions`.

The `Sender` uses 2 Reactor's `Scheduler`: one for the subscription when creating the connection and another one for resources management. The `Sender` defaults to 2 elastic schedulers, this can be overridden in the `SenderOptions`. The `Sender` takes care of disposing the default schedulers when closing. If not using the default schedulers, it's developer's job to dispose schedulers they passed in to the `SenderOptions`.

6.2.4. Closing the `Sender`

When the `Sender` is no longer required, the instance can be closed. The underlying `Connection` is closed, as well as the default schedulers if none has been explicitly provided.

```
sender.close();
```

6.2.5. Error handling during publishing

The `send` and `sendWithPublishConfirms` methods can take an additional `SendOptions` parameter to specify the behavior to adopt if the publishing of a message fails. The default behavior is to retry every 200 milliseconds for 10 seconds in case of connection failure. As [automatic connection recovery](#) is enabled by default, the connection is likely to be re-opened after a network glitch and the flux of outbound messages should stall only during connection recovery before restarting automatically. This default behavior tries to find a trade-off between reactivity and robustness.

You can customize the retry by settings your own instance of `RetrySendingExceptionHandler` in the `SendOptions`, e.g. to retry for 20 seconds every 500 milliseconds:

```
Sender sender = RabbitFlux.createSender();
sender.send(outboundFlux, new SendOptions().exceptionHandler(
    new ExceptionHandlers.RetrySendingExceptionHandler(
        Duration.ofSeconds(20), Duration.ofMillis(500),
        ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE
    )
));
```

The `RetrySendingExceptionHandler` uses a `Predicate<Throwable>` to decide whether an exception should trigger a retry or not. If the exception isn't retryable, the exception handler wraps the exception in a `RabbitFluxException` and throws it.

For consistency sake, the retry exception handler used with `ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE` (the default) will trigger retry attempts for the same conditions as connection recovery triggering. This means that if connection recovery has kicked in, publishing will be retried at least for the retry timeout configured (10 seconds by default).

Note the exception handler is a `BiConsumer<Sender.SendContext, Exception>`, where `Sender.SendContext` is a class providing access to the `OutboundMessage` and the underlying AMQP `Channel`. This makes it easy to customize the default behavior: logging `BiConsumer#andThen` retrying, only logging, trying to send the message somewhere else, etc.

6.2.6. Request/reply

Reactor RabbitMQ supports reactive request/reply. From RabbitMQ documentation:

RPC (request/reply) is a popular pattern to implement with a messaging broker like RabbitMQ. [...] The typical way to do this is for RPC clients to send requests that are routed to a long lived (known) server queue. The RPC server(s) consume requests from this queue and then send replies to each client using the queue named by the client in the reply-to header.

For performance reason, Reactor RabbitMQ builds on top [direct reply-to](#). The next snippet shows the usage of the `RpcClient` class:

```
String queue = "rpc.server.queue";
Sender sender = RabbitFlux.createSender();
RpcClient rpcClient = sender.rpcClient("", queue); ①
Mono<Delivery> reply = rpcClient.rpc(Mono.just(
    new RpcClient.RpcRequest("hello".getBytes()) ②
));
rpcClient.close(); ③
```

① Create `RpcClient` instance from a `Sender`

② Send request and get reply

③ Close `RpcClient` when done

In the example above, a consumer waits on the `rpc.server.queue` to process requests. A `RpcClient` is created from a `Sender`, it will send requests to a given exchange with a given routing key. The `RpcClient` handles the machinery to send the request and wait on a reply queue the result processed on the server queue, wrapping everything up with reactive API. Note a RPC client isn't meant to be used for only 1 request, it can be a long-lived object handling different requests, as long as they're directed to the same destination (defined by the exchange and the routing key passed in when the `RpcClient` is created).

A `RpcClient` uses a sequence of `Long` for correlation, but this can be changed by passing in a `Supplier<String>` when creating the `RpcClient`:

```
String queue = "rpc.server.queue";
Supplier<String> correlationIdSupplier = () -> UUID.randomUUID().toString(); ①
Sender sender = RabbitFlux.createSender();
RpcClient rpcClient = sender.rpcClient(
    "", queue, correlationIdSupplier ②
);
Mono<Delivery> reply = rpcClient.rpc(Mono.just(
    new RpcClient.RpcRequest("hello".getBytes())
));
rpcClient.close();
```

① Use random UUID correlation ID supplier

② Pass in supplier on `RpcClient` creation

This can be useful e.g. when the RPC server can make sense of the correlation ID.



`RpcClient#close()` does not close the underlying `Channel` the `RpcClient` uses. When creating the `RpcClient` with `Sender#rpcClient` the `Sender` instance provides a `Channel` that will be closed when the `Sender` is closed. It is possible to provide a given `Mono<Channel>` by using the `RpcClient` constructor, but the `Channel` will then need to be explicitly closed as well.

6.3. Reactive RabbitMQ Receiver

Messages stored in RabbitMQ queues are consumed using the reactive receiver `reactor.rabbitmq.Receiver`. Each instance of `Receiver` is associated with a single instance of `Connection` created by the options-provided `ConnectionFactory`.

A receiver is created with an instance of receiver configuration options `reactor.rabbitmq.ReceiverOptions`. The properties of `ReceiverOptions` contains the `ConnectionFactory` that creates connections to the broker and a Reactor `Scheduler` used for the connection creation.

```
ConnectionFactory connectionFactory = new ConnectionFactory();
connectionFactory.useNio();

ReceiverOptions receiverOptions = new ReceiverOptions()
    .connectionFactory(connectionFactory)           ①
    .connectionSubscriptionScheduler(Schedulers.elastic()); ②
```

① Specify connection factory

② Specify scheduler for connection creation

Note you can control the creation of the `Connection` thanks to the `connectionSupplier(ConnectionFactory)` method:

```
SenderOptions senderOptions = new SenderOptions()
    .connectionFactory(connectionFactory)
    .connectionSupplier(cf -> cf.newConnection(           ①
        new Address[] {new Address("192.168.0.1"), new Address("192.168.0.2")},
        "reactive-sender"))
    .resourceManagementScheduler(Schedulers.elastic());
```

① Specify array of addresses and connection name

In the snippet above the connection can be created from 2 different nodes (useful for failover) and the connection name is set up.

Once the required configuration options have been configured on the options instance, a new `Receiver` instance can be created with these options to consume inbound messages. The code snippet below creates a receiver instance and an inbound `Flux` for the receiver. The underlying `Connection` and `Consumer` instances are created lazily later when the inbound `Flux` is subscribed to.

```
Flux<Delivery> inboundFlux = RabbitFlux.createReceiver(receiverOptions)
    .consumeNoAck("reactive.queue");
```

The inbound RabbitMQ `Flux` is ready to be consumed. Each inbound message delivered by the Flux is represented as a `Delivery`.

See `SampleReceiver` for a full code listing for a `Receiver`.

6.3.1. Consuming options

The `Receiver` class has different flavors of the `receive*` method and each of them can accept a `ConsumeOptions` instance. Here are the different options:

- `overflowStrategy`: the `OverflowStrategy` used when creating the `Flux` of messages. Default is `BUFFER`.
- `qos`: the prefetch count used when message acknowledgment is enabled. Default is 250.
- `consumerTag`: Consumer tag used to register the consumer. Default is server-generated identifier.
- `hookBeforeEmitBiFunction`: a `BiFunction<Long, ? super Delivery, Boolean>` to decide whether a message should be emitted downstream or not. Default is to always emit.
- `stopConsumingBiFunction`: a `BiFunction<Long, ? super Delivery, Boolean>` to decide whether the flux should be completed after the emission of the message. Default is to never complete.

6.3.2. Acknowledgment

`Receiver` has several `receive*` methods that differ on the way consumer are acknowledged back to the broker. Acknowledgment mode can have profound impacts on performance and memory consumption.

- `consumeNoAck`: the broker forgets about a message as soon as it has sent it to the consumer. Use this mode if downstream subscribers are very fast, at least faster than the flow of inbound messages. Messages will pile up in the JVM process memory if subscribers are not able to cope with the flow of messages, leading to out-of-memory errors. Note this mode uses the auto-acknowledgment mode when registering the RabbitMQ `Consumer`.
- `consumeAutoAck`: with this mode, messages are acknowledged right after their arrival, in the `Flux#doOnNext` callback. This can help to cope with the flow of messages, avoiding the downstream subscribers to be overwhelmed. Note this mode **does not use** the auto-acknowledgment mode when registering the RabbitMQ `Consumer`. In this case, `consumeAutoAck` means messages are automatically acknowledged by the library in one the `Flux` hooks.
- `consumeManualAck`: this method returns a `Flux<AcknowledgableDelivery>` and messages must be manually acknowledged or rejected downstream with `AcknowledgableDelivery#ack` or `AcknowledgableDelivery#nack`, respectively. This mode lets the developer acknowledge messages in the most efficient way, e.g. by acknowledging several messages at the same time with `AcknowledgableDelivery#ack(true)` and letting Reactor control the batch size with one of the `Flux#buffer` methods.

To learn more on how the `ConsumeOptions#qos` setting can impact the behavior of `Receiver#consumeAutoAck` and `Receiver#consumeManualAck`, have a look at [this post about queuing theory](#).

6.3.3. Closing the `Receiver`

When the `Receiver` is no longer required, the instance can be closed. The underlying `Connection` is closed, as well as the default scheduler if none has been explicitly provided.


```
receiver.close();
```

6.3.4. Connection failure

Network connection between the broker and the client can fail. This is transparent for consumers thanks to RabbitMQ Java client [automatic connection recovery](#). Connection failures affect sending though, and acknowledgment is a sending operation.

When using `Receiver#consumeAutoAck`, acknowledgments are retried for 10 seconds every 200 milliseconds in case of connection failure. This can be changed by setting the `BiConsumer<Receiver.AcknowledgmentContext, Exception> exceptionHandler` in the `ConsumeOptions`, e.g. to retry for 20 seconds every 500 milliseconds:

```
Flux<Delivery> inboundFlux = RabbitFlux
    .createReceiver()
    .consumeAutoAck("reactive.queue", new ConsumeOptions()
        .exceptionHandler(new ExceptionHandlers.RetryAcknowledgmentExceptionHandler(
            Duration.ofSeconds(20), Duration.ofMillis(500), ①
            ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE
        ))
    );
```

① Retry acknowledgment for 20 seconds every 500 milliseconds on connection failure

When using `Receiver#consumeManualAck`, acknowledgment is handled by the developer, who can do pretty anything they want on acknowledgment failure.

`AcknowledgableDelivery#ack` and `AcknowledgableDelivery#nack` methods handle retry internally based on `BiConsumer<Receiver.AcknowledgmentContext, Exception> exceptionHandler` in the `ConsumeOptions`. Developer does not have to execute retry explicitly on acknowledgment failure and benefits from Reactor RabbitMQ retry support when acknowledging a message:

```
Receiver receiver = RabbitFlux.createReceiver();
BiConsumer<Receiver.AcknowledgmentContext, Exception> exceptionHandler =
    new ExceptionHandlers.RetryAcknowledgmentExceptionHandler( ①
        Duration.ofSeconds(20), Duration.ofMillis(500),
        ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE
    );
receiver.consumeManualAck("queue",
    new ConsumeOptions().exceptionHandler(exceptionHandler))
    .subscribe(msg -> {
        // ... ②
        msg.ack(); ③
    });
```

① Configure retry logic when exception occurs

② Process message

③ Send acknowledgment after business processing

Note the exception handler is a `BiConsumer<Receiver.AcknowledgmentContext, Exception>`. This means acknowledgment failure can be handled in any way, here we choose to retry the acknowledgment. Note also that by using `ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE`, we choose to retry only on unexpected connection failures and rely on the AMQP Java client to automatically re-create a new connection in the background. The decision to retry on a given exception can be customized by providing a `Predicate<Throwable>` in place of `ExceptionHandlers.CONNECTION_RECOVERY_PREDICATE`.

6.4. Advanced features

This section covers advanced uses of the Reactor RabbitMQ API.

6.4.1. Customizing connection creation

It is possible to specify only a `ConnectionFactory` for `Sender/ReceiverOptions` and let Reactor RabbitMQ create connection from this `ConnectionFactory`. Internally, Reactor RabbitMQ will create a `Mono<Connection>` to perform its operations and the connection will be created only when needed.

When the developer lets Reactor RabbitMQ create a `Mono<Connection>`, the library will take responsibility for the following actions for each instance of `Sender` and `Receiver`:

- using a cache to avoid creating several connections (by using `Mono#cache()`)
- making the `Mono<Connection>` register on `connectionSubscriptionScheduler` (with `Mono#subscribeOn()`)
- closing the connection when `Sender/Receiver#close()` is closed

Reactor RabbitMQ provides 2 ways to have more control over the connection creation, e.g. to provide a name or to connect to different nodes:

- using a connection supplier (simplest option, no Reactive API involved)
- using a custom `Mono<Connection>` (implies Reactive API but provides more control)

Creating a connection with a supplier

The following snippet shows how to create connections with a custom name:

```

ConnectionFactory connectionFactory = new ConnectionFactory();    ❶
connectionFactory.useNio();

Sender sender = RabbitFlux.createSender(new SenderOptions()
    .connectionFactory(connectionFactory)
    .connectionSupplier(cf -> cf.newConnection("sender"))    ❷
);

Receiver receiver = RabbitFlux.createReceiver(new ReceiverOptions()
    .connectionFactory(connectionFactory)
    .connectionSupplier(cf -> cf.newConnection("receiver"))    ❸
);

```

- ❶ Create and configure connection factory
- ❷ Create supplier that creates connection with a name
- ❸ Create supplier that creates connection with a name

When using a connection supplier, Reactor RabbitMQ will create a `Mono<Connection>` and will take care of the operations mentioned above (caching, registering on a scheduler, and closing).

Creating a connection with a custom `Mono`

The following snippet shows how to provide custom `Mono<Connection>`:

```

ConnectionFactory connectionFactory = new ConnectionFactory();
❶
connectionFactory.useNio();

Sender sender = RabbitFlux.createSender(new SenderOptions()
    .connectionMono(
        Mono.fromCallable(() -> connectionFactory.newConnection("sender")).cache()
❷
    );
Receiver receiver = RabbitFlux.createReceiver(new ReceiverOptions()
    .connectionMono(
        Mono.fromCallable(() -> connectionFactory.newConnection("receiver")).cache()
❸
    );

```

- ❶ Create and configure connection factory
- ❷ Create `Mono` that creates connection with a name
- ❸ Create `Mono` that creates connection with a name

Providing your own `Mono<Connection>` lets you take advantage of all the Reactor API (e.g. for caching) but has some caveats: Reactor RabbitMQ will not cache the provided `Mono<Connection>`, will not use it on a scheduler, and will not close it automatically. This is developer's responsibility to take care of these actions if they make sense in their context.

6.4.2. Sharing the same connection between **Sender** and **Receiver**

Sender and **Receiver** instances create their own **Connection** but it's possible to use only one or a few **Connection** instances to be able to use exclusive resources between a **Sender** and a **Receiver** or simply to control the number of created connections.

Both **SenderOptions** and **ReceiverOptions** have a **connectionSupplier** method that can encapsulate any logic to create the **Connection** the **Sender** or **Receiver** will end up using through a **Mono<Connection>**. Reactor RabbitMQ provides a way to share the exact same connection instance between some **Sender** and **Receiver** instances:

```
ConnectionFactory connectionFactory = new ConnectionFactory();
① connectionFactory.useNio();
Utils.ExceptionFunction<ConnectionFactory, ? extends Connection> connectionSupplier =
    Utils.singleConnectionSupplier(
②         connectionFactory, cf -> cf.newConnection()
    );

Sender sender = RabbitFlux.createSender(
    new SenderOptions().connectionSupplier(connectionSupplier)
③ );
Receiver receiver = RabbitFlux.createReceiver(
    new ReceiverOptions().connectionSupplier(connectionSupplier)
④ );
```

- ① Create and configure connection factory
- ② Create supplier that re-uses the same connection instance
- ③ Create sender with connection supplier
- ④ Create receiver with connection supplier

Be aware that closing the first **Sender** or **Receiver** will close the underlying AMQP connection for all the others.

6.4.3. Creating channels with a custom **Mono** in **Sender**

SenderOptions provides a **channelMono** property that is called when creating the **Channel** used in sending methods. This is a convenient way to provide any custom logic when creating the **Channel**, e.g. retry logic.

6.4.4. Threading considerations for resource management

A **Sender** instance maintains a **Mono<Channel>** to manage resources and by default the underlying **Channel** is cached. A new **Channel** is also automatically created in case of error. Channel creation is not a cheap operation, so this default behavior fits most use cases. Each resource management

method provides a counterpart method with an additional `ResourceManagementOptions` argument. This allows to provide a custom `Mono<Channel>` for a given resource operation. This can be useful when multiple threads are using the same `Sender` instance, to avoid using the same `Channel` from multiple threads.

```
Mono<Channel> channelMono = connectionMono.map(c -> {  
    try {  
        return c.createChannel();  
    } catch (Exception e) {  
        throw new RabbitFluxException(e);  
    }  
}).cache(); ①  
  
ResourceManagementOptions options = new ResourceManagementOptions()  
    .channelMono(channelMono); ②  
  
sender.declare(exchange("my.exchange"), options) ③  
    .then(sender.declare(queue("my.queue"), options)) ③  
    .then(sender.bind(binding("my.exchange", "a.b", "my.queue"), options)) ③  
    .subscribe(r -> System.out.println("Exchange and queue declared and bound"));
```

- ① Create `Channel` and cache it
- ② Use the `Mono<Channel>` in `ResourceManagementOptions`
- ③ Use `Mono<Channel>` for each operation

In the example above, each operation will use the same `Channel` as it is cached. This way these operations won't interfere with any other thread using the default resource management `Mono<Channel>` in the `Sender` instance.

6.4.5. Channel pooling in `Sender`

By default, `Sender#send*` methods open a new `Channel` for every call. This is OK for long-running calls, e.g. when the flux of outbound messages is infinite. For workloads whereby `Sender#send*` is called often for finite, short flux of messages, opening a new `Channel` every time may not be optimal.

It is possible to use a pool of channels as part of the `SendOptions` when sending outbound messages with `Sender`, as illustrated in the following snippet:

```
ChannelPool channelPool = ChannelPoolFactory.createChannelPool( ①  
    connectionMono,  
    new ChannelPoolOptions().maxCacheSize(5) ②  
);  
sender.send(outboundFlux, new SendOptions().channelPool(channelPool)); ③  
// ...  
channelPool.close(); ④
```

- ① Create `ChannelPool` with factory
- ②

Set the maximum size to 5 channels

- ③ Use a channel from the pool to send messages
- ④ Close the pool when no longer needed

Note it is developer's responsibility to close the pool when it is no longer necessary, typically at application shutdown.

[Micro-benchmarks](#) revealed channel pooling performs much better for sending short sequence of messages (1 to 10) repeatedly, without publisher confirms. With longer sequence of messages (100 or more), channel pooling can perform worse than without pooling at all. According to the same micro-benchmarks, channel pooling does not make sending with publisher confirms perform better, it appears to perform even worse. Don't take these conclusions for granted, you should always make your own benchmarks depending on your workloads.

6.4.6. Retry configuration on connection opening

Some applications may want to fail fast and throw an exception when the broker is unavailable. Other applications may want to retry connection opening when it fails.

Both `SenderOptions` and `ReceiverOptions` provide a `Function<Mono<? extends Connection>, Mono<? extends Connection>> connectionMonoConfigurator`, which is a hook in the `Mono<Connection>` creation of the `Sender` or `Receiver` instance. This is a good place to customize the `Mono<Connection>` to configure a retry policy.

The following snippet shows how to configure retry with an inline `connectionMonoConfigurator` for a `Sender`:

```
Receiver receiver = RabbitFlux.createReceiver(new ReceiverOptions()  
    .connectionMonoConfigurator(cm -> cm.retryBackoff(3, Duration.ofSeconds(5)))); ①
```

- ① Set up retry on connection opening

Please read the Reactor Core documentation for more information about [retry](#), [retry with exponential backoff](#), and [retry support in Reactor-Extra](#).



As `connectionMonoConfigurator` is simply a hook, operations the `Sender/Receiver` performs on the final `Mono<Connection>` like caching still happen. But note the `connectionMonoConfigurator` is not applied when a `Mono<Connection>` is provided to the `SenderOptions` or `ReceiverOptions`.