# RabbitMQ Stream Java Client

Version 0.12.0-SNAPSHOT: (d98c110)

# Table of Contents


What is a RabbitMQ Stream? . . . 1
When to Use RabbitMQ Stream? . . . 1
Other Way to Use Streams in RabbitMQ . . . 1
Guarantees . . . 2
Stream Client Overview . . . 2
Versioning . . . 2
Stability of Programming Interfaces . . . 3
The Stream Java Client . . . 3
- Setting up RabbitMQ . . . 3
  - With Docker . . . 3
    - With Docker Bridge Network Driver . . . 3
    - With Docker Host Network Driver . . . 4
  - With a RabbitMQ Package Running on the Host . . . 5
- Dependencies . . . 5
  - Maven . . . 5
  - Gradle . . . 5
- Snapshots . . . 5
- Sample Application . . . 6
- RabbitMQ Stream Java API . . . 9
  - Overview . . . 9
  - Environment . . . 9
    - Creating the Environment . . . 9
    - Understanding Connection Logic . . . 10
    - Enabling TLS . . . 10
    - Configuring the Environment . . . 11
    - When a Load Balancer is in Use . . . 13
    - Managing Streams . . . 14
  - Producer . . . 16
    - Creating a Producer . . . 16
    - Sending Messages . . . 17
    - Working with Complex Messages . . . 18
    - Message Deduplication . . . 19
    - Sub-Entry Batching and Compression . . . 22
  - Consumer . . . 24
    - Creating a Consumer . . . 24
    - Specifying an Offset . . . 26
    - Tracking the Offset for a Consumer . . . 27
    - Flow Control . . . 32

Single Active Consumer . . . 33
Super Streams (Partitioned Streams) . . . 36
Topology . . . 37
Super Stream Creation . . . 37
Publishing to a Super Stream . . . 37
Consuming From a Super Stream . . . 40
Advanced Topics . . . 47
Filtering . . . 47
Filtering on the Publishing Side . . . 48
Filtering on the Consuming Side . . . 48
Considerations on Filtering . . . 49
Using Native `epoll` . . . 50
Building the Client . . . 51
The Performance Tool . . . 51
Using the Performance Tool . . . 51
With Docker . . . 51
With Docker Host Network Driver . . . 52
With Docker Bridge Network Driver . . . 52
With the Java Binary . . . 53
Common Usage . . . 54
Connection . . . 54
Publishing Rate . . . 55
Number of Producers and Consumers . . . 55
Streams . . . 55
Publishing Batch Size . . . 56
Unconfirmed Messages . . . 57
Message Size . . . 57
Connection Pooling . . . 57
Advanced Usage . . . 58
Retention . . . 58
Offset (Consumer) . . . 58
Offset Tracking (Consumer) . . . 58
Consumer Names . . . 59
Producer Names . . . 59
Load Balancer in Front of the Cluster . . . 60
Single Active Consumer . . . 60
Super Streams . . . 60
Monitoring . . . 61
Synchronizing Several Instances . . . 62
Using Environment Variables as Options . . . 63
Logging . . . 63

Building the Performance Tool . . . . . 64

The RabbitMQ Stream Java Client is a Java library to communicate with the RabbitMQ Stream Plugin. It allows creating and deleting streams, as well as publishing to and consuming from these streams. Learn more in the client overview.

# What is a RabbitMQ Stream?

A RabbitMQ stream is a persistent and replicated data structure that models an append-only log. It differs from the classical RabbitMQ queue in the way message consumption works. In a classical RabbitMQ queue, consuming removes messages from the queue. In a RabbitMQ stream, consuming leaves the stream intact. So the content of a stream can be read and re-read without impact or destructive effect.

None of the stream or classical queue data structure is better than the other, they are usually suited for different use cases.

# When to Use RabbitMQ Stream?

RabbitMQ Stream was developed to cover the following messaging use cases:

- *Large fan-outs:* when several consumer applications need to read the same messages.
- *Replay / Time-traveling:* when consumer applications need to read the whole history of data or from a given point in a stream.
- *Throughput performance:* when higher throughput than with other protocols (AMQP, STOMP, MQTT) is required.
- *Large logs:* when large amount of data need to be stored, with minimal in-memory overhead.

# Other Way to Use Streams in RabbitMQ

It is also possible to use the stream abstraction in RabbitMQ with the AMQP 0-9-1 protocol. Instead of consuming from a stream with the stream protocol, one consumes from a "stream-powered" queue with the AMQP 0-9-1 protocol. A "stream-powered" queue is a special type of queue that is backed up with a stream infrastructure layer and adapted to provide the stream semantics (mainly non-destructive reading).

Using such a queue has the advantage to provide the features inherent to the stream abstraction (append-only structure, non-destructive reading) with any AMQP 0-9-1 client library. This is clearly interesting when considering the maturity of AMQP 0-9-1 client libraries and the ecosystem around AMQP 0-9-1.

But by using it, one does not benefit from the performance of the stream protocol, which has been designed for performance in mind, whereas AMQP 0-9-1 is a more general-purpose protocol.

It is not possible to use "stream-powered" queues with the stream Java client, you need to use an AMQP 0-9-1 client library.

# Guarantees

RabbitMQ stream provides at-least-once guarantees thanks to the publisher confirm mechanism, which is supported by the stream Java client.

Message deduplication is also supported on the publisher side.

# Stream Client Overview

The RabbitMQ Stream Java Client implements the RabbitMQ Stream protocol and avoids dealing with low-level concerns by providing high-level functionalities to build fast, efficient, and robust client applications.

- *administrate streams (creation/deletion) directly from applications.* This can also be useful for development and testing.
- *adapt publishing throughput* thanks to the configurable batch size and flow control.
- *avoid publishing duplicate messages* thanks to message deduplication.
- *consume asynchronously from streams and resume where left off* thanks to automatic or manual offset tracking.
- *enforce best practices to create client connections* – to stream leaders for publishers to minimize inter-node traffic and to stream replicas for consumers to offload leaders.
- *optimize resources* thanks to automatic growing and shrinking of connections depending on the number of publishers and consumers.
- *let the client handle network failure* thanks to automatic connection recovery and automatic re-subscription for consumers.

# Versioning

The RabbitMQ Stream Java Client is in development and stabilization phase. When the stabilization phase ends, a 1.0.0 version will be cut, and semantic versioning is likely to be enforced.

Before reaching the stable phase, the client will use a versioning scheme of `[0.MINOR.PATCH]` where:

- `0` indicates the project is still in a stabilization phase.
- `MINOR` is a 0-based number incrementing with each new release cycle. It generally reflects significant changes like new features and potentially some programming interfaces changes.
- `PATCH` is a 0-based number incrementing with each service release, that is bux fixes.

Breaking changes between releases can happen but will be kept to a minimum. The next section provides more details about the evolution of programming interfaces.

# Stability of Programming Interfaces

The RabbitMQ Stream Java Client is in active development but its programming interfaces will remain as stable as possible. There is no guarantee though that they will remain completely stable, at least until it reaches version 1.0.0.

The client contains 2 sets of programming interfaces whose stability are of interest for application developers:

- Application Programming Interfaces (API): those are the ones used to write application logic. They include the interfaces and classes in the `com.rabbitmq.stream` package (e.g. `Producer`, `Consumer`, `Message`). These API constitute the main programming model of the client and will be kept as stable as possible.
- Service Provider Interfaces (SPI): those are interfaces to implement mainly technical behavior in the client. They are not meant to be used to implement application logic. Application developers may have to refer to them in the configuration phase and if they want to custom some internal behavior in the client. SPI include interfaces and classes in the `com.rabbitmq.stream.codec`, `com.rabbitmq.stream.compression`, `com.rabbitmq.stream.metrics` packages, among others. *These SPI are susceptible to change, but this should not impact the majority of applications*, as the changes would typically stay intern to the client.

# The Stream Java Client

The library requires Java 8 or later. Java 11 is recommended (CRC calculation uses methods available as of Java 9.)

## Setting up RabbitMQ

A RabbitMQ 3.9+ node with the stream plugin enabled is required. The easiest way to get up and running is to use Docker.

### With Docker

There are different ways to make the broker visible to the client application when running in Docker. The next sections show a couple of options suitable for local development.

| | |
|---|---|
| **NOTE** | *Docker on macOS*<br>Docker runs on a virtual machine when using macOS, so do not expect high performance when using RabbitMQ Stream inside Docker on a Mac. |

#### With Docker Bridge Network Driver

This section shows how to start a broker instance for local development (the broker Docker container and the client application are assumed to run on the same host).

The following command creates a one-time Docker container to run RabbitMQ:

*Running the stream plugin with Docker*

```
docker run -it --rm --name rabbitmq -p 5552:5552 \
    -e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS='-rabbitmq_stream advertised_host
localhost' \
    rabbitmq:3.12
```

The previous command exposes only the stream port (5552), you can expose ports for other protocols:

*Exposing the AMQP 0.9.1 and management ports:*

```
docker run -it --rm --name rabbitmq -p 5552:5552 -p 5672:5672 -p 15672:15672 \
    -e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS='-rabbitmq_stream advertised_host
localhost' \
    rabbitmq:3.12-management
```

Refer to the official RabbitMQ Docker image web page to find out more about its usage.

Once the container is started, **the stream plugin must be enabled**:

*Enabling the stream plugin:*

```
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
```

#### With Docker Host Network Driver

This is the simplest way to run the broker locally. The container uses the host network, this is perfect for experimenting locally.

*Running RabbitMQ Stream with the host network driver*

```
docker run -it --rm --name rabbitmq --network host rabbitmq:3.12
```

Once the container is started, **the stream plugin must be enabled**:

*Enabling the stream plugin:*

```
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
```

The container will use the following ports: 5552 (for stream) and 5672 (for AMQP.)

**NOTE**

*Docker Host Network Driver Support*

The host networking driver **only works on Linux hosts**.

### With a RabbitMQ Package Running on the Host

Using a package implies installing Erlang.

- Make sure to use RabbitMQ 3.11 or later.
- Follow the steps to install Erlang and the appropriate package
- Enable the plugin `rabbitmq-plugins enable rabbitmq_stream`.
- The stream plugin listens on port 5552.

Refer to the stream plugin documentation for more information on configuration.

## Dependencies

Use your favorite build management tool to add the client dependencies to your project.

### Maven

*pom.xml*

```
<dependencies>

  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>stream-client</artifactId>
    <version>0.12.0-SNAPSHOT</version>
  </dependency>

</dependencies>
```

Snapshots require to declare the appropriate repository.

### Gradle

*build.gradle*

```
dependencies {
  compile "com.rabbitmq:stream-client:0.12.0-SNAPSHOT"
}
```

Snapshots require to declare the appropriate repository.

## Snapshots

Releases are available from Maven Central, which does not require specific declaration. Snapshots are available from a repositoriy which must be declared in the dependency management configuration.

With Maven:

*Snapshot repository declaration for Maven*

```
<repositories>

  <repository>
    <id>ossrh</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots</url>
    <snapshots><enabled>true</enabled></snapshots>
    <releases><enabled>false</enabled></releases>
  </repository>

</repositories>
```

With Gradle:

*Snapshot repository declaration for Gradle:*

```
repositories {
  maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
  mavenCentral()
}
```

# Sample Application

This section covers the basics of the RabbitMQ Stream Java API by building a small publish/consume application. This is a good way to get an overview of the API. If you want a more comprehensive introduction, you can go to the reference documentation section.

The sample application publishes some messages and then registers a consumer to make some computations out of them. The source code is available on GitHub.

The sample class starts with a few imports:

*Imports for the sample application*

```
import com.rabbitmq.stream.*;
import java.util.UUID;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.util.stream.IntStream;
```

The next step is to create the `Environment`. It is a management object used to manage streams and create producers as well as consumers. The next snippet shows how to create an `Environment` instance and create the stream used in the application:

*Creating the environment*

```
System.out.println("Connecting...");
Environment environment = Environment.builder().build();  ①
String stream = UUID.randomUUID().toString();
environment.streamCreator().stream(stream).create();  ②
```

① Use `Environment#builder` to create the environment

② Create the stream

Then comes the publishing part. The next snippet shows how to create a `Producer`, send messages, and handle publishing confirmations, to make sure the broker has taken outbound messages into account. The application uses a count down latch to move on once the messages have been confirmed.

*Publishing messages*

```
System.out.println("Starting publishing...");
int messageCount = 10000;
CountDownLatch publishConfirmLatch = new CountDownLatch(messageCount);
Producer producer = environment.producerBuilder()  ①
        .stream(stream)
        .build();
IntStream.range(0, messageCount)
        .forEach(i -> producer.send(  ②
                producer.messageBuilder()                    ③
                    .addData(String.valueOf(i).getBytes())   ③
                    .build(),                                ③
                confirmationStatus -> publishConfirmLatch.countDown()  ④
        ));
publishConfirmLatch.await(10, TimeUnit.SECONDS);  ⑤
producer.close();  ⑥
System.out.printf("Published %,d messages%n", messageCount);
```

① Create the `Producer` with `Environment#producerBuilder`

② Send messages with `Producer#send(Message, ConfirmationHandler)`

③ Create a message with `Producer#messageBuilder`

④ Count down on message publishing confirmation

⑤ Wait for all publishing confirmations to have arrived

⑥ Close the producer

It is now time to consume the messages. The `Environment` lets us create a `Consumer` and provide some logic on each incoming message by implementing a `MessageHandler`. The next snippet does this to calculate a sum and output it once all the messages have been received:

*Consuming messages*

```
System.out.println("Starting consuming...");
```

```
AtomicLong sum = new AtomicLong(0);
CountDownLatch consumeLatch = new CountDownLatch(messageCount);
Consumer consumer = environment.consumerBuilder()  ①
        .stream(stream)
        .offset(OffsetSpecification.first()) ②
        .messageHandler((offset, message) -> {  ③
            sum.addAndGet(Long.parseLong(new String(message.getBodyAsBinary())));  ④
            consumeLatch.countDown();  ⑤
        })
        .build();

consumeLatch.await(10, TimeUnit.SECONDS);  ⑥

System.out.println("Sum: " + sum.get());  ⑦

consumer.close();  ⑧
```

① Create the `Consumer` with `Environment#consumerBuilder`

② Start consuming from the beginning of the stream

③ Set up the logic to handle messages

④ Add the value in the message body to the sum

⑤ Count down on each message

⑥ Wait for all messages to have arrived

⑦ Output the sum

⑧ Close the consumer

The application has some cleaning to do before terminating, that is deleting the stream and closing the environment:

*Cleaning before terminating*

```
environment.deleteStream(stream);  ①
environment.close();  ②
```

① Delete the stream

② Close the environment

You can run the sample application from the root of the project (you need a running local RabbitMQ node with the stream plugin enabled):

```
$ ./mvnw -q test-compile exec:java -Dexec.classpathScope="test" \
    -Dexec.mainClass="com.rabbitmq.stream.docs.SampleApplication"
Starting publishing...
Published 10000 messages
Starting consuming...
Sum: 49995000
```

You can remove the `-q` flag if you want more insight on the execution of the build.

# RabbitMQ Stream Java API

## Overview

This section describes the API to connect to the RabbitMQ Stream Plugin, publish messages, and consume messages. There are 3 main interfaces:

- `com.rabbitmq.stream.Environment` for connecting to a node and optionally managing streams.
- `com.rabbitmq.stream.Producer` to publish messages.
- `com.rabbitmq.stream.Consumer` to consume messages.

## Environment

### Creating the Environment

The environment is the main entry point to a node or a cluster of nodes. `Producer` and `Consumer` instances are created from an `Environment` instance. Here is the simplest way to create an `Environment` instance:

*Creating an environment with all the defaults*

```
Environment environment = Environment.builder().build();  ①
// ...
environment.close(); ②
```

① Create an environment that will connect to localhost:5552

② Close the environment after usage

Note the environment must be closed to release resources when it is no longer needed.

Consider the environment like a long-lived object. An application will usually create one `Environment` instance when it starts up and close it when it exits.

It is possible to use a URI to specify all the necessary information to connect to a node:

*Creating an environment with a URI*

```
Environment environment = Environment.builder()
        .uri("rabbitmq-stream://guest:guest@localhost:5552/%2f")  ①
        .build();
```

① Use the `uri` method to specify the URI to connect to

The previous snippet uses a URI that specifies the following information: host, port, username, password, and virtual host (`/`, which is encoded as `%2f`). The URI follows the same rules as the AMQP 0.9.1 URI, except the protocol must be `rabbitmq-stream`. TLS is enabled by using the `rabbitmq-`

`stream+tls` scheme in the URI.

When using one URI, the corresponding node will be the main entry point to connect to. The `Environment` will then use the stream protocol to find out more about streams topology (leaders and replicas) when asked to create `Producer` and `Consumer` instances. The `Environment` may become blind if this node goes down though, so it may be more appropriate to specify several other URIs to try in case of failure of a node:

*Creating an environment with several URIs*

```
Environment environment = Environment.builder()
        .uris(Arrays.asList(                    ①
                "rabbitmq-stream://host1:5552",
                "rabbitmq-stream://host2:5552",
                "rabbitmq-stream://host3:5552")
        )
        .build();
```

① Use the `uris` method to specify several URIs

By specifying several URIs, the environment will try to connect to the first one, and will pick a new URI randomly in case of disconnection.

#### Understanding Connection Logic

Creating the environment to connect to a cluster node works usually seamlessly. Creating publishers and consumers can cause problems as the client uses hints from the cluster to find the nodes where stream leaders and replicas are located to connect to the appropriate nodes.

These connection hints can be accurate or less appropriate depending on the infrastructure. If you hit some connection problems at some point – like hostnames impossible to resolve for client applications - this blog post should help you understand what is going on and fix the issues.

#### Enabling TLS

TLS can be enabled by using the `rabbitmq-stream+tls` scheme in the URI. The default TLS port is 5551.

Use the `EnvironmentBuilder#tls` method to configure TLS. The most important setting is a `io.netty.handler.ssl.SslContext` instance, which is created and configured with the `io.netty.handler.ssl.SslContext#forClient` method. Note hostname verification is enabled by default.

The following snippet shows a common configuration, whereby the client is instructed to trust servers with certificates signed by the configured certificate authority (CA).

*Creating an environment that uses TLS*

```
X509Certificate certificate;
try (FileInputStream inputStream =
            new FileInputStream("/path/to/ca_certificate.pem")) {
```

```
    CertificateFactory fact = CertificateFactory.getInstance("X.509");
    certificate = (X509Certificate) fact.generateCertificate(inputStream); ①
}
SslContext sslContext = SslContextBuilder
    .forClient()
    .trustManager(certificate)  ②
    .build();

Environment environment = Environment.builder()
    .uri("rabbitmq-stream+tls://guest:guest@localhost:5551/%2f")  ③
    .tls().sslContext(sslContext)  ④
    .environmentBuilder()
    .build();
```

① Load certificate authority (CA) certificate from PEM file

② Configure Netty `SslContext` to trust CA certificate

③ Use TLS scheme in environment URI

④ Set `SslContext` in environment configuration

It is sometimes handy to trust any server certificates in development environments. `EnvironmentBuilder#tls` provides the `trustEverything` method to do so. **This should not be used in a production environment**.

*Creating a TLS environment that trusts all server certificates for development*

```
Environment environment = Environment.builder()
    .uri("rabbitmq-stream+tls://guest:guest@localhost:5551/%2f")
    .tls().trustEverything()  ①
    .environmentBuilder()
    .build();
```

① Trust all server certificates

### Configuring the Environment

The following table sums up the main settings to create an `Environment`:

| Parameter Name | Description | Default |
|---|---|---|
| `uri` | The URI of the node to connect to (single node). | `rabbitmq-stream://guest:guest@localhost:5552/%2f` |
| `uris` | The URI of the nodes to try to connect to (cluster). | `rabbitmq-stream://guest:guest@localhost:5552/%2f` singleton list |
| `host` | Host to connect to. | `localhost` |
| `port` | Port to use. | `5552` |
| `username` | Username to use to connect. | `guest` |

| Parameter Name | Description | Default |
| --- | --- | --- |
| `password` | Password to use to connect. | `guest` |
| `virtualHost` | Virtual host to connect to. | `/` |
| `rpcTimeout` | Timeout for RPC calls. | `Duration.ofSeconds(10)` |
| `recoveryBackOffDelayPolicy` | Delay policy to use for backoff on connection recovery. | Fixed delay of 5 seconds |
| `topologyUpdateBackOffDelayPolicy` | Delay policy to use for backoff on topology update, e.g. when a stream replica moves and a consumer needs to connect to another node. | Initial delay of 5 seconds then delay of 1 second. |
| `scheduledExecutorService` | Executor used to schedule infrastructure tasks like background publishing, producers and consumers migration after disconnection or topology update. If a custom executor is provided, it is the developer's responsibility to close it once it is no longer necessary. | `Executors .newScheduledThreadPool( Runtime .getRuntime() .availableProcessors() );` |
| `maxProducersByConnection` | The maximum number of `Producer` instances a single connection can maintain before a new connection is open. The value must be between 1 and 255. | 255 |
| `maxTrackingConsumersByConnection` | The maximum number of `Consumer` instances that store their offset a single connection can maintain before a new connection is open. The value must be between 1 and 255. | 50 |
| `maxConsumersByConnection` | The maximum number of `Consumer` instances a single connection can maintain before a new connection is open. The value must be between 1 and 255. | 255 |
| `lazyInitialization` | To delay the connection opening until necessary. | false |

| Parameter Name | Description | Default |
| --- | --- | --- |
| `id` | Informational ID for the environment instance. Used as a prefix for connection names. | `rabbitmq-stream` |
| `addressResolver` | Contract to change resolved node address to connect to. | Pass-through (no-op) |
| `tls` | Configuration helper for TLS. | TLS is enabled if a `rabbitmq-stream+tls` URI is provided. |
| `tls#hostnameVerification` | Enable or disable hostname verification. | Enabled by default. |
| `tls#sslContext` | Set the `io.netty.handler.ssl.SslContext` used for the TLS connection. Use `io.netty.handler.ssl.SslContextBuilder#forClient` to configure it. The server certificate chain and the client private key are the typical elements that need to be configured. | The JDK trust manager and no client private key. |
| `tls#trustEverything` | Helper to configure a `SslContext` that trusts all server certificates and does not use a client private key. **Only for development**. | Disabled by default. |
| `netty` | Configuration helper for Netty. | |
| `netty#eventLoopGroup` | Netty's event dispatcher. It is the developer's responsibility to close the `EventLoopGroup` they provide. | `NioEventLoopGroup` instance closed automatically with the `Environment` instance. |
| `netty#ByteBufAllocator` | `ByteBuf` allocator. | ByteBufAllocator.DEFAULT |
| `netty#channelCustomizer` | Extension point to customize Netty's `Channel` instances used for connections. | None |
| `netty#bootstrapCustomizer` | Extension point to customize Netty's `Bootstrap` instances used to configure connections. | None |

### When a Load Balancer is in Use

A load balancer can misguide the client when it tries to connect to nodes that host stream leaders and replicas. The "Connecting to Streams" blog post covers why client applications must connect to the appropriate nodes in a cluster and how a load balancer can make things complicated for them.

The `EnvironmentBuilder#addressResolver(AddressResolver)` method allows intercepting the node resolution after metadata hints and before connection. Applications can use this hook to ignore metadata hints and always use the load balancer, as illustrated in the following snippet:

*Using a custom address resolver to always use a load balancer*

```
Address entryPoint = new Address("my-load-balancer", 5552);  ①
Environment environment = Environment.builder()
    .host(entryPoint.host())  ②
    .port(entryPoint.port())  ②
    .addressResolver(address -> entryPoint)  ③
    .build();
```

① Set the load balancer address

② Use load balancer address for initial connection

③ Ignore metadata hints, always use load balancer

The blog post covers the underlying details of this workaround.

**Managing Streams**

Streams are usually long-lived, centrally-managed entities, that is, applications are not supposed to create and delete them. It is nevertheless possible to create and delete stream with the `Environment`. This comes in handy for development and testing purposes.

Streams are created with the `Environment#streamCreator()` method:

*Creating a stream*

```
environment.streamCreator().stream("my-stream").create();  ①
```

① Create the `my-stream` stream

`StreamCreator#create` is idempotent: trying to re-create a stream with the same name and same properties (e.g. maximum size, see below) will not throw an exception. In other words, you can be sure the stream has been created once `StreamCreator#create` returns. Note it is not possible to create a stream with the same name as an existing stream but with different properties. Such a request will result in an exception.

Streams can be deleted with the `Environment#delete(String)` method:

*Deleting a stream*

```
environment.deleteStream("my-stream");  ①
```

① Delete the `my-stream` stream

Note you should avoid stream churn (creating and deleting streams repetitively) as their creation and deletion imply some significant housekeeping on the server side (interactions with the file system, communication between nodes of the cluster).

It is also possible to limit the size of a stream when creating it. A stream is an append-only data structure and reading from it does not remove data. This means a stream can grow indefinitely. RabbitMQ Stream supports a size-based and time-based retention policies: once the stream reaches a given size or a given age, it is truncated (starting from the beginning).

**IMPORTANT**

*Limit the size of streams if appropriate!*

Make sure to set up a retention policy on potentially large streams if you don't want to saturate the storage devices of your servers. Keep in mind that this means some data will be erased!

It is possible to set up the retention policy when creating the stream:

*Setting the retention policy when creating a stream*

```
environment.streamCreator()
        .stream("my-stream")
        .maxLengthBytes(ByteCapacity.GB(10))  ①
        .maxSegmentSizeBytes(ByteCapacity.MB(500))  ②
        .create();
```

① Set the maximum size to 10 GB

② Set the segment size to 500 MB

The previous snippet mentions a segment size. RabbitMQ Stream does not store a stream in a big, single file, it uses segment files for technical reasons. A stream is truncated by deleting whole segment files (and not part of them)so the maximum size of a stream is usually significantly higher than the size of segment files. 500 MB is a reasonable segment file size to begin with.

**NOTE**

*When does the broker enforce the retention policy?*

The broker enforces the retention policy when the segments of a stream roll over, that is when the current segment has reached its maximum size and is closed in favor of a new one. This means the maximum segment size is a critical setting in the retention mechanism.

RabbitMQ Stream also supports a time-based retention policy: segments get truncated when they reach a certain age. The following snippet illustrates how to set the time-based retention policy:

*Setting a time-based retention policy when creating a stream*

```
environment.streamCreator()
        .stream("my-stream")
        .maxAge(Duration.ofHours(6))  ①
        .maxSegmentSizeBytes(ByteCapacity.MB(500))  ②
        .create();
```

① Set the maximum age to 6 hours

② Set the segment size to 500 MB

## Producer

### Creating a Producer

A `Producer` instance is created from the `Environment`. The only mandatory setting to specify is the stream to publish to:

*Creating a producer from the environment*

```
Producer producer = environment.producerBuilder()  ①
        .stream("my-stream")  ②
        .build();  ③
// ...
producer.close();  ④
```

① Use `Environment#producerBuilder()` to define the producer

② Specify the stream to publish to

③ Create the producer instance with `build()`

④ Close the producer after usage

Consider a `Producer` instance like a long-lived object, do not create one to send just one message.

**NOTE**

*Producer thread safety*

Producer instances are thread-safe. Deduplication imposes restrictions on the usage of threads though.

Internally, the `Environment` will query the broker to find out about the topology of the stream and will create or re-use a connection to publish to the leader node of the stream.

The following table sums up the main settings to create a `Producer`:

| Parameter Name | Description | Default |
|---|---|---|
| `stream` | The stream to publish to. | No default, mandatory setting. |
| `name` | The logical name of the producer. Specify a name to enable message deduplication. | `null` (no deduplication) |
| `batchSize` | The maximum number of messages to accumulate before sending them to the broker. | 100 |

| Parameter Name | Description | Default |
| --- | --- | --- |
| `subEntrySize` | The number of messages to put in a sub-entry. A sub-entry is one "slot" in a publishing frame, meaning outbound messages are not only batched in publishing frames, but in sub-entries as well. Use this feature to increase throughput at the cost of increased latency and potential duplicated messages even when deduplication is enabled. See the dedicated section for more information. | 1 (meaning no use of sub-entry batching) |
| `compression` | Compression algorithm to use when sub-entry batching is in use. See the dedicated section for more information. | Compression.NONE |
| `maxUnconfirmedMessages` | The maximum number of unconfirmed outbound messages. `Producer#send` will start blocking when the limit is reached. | 10,000 |
| `batchPublishingDelay` | Period to send a batch of messages. | 100 ms |
| `confirmTimeout` | Time before the client calls the confirm callback to signal outstanding unconfirmed messages timed out. | 30 seconds |
| `enqueueTimeout` | Time before enqueueing of a message fail when the maximum number of unconfirmed is reached. The callback of the message will be called with a negative status. Set the value to `Duration.ZERO` if there should be no timeout. | 10 seconds. |

### Sending Messages

Once a `Producer` has been created, it is possible to send a message with the `Producer#send(Message, ConfirmationHandler)` method. The following snippet shows how to publish a message with a byte array payload:

*Sending a message*

```
byte[] messagePayload = "hello".getBytes(StandardCharsets.UTF_8);  ①
producer.send(
        producer.messageBuilder().addData(messagePayload).build(),  ②
        confirmationStatus -> {  ③
            if (confirmationStatus.isConfirmed()) {
                // the message made it to the broker
            } else {
                // the message did not make it to the broker
            }
        });
```

① The payload of a message is an array of bytes

② Create the message with `Producer#messageBuilder()`

③ Define the behavior on publish confirmation

Messages are not only made of a `byte[]` payload, we will see in the next section they can also carry pre-defined and application properties.

**NOTE**

*Use a `MessageBuilder` instance only once*

A `MessageBuilder` instance is meant to create only one message. You need to create a new instance of `MessageBuilder` for every message you want to create.

The `ConfirmationHandler` defines an asynchronous callback invoked when the client received from the broker the confirmation the message has been taken into account. The `ConfirmationHandler` is the place for any logic on publishing confirmation, including re-publishing the message if it is negatively acknowledged.

**WARNING**

*Keep the confirmation callback as short as possible*

The confirmation callback should be kept as short as possible to avoid blocking the connection thread. Not doing so can make the `Environment`, `Producer`, `Consumer` instances sluggish or even block them. Any long processing should be done in a separate thread (e.g. with an asynchronous `ExecutorService`).

### Working with Complex Messages

The publishing example above showed that messages are made of a byte array payload, but it did not go much further. Messages in RabbitMQ Stream can actually be more sophisticated, as they comply to the AMQP 1.0 message format.

In a nutshell, a message in RabbitMQ Stream has the following structure:

- properties: *a defined set of standard properties of the message* (e.g. message ID, correlation ID, content type, etc).
- application properties: a set of arbitrary key/value pairs.
- body: typically an array of bytes.

- message annotations: a set of key/value pairs (aimed at the infrastructure).

The RabbitMQ Stream Java client uses the `Message` interface to abstract a message and the recommended way to create `Message` instances is to use the `Producer#messageBuilder()` method. To publish a `Message`, use the `Producer#send(Message,ConfirmationHandler)`:

*Creating a message with properties*

```
Message message = producer.messageBuilder()  ①
        .properties()  ②
            .messageId(UUID.randomUUID())
            .correlationId(UUID.randomUUID())
            .contentType("text/plain")
        .messageBuilder()  ③
            .addData("hello".getBytes(StandardCharsets.UTF_8))  ④
        .build();  ⑤
producer.send(message, confirmationStatus -> { }); ⑥
```

① Get the message builder from the producer

② Get the properties builder and set some properties

③ Go back to message builder

④ Set byte array payload

⑤ Build the message instance

⑥ Publish the message

**NOTE**

*Is RabbitMQ Stream based on AMQP 1.0?*

AMQP 1.0 is a standard that defines *an efficient binary peer-to-peer protocol for transporting messages between two processes over a network*. It also defines *an abstract message format, with concrete standard encoding*. This is only the latter part that RabbitMQ Stream uses. The AMQP 1.0 protocol is not used, only AMQP 1.0 encoded messages are wrapped into the RabbitMQ Stream binary protocol.

The actual AMQP 1.0 message encoding and decoding happen on the client side, the RabbitMQ Stream plugin stores only bytes, it has no idea that AMQP 1.0 message format is used.

AMQP 1.0 message format was chosen because of its flexibility and its advanced type system. It provides good interoperability, which allows streams to be accessed as AMQP 0-9-1 queues, without data loss.

### Message Deduplication

RabbitMQ Stream provides publisher confirms to avoid losing messages: once the broker has persisted a message it sends a confirmation for this message. But this can lead to duplicate messages: imagine the connection closes because of a network glitch after the message has been persisted but *before* the confirmation reaches the producer. Once reconnected, the producer will retry to send the same message, as it never received the confirmation. So the message will be

persisted twice.

Luckily RabbitMQ Stream can detect and filter out duplicated messages, based on 2 client-side elements: the *producer name* and the *message publishing ID*.

**WARNING**

*Deduplication is not guaranteed when using sub-entries batching*

It is not possible to guarantee deduplication when sub-entry batching is in use. Sub-entry batching is disabled by default and it does not prevent from batching messages in a single publish frame, which can already provide very high throughput.

**WARNING**

*Deduplication is not guaranteed when publishing on several threads*

We'll see below that deduplication works using a strictly increasing sequence for messages. This means messages must be published in order and the preferred way to do this is usually *within a single thread*. Even if messages are *created* in order, with the proper sequence ID, if they are published in several threads, they can get out of order, e.g. message 5 can be *published* before message 2. The deduplication mechanism will then filter out message 2 in this case.

So you have to be very careful about the way your applications publish messages when deduplication is in use. If you worry about performance, note it is possible to publish hundreds of thousands of messages in a single thread with RabbitMQ Stream.

#### Setting the Name of a Producer

The producer name is set when creating the producer instance, which automatically enables deduplication:

*Naming a producer to enable message deduplication*

```
Producer producer = environment.producerBuilder()
    .name("my-app-producer")  ①
    .confirmTimeout(Duration.ZERO)  ②
    .stream("my-stream")
    .build();
```

① Set a name for the producer

② Disable confirm timeout check

Thanks to the name, the broker will be able to track the messages it has persisted on a given stream for this producer. If the producer connection unexpectedly closes, it will automatically recover and retry outstanding messages. The broker will then filter out messages it has already received and persisted. No more duplicates!

**IMPORTANT**

*Why setting `confirmTimeout` to 0 when using deduplication?*

The point of deduplication is to avoid duplicates when retrying unconfirmed

messages. But why retrying in the first place? To avoid *losing* messages, that is enforcing *at-least-once* semantics. If the client does not stubbornly retry messages and gives up at some point, messages can be lost, which maps to *at-most-once* semantics. This is why the deduplication examples set the `confirmTimeout` setting to `Duration.ZERO`: to disable the background task that calls the confirmation callback for outstanding messages that time out. This way the client will do its best to retry messages until they are confirmed.

Consider the producer name a logical name. It should not be a random sequence that changes when the producer application is restarted. Names like `online-shop-order` or `online-shop-invoice` are better names than `3d235e79-047a-46a6-8c80-9d159d3e1b05`. There should be only one living instance of a producer with a given name on a given stream at the same time.

#### Understanding Publishing ID

The producer name is only one part of the deduplication mechanism, the other part is the *message publishing ID*. If the producer has a name, the client automatically assigns a publishing ID to each outbound message for the producer. The publishing ID is a strictly increasing sequence, starting at 0 and incremented for each message. The default publishing sequence is good enough for deduplication, but it is possible to assign a publishing ID to each message:

*Using an explicit publishing ID*

```
Message message = producer.messageBuilder()
    .publishingId(1)  ①
    .addData("hello".getBytes(StandardCharsets.UTF_8))
    .build();
producer.send(message, confirmationStatus -> { });
```

① Set a publishing ID on a message

There are a few rules to follow when using a custom publishing ID sequence:

- the sequence should start at 0
- the sequence must be strictly increasing
- there can be gaps in the sequence (e.g. 0, 1, 2, 3, 6, 7, 9, 10, etc)

A custom publishing ID sequence has usually a meaning: it can be the line number of a file or the primary key in a database.

Note the publishing ID is not part of the message: it is not stored with the message and so is not available when consuming the message. It is still possible to store the value in the AMQP 1.0 message application properties or in an appropriate properties (e.g. `messageId`).

**IMPORTANT**

*Do not mix client-assigned and custom publishing ID*

As soon as a producer name is set, message deduplication is enabled. It is then possible to let the producer assign a publishing ID to each message or assign custom publishing IDs. **Do one or the other, not both!**

#### Restarting a Producer Where It Left Off

Using a custom publishing sequence is even more useful to restart a producer where it left off. Imagine a scenario whereby the producer is sending a message for each line in a file and the application uses the line number as the publishing ID. If the application restarts because of some necessary maintenance or even a crash, the producer can restart from the beginning of the file: there would no duplicate messages because the producer has a name and the application sets publishing IDs appropriately. Nevertheless, this is far from ideal, it would be much better to restart just after the last line the broker successfully confirmed. Fortunately this is possible thanks to the `Producer#getLastPublishing()` method, which returns the last publishing ID for a given producer. As the publishing ID in this case is the line number, the application can easily scroll to the next line and restart publishing from there.

The next snippet illustrates the use of `Producer#getLastPublishing()`:

*Setting a producer where it left off*

```
Producer producer = environment.producerBuilder()
    .name("my-app-producer")  ①
    .confirmTimeout(Duration.ZERO)  ②
    .stream("my-stream")
    .build();
long nextPublishingId = producer.getLastPublishingId() + 1;  ③
while (moreContent(nextPublishingId)) {
    byte[] content = getContent(nextPublishingId); ④
    Message message = producer.messageBuilder()
        .publishingId(nextPublishingId) ⑤
        .addData(content)
        .build();
    producer.send(message, confirmationStatus -> {});
    nextPublishingId++;
}
```

① Set a name for the producer

② Disable confirm timeout check

③ Query last publishing ID for this producer and increment it

④ Scroll to the content for the next publishing ID

⑤ Set the message publishing

#### Sub-Entry Batching and Compression

RabbitMQ Stream provides a special mode to publish, store, and dispatch messages: sub-entry batching. This mode increases throughput at the cost of increased latency and potential duplicated messages even when deduplication is enabled. It also allows using compression to reduce bandwidth and storage if messages are reasonably similar, at the cost of increasing CPU usage on the client side.

Sub-entry batching consists in squeezing several messages – a batch – in the slot that is usually used for one message. This means outbound messages are not only batched in publishing frames, but in

sub-entries as well.

You can enable sub-entry batching by setting the `ProducerBuilder#subEntrySize` parameter to a value greater than 1, like in the following snippet:

*Enabling sub-entry batching*

```
Producer producer = environment.producerBuilder()
    .stream("my-stream")
    .batchSize(100) ①
    .subEntrySize(10) ②
    .build();
```

① Set batch size to 100 (the default)

② Set sub-entry size to 10

Reasonable values for the sub-entry size usually go from 10 to a few dozens.

A sub-entry batch will go directly to disc after it reached the broker, so the publishing client has complete control over it. This is the occasion to take advantage of the similarity of messages and compress them. There is no compression by default but you can choose among several algorithms with the `ProducerBuilder#compression(Compression)` method:

*Enabling compression of sub-entry messages*

```
Producer producer = environment.producerBuilder()
    .stream("my-stream")
    .batchSize(100) ①
    .subEntrySize(10) ②
    .compression(Compression.ZSTD) ③
    .build();
```

① Set batch size to 100 (the default)

② Set sub-entry size to 10

③ Use the Zstandard compression algorithm

Note the messages in a sub-entry are compressed altogether to benefit from their potential similarity, not one by one.

The following table lists the supported algorithms, general information about them, and the respective implementations used by default.

| Algorithm | Overview | Implementation used |
|---|---|---|
| gzip | Has a high compression ratio but is slow compared to other algorithms. | JDK implementation |

| Algorithm | Overview | Implementation used |
|---|---|---|
| Snappy | Aims for reasonable compression ratio and very high speeds. | Xerial Snappy (framed) |
| LZ4 | Aims for good trade-off between speed and compression ratio. | LZ4 Java (framed) |
| zstd (Zstandard) | Aims for high compression ratio and high speed, especially for decompression. | zstd-jni |

You are encouraged to test and evaluate the compression algorithms depending on your needs.

The compression libraries are pluggable thanks to the `EnvironmentBuilder#compressionCodecFactory(CompressionCodecFactory)` method.

**NOTE**

*Consumers, sub-entry batching, and compression*

There is no configuration required for consumers with regard to sub-entry batching and compression. The broker dispatches messages to client libraries: they are supposed to figure out the format of messages, extract them from their sub-entry, and decompress them if necessary. So when you set up sub-entry batching and compression in your publishers, the consuming applications must use client libraries that support this mode, which is the case for the stream Java client.

## Consumer

`Consumer` is the API to consume messages from a stream.

### Creating a Consumer

A `Consumer` instance is created with `Environment#consumerBuilder()`. The main settings are the stream to consume from, the place in the stream to start consuming from (the *offset*), and a callback when a message is received (the `MessageHandler`). The next snippet shows how to create a `Consumer`:

*Creating a consumer*

```
Consumer consumer = environment.consumerBuilder()  ①
        .stream("my-stream")  ②
        .offset(OffsetSpecification.first())  ③
        .messageHandler((offset, message) -> {
            message.getBodyAsBinary(); ④
        })
        .build();  ⑤
// ...
consumer.close();  ⑥
```

① Use `Environment#consumerBuilder()` to define the consumer

② Specify the stream to consume from

③ Specify where to start consuming from

④ Define behavior on message consumption

⑤ Build the consumer

⑥ Close consumer after usage

The broker start sending messages as soon as the `Consumer` instance is created.

**WARNING**

*The message processing callback can take its time, but not too much*

The message processing callback should not take too long or it could impact other consumers sharing the same connection. The `EnvironmentBuilder#maxConsumersByConnection(int)` method allows isolating consumers from each other, at the cost of creating and maintaining more connections. Consider using a separate thread for long processing (e.g. with an asynchronous `ExecutorService`). Note message processing callbacks run in a dedicated thread, they do not impact other network frames, which run in their own thread.

The following table sums up the main settings to create a `Consumer`:

| Parameter Name | Description | Default |
|---|---|---|
| `stream` | The stream to consume from. | No default, mandatory setting. |
| `offset` | The offset to start consuming from. | `OffsetSpecification#next()` |
| `messageHandler` | The callback for inbound messages. | No default, mandatory setting. |
| `name` | The consumer name (for offset tracking.) | `null` (no offset tracking) |
| `AutoTrackingStrategy` | Enable and configure the auto-tracking strategy. | This is the default tracking strategy if a consumer `name` is provided. |
| `AutoTrackingStrategy#messageCountBeforeStorage` | Number of messages before storing. | 10,000 |
| `AutoTrackingStrategy#flushInterval` | Interval to check and store the last received offset in case of inactivity. | `Duration.ofSeconds(5)` |
| `ManualTrackingStrategy` | Enable and configure the manual tracking strategy. | Disabled by default. |
| `ManualTrackingStrategy#checkInterval` | Interval to check if the last requested stored offset has been actually stored. | `Duration.ofSeconds(5)` |

| Parameter Name | Description | Default |
|---|---|---|
| noTrackingStrategy | Disable server-side offset tracking even if a name is provided. Useful when single active consumer is enabled and an external store is used for offset tracking. | `false` |
| subscriptionListener | A callback before the subscription is created. Useful when using an external store for offset tracking. | `null` |
| `flow` | Configuration helper for flow control. | |
| `flow#initialCredits` | Number of credits when the subscription is created. Increase for higher throughput at the expense of memory usage. | 1 |
| `flow#strategy` | The `ConsumerFlowStrategy` to use. | `ConsumerFlowStrategy#creditOnChunkArrival(1)` |

**NOTE**

*Why is my consumer not consuming?*

A consumer starts consuming at the very end of a stream by default (`next` offset). This means the consumer will receive messages as soon as a producer publishes to the stream. *This also means that if no producers are currently publishing to the stream, the consumer will stay idle, waiting for new messages to come in*. Use the `ConsumerBuilder#offset(OffsetSpecification)` to change the default behavior and see the offset section to find out more about the different types of offset specification.

### Specifying an Offset

The offset is the place in the stream where the consumer starts consuming from. The possible values for the offset parameter are the following:

- `OffsetSpecification.first()`: starting from the first available offset. If the stream has not been truncated, this means the beginning of the stream (offset 0).
- `OffsetSpecification.last()`: starting from the end of the stream and returning the last chunk of messages immediately (if the stream is not empty).
- `OffsetSpecification.next()`: starting from the next offset to be written. Contrary to `OffsetSpecification.last()`, consuming with `OffsetSpecification.next()` will not return anything if no-one is publishing to the stream. The broker will start sending messages to the consumer when messages are published to the stream.
- `OffsetSpecification.offset(offset)`: starting from the specified offset. 0 means consuming from the beginning of the stream (first messages). The client can also specify any number, for example the offset where it left off in a previous incarnation of the application.

- `OffsetSpecification.timestamp(timestamp)`: starting from the messages stored after the specified timestamp. Note consumers can receive messages published a bit before the specified timestamp. Application code can filter out those messages if necessary.

**NOTE**

*What is a chunk of messages?*

A chunk is simply a batch of messages. This is the storage and transportation unit used in RabbitMQ Stream, that is messages are stored contiguously in a chunk and they are delivered as part of a chunk. A chunk can be made of one to several thousands of messages, depending on the ingress.

The following figure shows the different offset specifications in a stream made of 2 chunks:

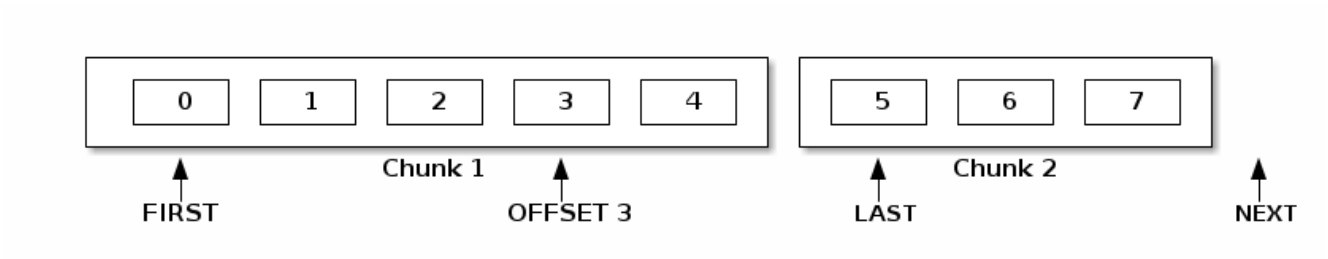


*Figure 1. Offset specifications in a stream made of 2 chunks*

Each chunk contains a timestamp of its creation time. This is this timestamp the broker uses to find the appropriate chunk to start from when using a timestamp specification. The broker chooses the closest chunk *before* the specified timestamp, that is why consumers may see messages published a bit before what they specified.

### Tracking the Offset for a Consumer

RabbitMQ Stream provides server-side offset tracking. This means a consumer can track the offset it has reached in a stream. It allows a new incarnation of the consumer to restart consuming where it left off. All of this without an extra datastore, as the broker stores the offset tracking information.

Offset tracking works in 2 steps:

- the consumer must have a **name**. The name is set with `ConsumerBuilder#name(String)`. The name can be any value (under 256 characters) and is expected to be unique (from the application point of view). Note neither the client library, nor the broker enforces uniqueness of the name: if 2 `Consumer` Java instances share the same name, their offset tracking will likely be interleaved, which applications usually do not expect.
- the consumer must periodically **store the offset** it has reached so far. The way offsets are stored depends on the tracking strategy: automatic or manual.

Whatever tracking strategy you use, **a consumer must have a name to be able to store offsets**.

#### Automatic Offset Tracking

The following snippet shows how to enable automatic tracking with the defaults:

*Using automatic tracking strategy with the defaults*

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .autoTrackingStrategy()   ②
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use automatic tracking strategy with defaults

The automatic tracking strategy has the following available settings:

- **message count before storage**: the client will store the offset after the specified number of messages, right after the execution of the message handler. *The default is every 10,000 messages*.
- **flush interval**: the client will make sure to store the last received offset at the specified interval. This avoids having pending, not stored offsets in case of inactivity. *The default is 5 seconds*.

Those settings are configurable, as shown in the following snippet:

*Configuring the automatic tracking strategy*

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .autoTrackingStrategy()   ②
            .messageCountBeforeStorage(50_000)   ③
            .flushInterval(Duration.ofSeconds(10))   ④
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use automatic tracking strategy

③ Store every 50,000 messages

④ Make sure to store offset at least every 10 seconds

Note the automatic tracking is the default tracking strategy, so if you are fine with its defaults, it is enabled as soon as you specify a name for the consumer:

*Setting only the consumer name to enable automatic tracking*

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set only the consumer name to enable automatic tracking with defaults

Automatic tracking is simple and provides good guarantees. It is nevertheless possible to have more fine-grained control over offset tracking by using manual tracking.

#### Manual Offset Tracking

The manual tracking strategy lets the developer in charge of storing offsets whenever they want, not only after a given number of messages has been received and supposedly processed, like automatic tracking does.

The following snippet shows how to enable manual tracking and how to store the offset at some point:

*Using manual tracking with defaults*

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .manualTrackingStrategy()   ②
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...

          if (conditionToStore()) {
            context.storeOffset();   ③
          }
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use manual tracking with defaults

③ Store the current offset on some condition

Manual tracking has only one setting: the **check interval**. The client checks that the last requested stored offset has been actually stored at the specified interval. *The default check interval is 5 seconds*.

The following snippet shows the configuration of manual tracking:

*Configuring manual tracking strategy*

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .manualTrackingStrategy()   ②
            .checkInterval(Duration.ofSeconds(10))   ③
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...

          if (conditionToStore()) {
            context.storeOffset();   ④
          }
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use manual tracking with defaults

③ Check last requested offset every 10 seconds

④ Store the current offset on some condition

The snippet above uses `MessageHandler.Context#storeOffset()` to store at the offset of the current message, but it is possible to store anywhere in the stream with `MessageHandler.Context#consumer()#store(long)` or simply `Consumer#store(long)`.

#### Considerations On Offset Tracking

*When to store offsets?* Avoid storing offsets too often or, worse, for each message. Even though offset tracking is a small and fast operation, it will make the stream grow unnecessarily, as the broker persists offset tracking entries in the stream itself.

A good rule of thumb is to store the offset every few thousands of messages. Of course, when the consumer will restart consuming in a new incarnation, the last tracked offset may be a little behind the very last message the previous incarnation actually processed, so the consumer may see some messages that have been already processed.

A solution to this problem is to make sure processing is idempotent or filter out the last duplicated messages.

---

*Is the offset a reliable absolute value?* Message offsets may not be contiguous. This implies that the message at offset 500 in a stream may not be the 501 message in the stream (offsets start at 0). There can be different types of entries in a stream storage, a message is just one of them. For example, storing an offset creates an offset tracking entry, which has its own offset.

This means one must be careful when basing some decision on offset values, like a modulo to perform an operation every X messages. As the message offsets have no guarantee to be contiguous, the operation may not happen exactly every X messages.

#### Subscription Listener

The client provides a `SubscriptionListener` interface callback to add behavior before a subscription is created. This callback can be used to customize the offset the client library computed for the subscription. The callback is called when the consumer is first created and when the client has to re-subscribe (e.g. after a disconnection or a topology change).

| | |
|---|---|
| **WARNING** | This API is **experimental**, it is subject to change. |

It is possible to use the callback to get the last processed offset from an external store, that is not using the server-side offset tracking feature RabbitMQ Stream provides. The following code snippet shows how this can be done (note the interaction with the external store is not detailed):

*Using an external store for offset tracking with a subscription listener*

```
Consumer consumer = environment.consumerBuilder()
    .stream("my-stream")
    .subscriptionListener(subscriptionContext -> {  ①
        long offset = getOffsetFromExternalStore();  ②
        subscriptionContext.offsetSpecification(OffsetSpecification.offset(offset + 1
));  ③
    })
    .messageHandler((context, message) -> {
        // message handling code...

        storeOffsetInExternalStore(context.offset());  ④
    })
    .build();
```

① Set subscription listener

② Get offset from external store

③ Set offset to use for the subscription

④ Store the offset in the external store after processing

When using an external store for offset tracking, it is no longer necessary to set a name and an offset strategy, as these only apply when server-side offset tracking is in use.

Using a subscription listener can also be useful to have more accurate offset tracking on re-subscription, at the cost of making the application code slightly more complex. This requires a good understanding on how and when subscription occurs in the client, and so when the subscription listener is called:

- for a consumer with no name (server-side offset tracking *disabled*)
  - on the first subscription (when the consumer is created): the offset specification is the one

specified with `ConsumerBuilder#offset(OffsetSpecification)`, the default being `OffsetSpecification#next()`

  - on re-subscription (after a disconnection or topology change): the offset specification is the offset of the last dispatched message

- for a consumer with a name (server-side offset tracking *enabled*)
  - on the first subscription (when the consumer is created): the server-side stored offset (if any) overrides the value specified with `ConsumerBuilder#offset(OffsetSpecification)`
  - on re-subscription (after a disconnection or topology change): the server-side stored offset is used

The subscription listener comes in handy on re-subscription. The application can track the last processed offset in-memory, with an `AtomicLong` for example. The application knows exactly when a message is processed and updates its in-memory tracking accordingly, whereas the value computed by the client may not be perfectly appropriate on re-subscription.

Let's take the example of a named consumer with an offset tracking strategy that is lagging because of bad timing and a long flush interval. When a glitch happens and triggers the re-subscription, the server-side stored offset can be quite behind what the application actually processed. Using this server-side stored offset can lead to duplicates, whereas using the in-memory, application-specific offset tracking variable is more accurate. A custom `SubscriptionListener` lets the application developer uses what's best for the application if the computed value is not optimal.

#### Flow Control

This section covers how a consumer can tell the broker when to send more messages.

By default, the broker keeps sending messages as long as messages are processed and the `MessageHandler#handle(Context, Message)` method returns. This strategy works fine if message processing is fast enough. If message processing takes longer, one can be tempted to process messages in parallel with an `ExecutorService`. This will make the `handle` method return immediately and the broker will keep sending messages, potentially overflowing the consumer.

What we miss in the parallel processing case is a way to tell the library we are done processing a message and that we are ready at some point to handle more messages. This is the goal of the `MessageHandler.Context#processed()` method.

This method is by default a no-op because the default flow control strategy keeps asking for more messages as soon as message processing is done. This method gets some real behavior to control the flow of messages when an appropriate `ConsumerFlowStrategy` is set `ConsumerBuilder#flow()`. The following code snippet shows how to set a handy consumer flow strategy:

*Setting a consumer flow control strategy*

```
Consumer consumer = environment.consumerBuilder()
    .stream("my-stream")
    .flow()
        .strategy(ConsumerFlowStrategy.creditWhenHalfMessagesProcessed())  ①
    .builder()
```

```
    .messageHandler((context, message) -> {
      // message handling code (possibly asynchronous)...
      context.processed();  ②
    })
    .build();
```

① Set the flow control strategy

② Make sure to call `Context#processed()`

In the example we set up the `creditWhenHalfMessagesProcessed` strategy which asks for more messages once half of the current messages have been marked as processed. The broker does not send messages one by one, it sends chunks of messages. A chunk of messages can contain 1 to several thousands of messages. So with the strategy set above, once `processed()` has been called for half of the messages of the current chunk, the library will ask the broker for another one (it will provide a *credit* for the subscription). By doing this, the next chunk should arrive by the time we are done with the other half of the current chunk. This way the consumer is neither overwhelmed nor idle.

The `ConsumerFlowStrategy` interface provides some static helpers to configure the appropriate strategy.

Additional notes on consumer flow control:

- Make sure to **call the `processed()` method** once you set up a `ConsumerFlowStrategy`. The method is a no-op by default, but it is essential to call it with count-based strategies like `creditWhenHalfMessagesProcessed` or `creditOnProcessedMessageCount`. No calling it will stop the dispatching of messages.
- Make sure to call `processed()` only once. Whether the method is idempotent depends on the flow strategy implementation. Apart from the default one, the implementations the library provides does not make `processed()` idempotent.

#### Single Active Consumer

| | |
|---|---|
| **WARNING** | Single Active Consumer requires **RabbitMQ 3.11** or more. |

When the single active consumer feature is enabled for several consumer instances sharing the same stream and name, only one of these instances will be active at a time and so will receive messages. The other instances will be idle.

The single active consumer feature provides 2 benefits:

- Messages are processed in order: there is only one consumer at a time.
- Consumption continuity is maintained: a consumer from the group will take over if the active one stops or crashes.

A typical sequence of events would be the following:

- Several instances of the same consuming application start up.

- Each application instance registers a single active consumer. The consumer instances share the same name.
- The broker makes the first registered consumer the active one.
- The active consumer receives and processes messages, the other consumer instances remain idle.
- The active consumer stops or crashes.
- The broker chooses the consumer next in line to become the new active one.
- The new active consumer starts receiving messages.

The next figures illustrates this mechanism. There can be only one active consumer:

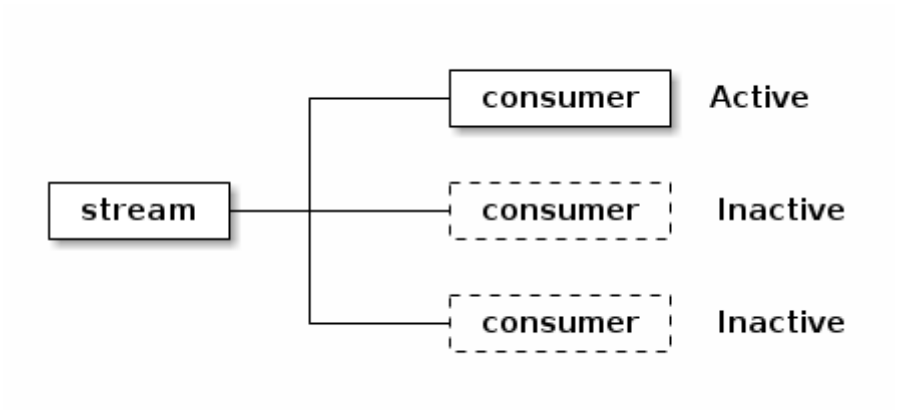


*Figure 2. The first registered consumer is active, the next ones are inactive*

The broker rolls over to another consumer when the active one stops or crashes:

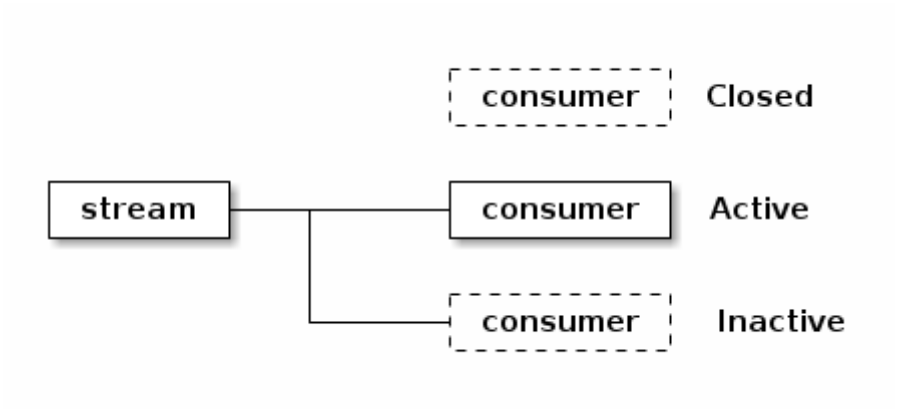


*Figure 3. When the active consumer stops, the next in line becomes active*

Note there can be several groups of single active consumers on the same stream. What makes them different from each other is the name used by the consumers. The broker deals with them independently. Let's use an example. Imagine 2 different `app-1` and `app-2` applications consuming from the same stream, with 3 identical instances each. Each instance registers 1 single active consumer with the name of the application. We end up with 3 `app-1` consumers and 3 `app-2` consumers, 1 active consumer in each group, so overall 6 consumers and 2 active ones, all of this on the same stream.

Let's see now the API for single active consumer.

#### Enabling Single Active Consumer

Use the `ConsumerBuilder#singleActiveConsumer()` method to enable the feature:

*Enabling single active consumer*

```
Consumer consumer = environment.consumerBuilder()
    .stream("my-stream")
    .name("application-1")  ①
    .singleActiveConsumer()  ②
    .messageHandler((context, message) -> {
      // message handling code...
    })
    .build();
```

① Set the consumer name (mandatory to enable single active consumer)

② Enable single active consumer

With the configuration above, the consumer will take part in the `application-1` group on the `my-stream` stream. If the consumer instance is the first in a group, it will get messages as soon as there are some available. If it is not the first in the group, it will remain idle until it is its turn to be active (likely when all the instances registered before it are gone).

#### Offset Tracking

Single active consumer and offset tracking work together: when the active consumer goes away, another consumer takes over and resumes when the former active left off. Well, this is how things should work and luckily this is what happens when using server-side offset tracking. So as long as you use automatic offset tracking or manual offset tracking, the handoff between a former active consumer and the new one will go well.

The story is different is you are using an external store for offset tracking. In this case you need to tell the client library where to resume from and you can do this by implementing the `ConsumerUpdateListener` API.

#### Reacting to Consumer State Change

The broker notifies a consumer that becomes active before dispatching messages to it. The broker expects a response from the consumer and this response contains the offset the dispatching should start from. So this is the consumer's responsibility to compute the appropriate offset, not the broker's. The default behavior is to look up the last stored offset for the consumer on the stream. This works when server-side offset tracking is in use, but it does not when the application chose to use an external store for offset tracking. In this case, it is possible to use the `ConsumerBuilder#consumerUpdateListener(ConsumerUpdateListener)` method like demonstrated in the following snippet:

*Fetching the last stored offset from an external store in the consumer update listener callback*

```
Consumer consumer = environment.consumerBuilder()
    .stream("my-stream")
    .name("application-1")  ①
```

```
    .singleActiveConsumer()  ②
    .noTrackingStrategy()  ③
    .consumerUpdateListener(context -> {  ④
      long offset = getOffsetFromExternalStore();  ⑤
      return OffsetSpecification.offset(offset + 1); ⑥
    })
    .messageHandler((context, message) -> {
      // message handling code...

      storeOffsetInExternalStore(context.offset());
    })
    .build();
```

① Set the consumer name (mandatory to enable single active consumer)

② Enable single active consumer

③ Disable server-side offset tracking

④ Set the consumer update listener

⑤ Fetch last offset from external store

⑥ Return the offset to resume consuming from to the broker

## Super Streams (Partitioned Streams)

**WARNING** Super Streams require **RabbitMQ 3.11** or more.

A super stream is a logical stream made of several individual streams. In essence, a super stream is a partitioned stream that brings scalability compared to a single stream.

The stream Java client uses the same programming model for super streams as with individual streams, that is the `Producer`, `Consumer`, `Message`, etc API are still valid when super streams are in use. Application code should not be impacted whether it uses individual or super streams.

Consuming applications can use super streams and single active consumer at the same time. The 2 features combined make sure only one consumer instance consumes from an individual stream at a time. In this configuration, super streams provide scalability and single active consumer provides the guarantee that messages of an individual stream are processed in order.

**WARNING**

*Super streams do not deprecate streams*

Super streams are a partitioning solution. They are not meant to replace individual streams, they sit on top of them to handle some use cases in a better way. If the stream data is likely to be large – hundreds of gigabytes or even terabytes, size remains relative – and even presents an obvious partition key (e.g. country), a super stream can be appropriate. It can help to cope with the data size and to take advantage of data locality for some processing use cases. Remember that partitioning always comes with complexity though, even if the implementation of super streams strives to make it as transparent as possible for the application developer.

### Topology

A super stream is made of several individual streams, so it can be considered a logical entity rather than an actual physical entity. The topology of a super stream is based on the AMQP 0.9.1 model, that is exchange, queues, and bindings between them. This does not mean AMQP resources are used to transport or store stream messages, it means that they are used to *describe* the super stream topology, that is the streams it is made of.

Let's take the example of an `invoices` super stream made of 3 streams (i.e. partitions):

- an `invoices` exchange represents the super stream
- the `invoices-0`, `invoices-1`, `invoices-2` streams are the partitions of the super stream (streams are also AMQP queues in RabbitMQ)
- 3 bindings between the exchange and the streams link the super stream to its partitions and represent *routing rules*

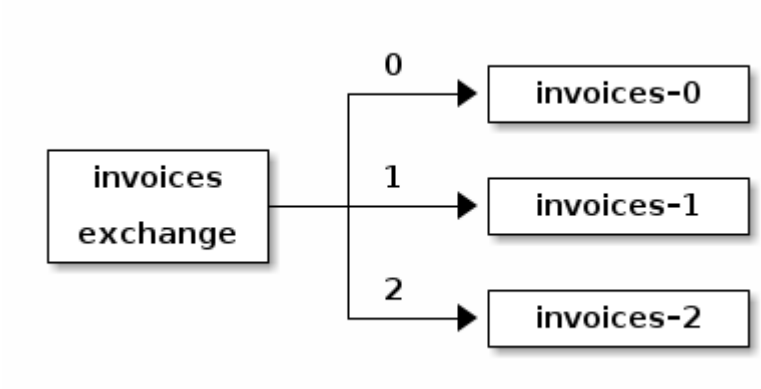


*Figure 4. The topology of a super stream is defined with bindings between an exchange and queues*

When a super stream is in use, the stream Java client queries this information to find out about the partitions of a super stream and the routing rules. From the application code point of view, using a super stream is mostly configuration-based. Some logic must also be provided to extract routing information from messages.

### Super Stream Creation

It is possible to create the topology of a super stream with any AMQP 0.9.1 library or with the management plugin, but the `rabbitmq-streams add_super_stream` command is a handy shortcut. Here is how to create an invoices super stream with 3 partitions:

*Creating a super stream from the CLI*

```
rabbitmq-streams add_super_stream invoices --partitions 3
```

Use `rabbitmq-streams add_super_stream --help` to learn more about the command.

### Publishing to a Super Stream

When the topology of a super stream like the one described above has been set, creating a producer for it is straightforward:

*Creating a Producer for a Super Stream*

```
Producer producer = environment.producerBuilder()
        .superStream("invoices")  ①
        .routing(message -> message.getProperties().getMessageIdAsString()) ②
        .producerBuilder()
        .build();  ③
// ...
producer.close();  ④
```

① Set the super stream name

② Provide the logic to get the routing key from a message

③ Create the producer instance

④ Close the producer when it's no longer necessary

Note that even though the `invoices` super stream is not an actual stream, its name must be used to declare the producer. Internally the client will figure out the streams that compose the super stream. The application code must provide the logic to extract a routing key from a message as a `Function<Message, String>`. The client will hash the routing key to determine the stream to send the message to (using partition list and a modulo operation).

The client uses 32-bit MurmurHash3 by default to hash the routing key. This hash function provides good uniformity, performance, and portability, making it a good default choice, but it is possible to specify a custom hash function:

*Specifying a custom hash function*

```
Producer producer = environment.producerBuilder()
    .superStream("invoices")
    .routing(message -> message.getProperties().getMessageIdAsString())
    .hash(rk -> rk.hashCode())  ①
    .producerBuilder()
    .build();
```

① Use `String#hashCode()` to hash the routing key

Note using Java's `hashCode()` method is a debatable choice as potential producers in other languages are unlikely to implement it, making the routing different between producers in different languages.

#### Resolving Routes with Bindings

Hashing the routing key to pick a partition is only one way to route messages to the appropriate streams. The stream Java client provides another way to resolve streams, based on the routing key *and* the bindings between the super stream exchange and the streams.

This routing strategy makes sense when the partitioning has a business meaning, e.g. with a partition for a region in the world, like in the diagram below:

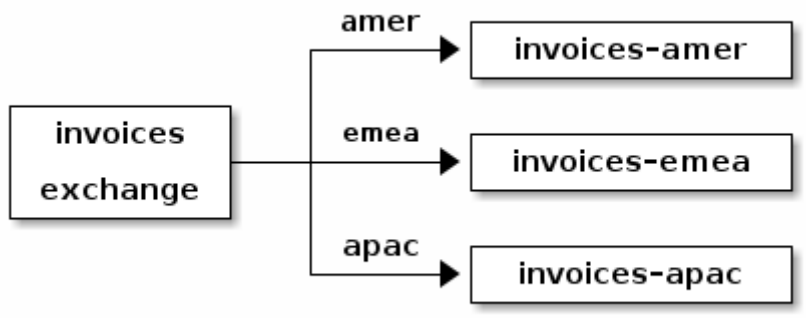


*Figure 5. A super stream with a partition for a region in a world*

In such a case, the routing key will be a property of the message that represents the region:

*Enabling the "key" routing strategy*

```
Producer producer = environment.producerBuilder()
    .superStream("invoices")
    .routing(msg -> msg.getApplicationProperties().get("region").toString())  ①
    .key()  ②
    .producerBuilder()
    .build();
```

① Extract the routing key

② Enable the "key" routing strategy

Internally the client will query the broker to resolve the destination streams for a given routing key, making the routing logic from any exchange type available to streams. Note the client caches results, it does not query the broker for every message.

#### Using a Custom Routing Strategy

The solution that provides the most control over routing is using a custom routing strategy. This should be needed only for specific cases.

Here is an excerpt of the `RoutingStrategy` interface:

*The routing strategy interface*

```
public interface RoutingStrategy {

  /** Where to route a message. */
  List<String> route(Message message, Metadata metadata);

  /** Metadata on the super stream. */
  interface Metadata {

    List<String> partitions();

    List<String> route(String routingKey);
```

```
  }
}
```

Note it is possible to route a message to several streams or even nowhere. The "hash" routing strategy always routes to 1 stream and the "key" routing strategy can route to several streams.

The following code sample shows how to implement a simplistic round-robin `RoutingStrategy` and use it in the producer. Note this implementation should not be used in production as the modulo operation is not sign-safe for simplicity's sake.

*Setting a round-robin routing strategy*

```
AtomicLong messageCount = new AtomicLong(0);
RoutingStrategy routingStrategy = (message, metadata) -> {
    List<String> partitions = metadata.partitions();
    String stream = partitions.get(
        (int) messageCount.getAndIncrement() % partitions.size()
    );
    return Collections.singletonList(stream);
};
Producer producer = environment.producerBuilder()
    .superStream("invoices")
    .routing(null)  ①
    .strategy(routingStrategy)  ②
    .producerBuilder()
    .build();
```

① No need to set the routing key extraction logic

② Set the custom routing strategy

#### Deduplication

Deduplication for a super stream producer works the same way as with a single stream producer. The publishing ID values are spread across the streams but this does affect the mechanism.

### Consuming From a Super Stream

A super stream consumer is a composite consumer: it will look up the super stream partitions and create a consumer for each or them. The programming model is the same as with regular consumers for the application developer: their main job is to provide the application code to process messages, that is a `MessageHandler` instance. The configuration is different though and this section covers its subtleties. But let's focus on the behavior of a super stream consumer first.

#### Super Stream Consumer in Practice

Imagine you have a super stream made of 3 partitions (individual streams). You start an instance of your application, that itself creates a super stream consumer for this super stream. The super stream consumer will create 3 consumers internally, one for each partition, and messages will flow in your `MessageHandler`.

Imagine now that you start another instance of your application. It will do the exact same thing as previously and the 2 instances will process the exact same messages in parallel. This may be not what you want: the messages will be processed twice!

Having one instance of your application may be enough: the data are spread across several streams automatically and the messages from the different partitions are processed in parallel from a single OS process.

But if you want to scale the processing across several OS processes (or bare-metal machines, or virtual machines) and you don't want your messages to be processed several times as illustrated above, you'll have to enable the **single active consumer** feature on your super stream consumer.

The next subsections cover the basic settings of a super stream consumer and a dedicated section covers how super stream consumers and single active consumer play together.

#### Declaring a Super Stream Consumer

Declaring a super stream consumer is not much different from declaring a single stream consumer. The `ConsumerBuilder#superStream(String)` must be used to set the super stream to consume from:

*Declaring a super stream consumer*

```
Consumer consumer = environment.consumerBuilder()
    .superStream("invoices")  ①
    .messageHandler((context, message) -> {
        // message processing
    })
    .build();
// ...
consumer.close();  ②
```

① Set the super stream name

② Close the consumer when it is no longer necessary

That's all. The super stream consumer will take of the details (partitions lookup, coordination of the single consumers, etc).

#### Offset Tracking

The semantic of offset tracking for a super stream consumer are roughly the same as for an individual stream consumer. There are still some subtle differences, so a good understanding of offset tracking in general and of the automatic and manual offset tracking strategies is recommended.

Here are the main differences for the automatic/manual offset tracking strategies between single and super stream consuming:

- **automatic offset tracking**: internally, *the client divides the* `messageCountBeforeStorage` *setting by the number of partitions for each individual consumer*. Imagine a 3-partition super stream, `messageCountBeforeStorage` set to 10,000, and 10,000 messages coming in, perfectly balanced

across the partitions (that is about 3,333 messages for each partition). In this case, the automatic offset tracking strategy will not kick in, because the expected count message has not been reached on any partition. Making the client divide `messageCountBeforeStorage` by the number of partitions can be considered "more accurate" if the message are well balanced across the partitions. A good rule of thumb is to then multiply the expected per-stream `messageCountBeforeStorage` by the number of partitions, to avoid storing offsets too often. So the default being 10,000, it can be set to 30,000 for a 3-partition super stream.

- **manual offset tracking**: the `MessageHandler.Context#storeOffset()` method must be used, the `Consumer#store(long)` will fail, because an offset value has a meaning only in one stream, not in other streams. A call to `MessageHandler.Context#storeOffset()` will store the current message offset in *its* stream, but also the offset of the last dispatched message for the other streams of the super stream.

#### Single Active Consumer Support

| | |
|---|---|
| **WARNING** | Single Active Consumer requires **RabbitMQ 3.11** or more. |

As stated previously, super stream consumers and single active consumer provide scalability and the guarantee that messages of an individual stream are processed in order.

Let's take an example with a 3-partition super stream:

- You have an application that creates a super stream consumer instance with single active consumer enabled.
- You start 3 instances of this application. An instance in this case is a JVM process, which can be in a Docker container, a virtual machine, or a bare-metal server.
- As the super stream has 3 partitions, each application instance will create a super stream consumer that maintains internally 3 consumer instances. That is 9 Java instances of consumer overall. Such a super stream consumer is a *composite consumer*.
- The broker and the different application instances coordinate so that only 1 consumer instance for a given partition receives messages at a time. So among these 9 consumer instances, only 3 are actually *active*, the other ones are idle or *inactive*.
- If one of the application instances stops, the broker will *rebalance* its active consumer to one of the other instances.

The following figure illustrates how the client library supports the combination of the super stream and single active consumer features. It uses a composite consumer that creates an individual consumer for each partition of the super stream. If there is only one single active consumer instance with a given name for a super stream, each individual consumer is active.

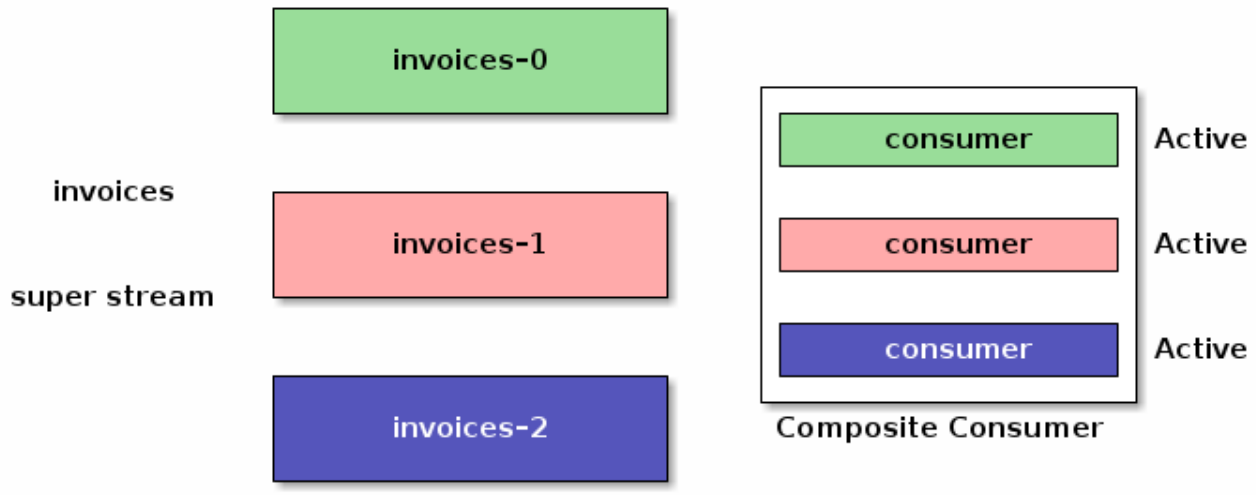


*Figure 6. A single active consumer on a super stream is a composite consumer that creates an individual consumer for each partition*

Imagine now we start 3 instances of the consuming application to scale out the processing. The individual consumer instances spread out across the super stream partitions and only one is active for each partition, as illustrated in the following figure:

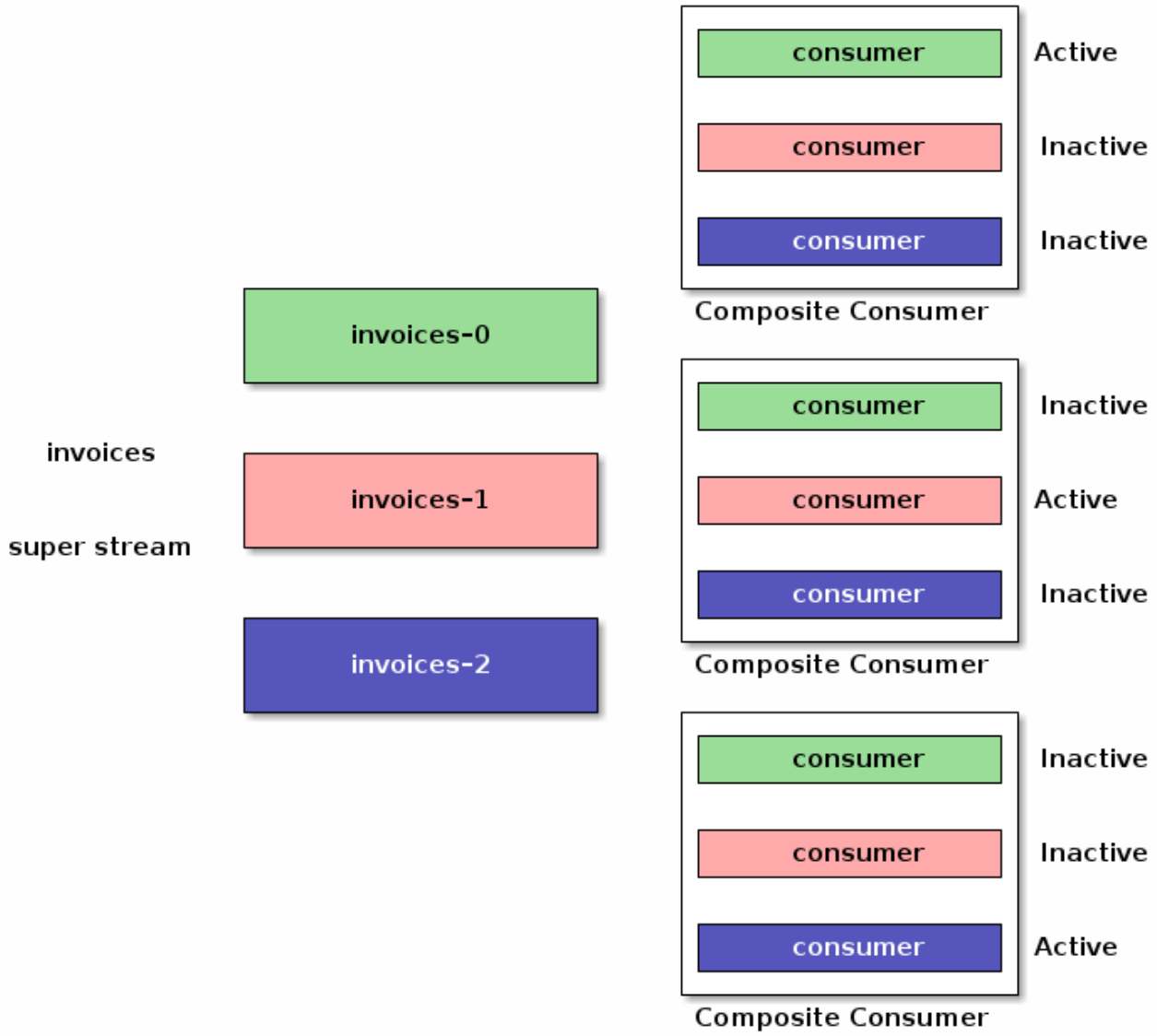


*Figure 7. Consumer instances spread across the super stream partitions and are activated accordingly*

After this overview, let's see the API and the configuration details.

The following snippet shows how to declare a single active consumer on a super stream with the `ConsumerBuilder#superStream(String)` and `ConsumerBuilder#singleActiveConsumer()` methods:

*Enabling single active consumer on a super stream*

```
Consumer consumer = environment.consumerBuilder()
    .superStream("invoices")  ①
    .name("application-1")  ②
    .singleActiveConsumer()  ③
    .messageHandler((context, message) -> {
        // message processing
    })
    .build();
// ...
```

① Set the super stream name

② Set the consumer name (mandatory to enable single active consumer)

③ Enable single active consumer

Note it is mandatory to specify a name for the consumer. This name will be used to identify the *group* of consumer instances and make sure only one is active for each partition. The name is also the reference for offset tracking.

The example above uses by default automatic offset tracking. With this strategy, the client library takes care of offset tracking when consumers become active or inactive. It looks up the latest stored offset when a consumer becomes active to start consuming at the appropriate offset and it stores the last dispatched offset when a consumer becomes inactive.

The story is not the same with manual offset tracking as the client library does not know which offset it should store when a consumer becomes inactive. The application developer can use the `ConsumerUpdateListener)` callback to react appropriately when a consumer changes state. The following snippet illustrates the use of the `ConsumerUpdateListener` callback:

*Using manual offset tracking for a super stream single active consumer*

```
Consumer consumer =
    environment.consumerBuilder()
        .superStream("invoices")  ①
        .name("application-1")  ②
        .singleActiveConsumer()  ③
        .manualTrackingStrategy()  ④
        .builder()
        .consumerUpdateListener(context -> {  ⑤
           if(context.isActive()) {  ⑥
               try {
                   return OffsetSpecification.offset(
                       context.consumer().storedOffset() + 1
                   );
               } catch (NoOffsetException e) {
                   return OffsetSpecification.next();
               }
           } else {
               context.consumer().store(lastProcessedOffsetForThisStream);  ⑦
               return null;
           }
        })
        .messageHandler((context, message) -> {
            // message handling code...

            if (conditionToStore()) {
                context.storeOffset();  ⑧
            }
        })
        .build();
```

```
// ...
```

① Set the super stream name

② Set the consumer name (mandatory to enable single active consumer)

③ Enable single active consumer

④ Enable manual offset tracking strategy

⑤ Set `ConsumerUpdateListener`

⑥ Return stored offset + 1 or default when consumer becomes active

⑦ Store last processed offset for the stream when consumer becomes inactive

⑧ Store the current offset on some condition

The `ConsumerUpdateListener` callback must return the offset to start consuming from when a consumer becomes active. This is what the code above does: it checks if the consumer is active with `ConsumerUpdateListener.Context#isActive()` and looks up the last stored offset. If there is no stored offset yet, it returns a default value, `OffsetSpecification#next()` here.

When a consumer becomes inactive, it should store the last processed offset, as another consumer instance will take over elsewhere. It is expected this other consumer runs the exact same code, so it will execute the same sequence when it becomes active (looking up the stored offset, returning the value + 1).

Note the `ConsumerUpdateListener` is called for a *partition*, that is an individual stream. The application code should take care of maintaining a reference of the last processed offset for each partition of the super stream, e.g. with a `Map<String, Long>` (partition-to-offset map). To do so, the `context` parameter of the `MessageHandler` and `ConsumerUpdateListener` callbacks provide a `stream()` method.

RabbitMQ Stream provides server-side offset tracking, but it is possible to use an external store to track offsets for streams. The `ConsumerUpdateListener` callback is still your friend in this case. The following snippet shows how to leverage it when an external store is in use:

*Using external offset tracking for a super stream single active consumer*

```
Consumer consumer = environment.consumerBuilder()
    .superStream("invoices")  ①
    .name("application-1")  ②
    .singleActiveConsumer()  ③
    .noTrackingStrategy()  ④
    .consumerUpdateListener(context -> {  ⑤
      if (context.isActive()) {  ⑥
        long offset = getOffsetFromExternalStore();
        return OffsetSpecification.offset(offset + 1);
      }
      return null;  ⑦
    })
    .messageHandler((context, message) -> {
      // message handling code...
```

```
      storeOffsetInExternalStore(context.stream(), context.offset());  ⑧
    })
    .build();
```

① Set the super stream name

② Set the consumer name (mandatory to enable single active consumer)

③ Enable single active consumer

④ Disable server-side offset tracking

⑤ Set `ConsumerUpdateListener`

⑥ Use external store for stored offset when consumer becomes active

⑦ Assume offset already stored when consumer becomes inactive

⑧ Use external store for offset tracking

Here are the takeaway points of this code:

- Even though there is no server-side offset tracking to use it, the consumer must still have a name to identify the group it belongs to. The external offset tracking mechanism is free to use the same name or not.
- Calling `ConsumerBuilder#noTrackingStrategy()` is necessary to disable server-side offset tracking, or the automatic tracking strategy will kick in.
- The snippet does not provide the details, but the offset tracking mechanism seems to store the offset for each message. The external store must be able to cope with the message rate in a real-world scenario.
- The `ConsumerUpdateListener` callback returns the last stored offset + 1 when the consumer becomes active. This way the broker will resume the dispatching at this location in the stream.
- A well-behaved `ConsumerUpdateListener` must make sure the last processed offset is stored when the consumer becomes inactive, so that the consumer that will take over can look up the offset and resume consuming at the right location. Our `ConsumerUpdateListener` does not do anything when the consumer becomes inactive (it returns `null`): it can afford this because the offset is stored for each message. Make sure to store the last processed offset when the consumer becomes inactive to avoid duplicates when the consumption resumes elsewhere.

# Advanced Topics

## Filtering

| | |
|---|---|
| **WARNING** | Filtering requires **RabbitMQ 3.13** or more. |

RabbitMQ Stream provides a server-side filtering feature that avoids reading all the messages of a stream and filtering only on the client side. This helps to save network bandwidth when a consuming application needs only a subset of messages, e.g. the messages from a given geographical region.

The filtering feature works as follows:

- each message is published with an associated *filter value*
- a consumer that wants to enable filtering must:
  - define one or several filter values
  - define some client-side filtering logic

Why does the consumer need to define some client-side filtering logic? Because the server-side filtering is probabilistic: messages that do not match the filter value(s) can still be sent to the consumer. The server uses a Bloom filter, *a space-efficient probabilistic data structure*, where false positives are possible. Despite this, the filtering saves some bandwidth, which is its primary goal.

#### Filtering on the Publishing Side

Filtering on the publishing side consists in defining some logic to extract the filter value from a message. The following snippet shows how to extract the filter value from an application property:

*Declaring a producer with logic to extract a filter value from each message*

```
Producer producer = environment.producerBuilder()
  .stream("invoices")
  .filterValue(msg ->
    msg.getApplicationProperties().get("state").toString())  ①
  .build();
```

① Get filter value from `state` application property

Note the filter value can be null: the message is then published in a regular way. It is called in this context an *unfiltered* message.

#### Filtering on the Consuming Side

A consumer needs to set up one or several filter values and some filtering logic to enable filtering. The filtering logic must be consistent with the filter values. In the next snippet, the consumer wants to process only messages from the state of California. It sets a filter value to `california` and a predicate that accepts a message only if the `state` application properties is `california`:

*Declaring a consumer with a filter value and filtering logic*

```
String filterValue = "california";
Consumer consumer = environment.consumerBuilder()
  .stream("invoices")
  .filter()
    .values(filterValue)  ①
    .postFilter(msg ->
      filterValue.equals(msg.getApplicationProperties().get("state")))  ②
  .builder()
  .messageHandler((ctx, msg) -> { })
  .build();
```

① Set filter value

② Set filtering logic

The filter logic is a `Predicate<Message>`. It must return `true` if a message is accepted, following the same semantics as `java.util.stream.Stream#filter(Predicate)`.

As stated above, not all messages must have an associated filter value. Many applications may not need some filtering, so they can publish messages the regular way. So a stream can contain messages with and without an associated filter value.

By default, messages without a filter value (a.k.a *unfiltered* messages) are not sent to a consumer that enabled filtering.

But what if a consumer wants to process messages with a filter value and messages without any filter value as well? It must use the `matchUnfiltered()` method in its declaration and also make sure to keep the filtering logic consistent:

*Getting unfiltered messages as well when enabling filtering*

```
String filterValue = "california";
Consumer consumer = environment.consumerBuilder()
  .stream("invoices")
  .filter()
    .values(filterValue)  ①
    .matchUnfiltered()  ②
    .postFilter(msg ->
        filterValue.equals(msg.getApplicationProperties().get("state"))
        || !msg.getApplicationProperties().containsKey("state")  ③
    )
  .builder()
  .messageHandler((ctx, msg) -> { })
  .build();
```

① Request messages from California

② Request messages without a filter value as well

③ Let both types of messages pass

In the example above, the filtering logic has been adapted to let pass `california` messages *and* messages without a state set as well.

#### Considerations on Filtering

As stated previously, the server can send messages that do not match the filter value(s) set by consumers. This is why application developers must be very careful with the filtering logic they define to avoid processing unwanted messages.

What are good candidates for filter values? Unique identifiers are *not*: if you know a given message property will be unique in a stream, do not use it as a filter value. A defined set of values shared across the messages is a good candidate: geographical locations (e.g. countries, states), document

types in a stream that stores document information (e.g. payslip, invoice, order), categories of products (e.g. book, luggage, toy).

Cardinality of filter values can be from a few to a few thousands. Extreme cardinality (a couple or dozens of thousands) can make filtering less efficient.

### Using Native `epoll`

The stream Java client uses the Netty network framework and its Java NIO transport implementation by default. This should be a reasonable default for most applications.

Netty also allows using JNI transports. They are less portable than Java NIO, but they can be more performant for some workloads (even though the RabbitMQ team has not seen any significant improvement in their own tests).

The Linux `epoll` transport is a popular choice, so we'll see how to configure with the stream Java client. Other JNI transports can be configured in the same way.

The native transport dependency must be added to the dependency manager. We must pull the native binaries compiled for our OS and architecture, in our example Linux x86-64, so we are using the `linux-x86_64` classifier. Here is the declaration for Maven:

*Declaring the Linux x86-64 native `epoll` transport dependency with Maven*

```
<dependencies>

  <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <version>4.1.95.Final</version>
    <classifier>linux-x86_64</classifier>
  </dependency>

</dependencies>
```

And for Gradle:

*Declaring the Linux x86-64 native `epoll` transport dependency with Gradle*

```
dependencies {
  compile "io.netty:netty-transport-native-epoll:4.1.95.Final:linux-x86_64"
}
```

The native `epoll` transport is set up when the environment is configured:

*Configuring the native `epoll` transport in the environment*

```
EventLoopGroup epollEventLoopGroup = new EpollEventLoopGroup();  ①
Environment environment = Environment.builder()
    .netty()  ②
```

```
        .eventLoopGroup(epollEventLoopGroup)  ③
        .bootstrapCustomizer(b -> b.channel(EpollSocketChannel.class))  ④
        .environmentBuilder()
    .build();
```

① Create the `epoll` event loop group (don't forget to close it!)

② Use the Netty configuration helper

③ Set the event loop group

④ Set the channel class to use

Note the event loop group must be closed explicitly: the environment will not close it itself as it is provided externally.

## Building the Client

You need JDK 1.8 or more installed.

To build the JAR file:

```
./mvnw clean package -DskipITs -DskipTests
```

To launch the test suite (requires a local RabbitMQ node with stream plugin enabled):

```
./mvnw verify -Drabbitmqctl.bin=/path/to/rabbitmqctl
```

# The Performance Tool

The library contains also a performance tool to test the RabbitMQ Stream plugin. It is usable as an uber JAR downloadable from GitHub Release or as a Docker image. It can be built separately as well.

Snapshots are on GitHub release as well. Use the `pivotalrabbitmq/stream-perf-test:dev` image to use the latest snapshot in Docker.

## Using the Performance Tool

### With Docker

The performance tool is available as a Docker image. You can use the Docker image to list the available options:

*Listing the available options of the performance tool*

```
docker run -it --rm pivotalrabbitmq/stream-perf-test --help
```

There are all sorts of options, if none is provided, the tool will start publishing to and consuming from a stream created only for the test.

When using Docker, the container running the performance tool must be able to connect to the broker, so you have to figure out the appropriate Docker configuration to make this possible. You can have a look at the Docker network documentation to find out more.

**NOTE**

*Docker on macOS*

Docker runs on a virtual machine when using macOS, so do not expect high performance when using RabbitMQ Stream and the performance tool inside Docker on a Mac.

We show next a couple of options to easily use the Docker image.

#### With Docker Host Network Driver

This is the simplest way to run the image locally, with a local broker running in Docker as well. The containers use the host network, this is perfect for experimenting locally.

*Running the broker and performance tool with the host network driver*

```
# run the broker
docker run -it --rm --name rabbitmq --network host rabbitmq:3.12
# open another terminal and enable the stream plugin
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
# run the performance tool
docker run -it --rm --network host pivotalrabbitmq/stream-perf-test
```

**NOTE**

*Docker Host Network Driver Support*

According to Docker's documentation, the host networking driver **only works on Linux hosts**. Nevertheless, the commands above work on some Mac hosts.

#### With Docker Bridge Network Driver

Containers need to be able to communicate with each other with the bridge network driver, this can be done by defining a network and running the containers in this network.

*Running the broker and performance tool with the bridge network driver*

```
# create a network
docker network create stream-perf-test
# run the broker
docker run -it --rm --network stream-perf-test --name rabbitmq rabbitmq:3.12
# open another terminal and enable the stream plugin
docker exec rabbitmq rabbitmq-plugins enable rabbitmq_stream
# run the performance tool
docker run -it --rm --network stream-perf-test pivotalrabbitmq/stream-perf-test \
    --uris rabbitmq-stream://rabbitmq:5552
```

### With the Java Binary

The Java binary is available on GitHub Release. Snaphots are available as well. To use the latest snapshot:

```
wget https://github.com/rabbitmq/rabbitmq-java-tools-binaries-dev/releases/download/v-
stream-perf-test-latest/stream-perf-test-latest.jar
```

To launch a run:

```
$ java -jar stream-perf-test-latest.jar
17:51:26.207 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting producer
1, published 560277 msg/s, confirmed 554088 msg/s, consumed 556983 msg/s, latency
min/median/75th/95th/99th 2663/9799/13940/52304/57995 µs, chunk size 1125
2, published 770722 msg/s, confirmed 768209 msg/s, consumed 768585 msg/s, latency
min/median/75th/95th/99th 2454/9599/12206/23940/55519 µs, chunk size 1755
3, published 915895 msg/s, confirmed 914079 msg/s, consumed 916103 msg/s, latency
min/median/75th/95th/99th 2338/8820/11311/16750/52985 µs, chunk size 2121
4, published 1004257 msg/s, confirmed 1003307 msg/s, consumed 1004981 msg/s, latency
min/median/75th/95th/99th 2131/8322/10639/14368/45094 µs, chunk size 2228
5, published 1061380 msg/s, confirmed 1060131 msg/s, consumed 1061610 msg/s, latency
min/median/75th/95th/99th 2131/8247/10420/13905/37202 µs, chunk size 2379
6, published 1096345 msg/s, confirmed 1095947 msg/s, consumed 1097447 msg/s, latency
min/median/75th/95th/99th 2131/8225/10334/13722/33109 µs, chunk size 2454
7, published 1127791 msg/s, confirmed 1127032 msg/s, consumed 1128039 msg/s, latency
min/median/75th/95th/99th 1966/8150/10172/13500/23940 µs, chunk size 2513
8, published 1148846 msg/s, confirmed 1148086 msg/s, consumed 1149121 msg/s, latency
min/median/75th/95th/99th 1966/8079/10135/13248/16771 µs, chunk size 2558
9, published 1167067 msg/s, confirmed 1166369 msg/s, consumed 1167311 msg/s, latency
min/median/75th/95th/99th 1966/8063/9986/12977/16757 µs, chunk size 2631
10, published 1182554 msg/s, confirmed 1181938 msg/s, consumed 1182804 msg/s, latency
min/median/75th/95th/99th 1966/7963/9949/12632/16619 µs, chunk size 2664
11, published 1197069 msg/s, confirmed 1196495 msg/s, consumed 1197291 msg/s, latency
min/median/75th/95th/99th 1966/7917/9955/12503/15386 µs, chunk size 2761
12, published 1206687 msg/s, confirmed 1206176 msg/s, consumed 1206917 msg/s, latency
min/median/75th/95th/99th 1966/7893/9975/12503/15280 µs, chunk size 2771
...
^C
Summary: published 1279444 msg/s, confirmed 1279019 msg/s, consumed 1279019 msg/s,
latency 95th 12161 µs, chunk size 2910
```

The previous command will start publishing to and consuming from a `stream` stream that will be created. The tool outputs live metrics on the console and write more detailed metrics in a `stream-perf-test-current.txt` file that get renamed to `stream-perf-test-yyyy-MM-dd-HHmmss.txt` when the run ends.

To see the options:

```
java -jar stream-perf-test-latest.jar --help
```

The performance tool comes also with a completion script. You can download it and enable it in your `~/.zshrc` file:

```
alias stream-perf-test='java -jar target/stream-perf-test.jar'
source ~/.zsh/stream-perf-test_completion
```

Note the activation requires an alias which must be `stream-perf-test`. The command can be anything though.

## Common Usage

### Connection

The performance tool connects by default to localhost, on port 5552, with default credentials (`guest`/`guest`), on the default `/` virtual host. This can be changed with the `--uris` option:

```
java -jar stream-perf-test.jar --uris rabbitmq-stream://rabbitmq-1:5552
```

The URI follows the same rules as the AMQP 0.9.1 URI, except the protocol must be `rabbitmq-stream`. The next command shows how to set up the different elements of the URI:

```
java -jar stream-perf-test.jar \
  --uris rabbitmq-stream://guest:guest@localhost:5552/%2f
```

The option accepts several values, separated by commas. By doing so, the tool will be able to pick another URI for its "locator" connection, in case a node crashes:

```
java -jar stream-perf-test.jar \
  --uris rabbitmq-stream://rabbitmq-1:5552,rabbitmq-stream://rabbitmq-2:5552
```

Note the tool uses those URIs only for management purposes, it does not use them to distribute publishers and consumers across a cluster.

It is also possible to enable TLS by using the `rabbitmq-stream+tls` scheme:

```
java -jar stream-perf-test.jar \
  --uris rabbitmq-stream+tls://guest:guest@localhost:5551/%2f
```

Note the performance tool will automatically configure the client to trust all server certificates and to not use a private key (for client authentication).

Have a look at the connection logic section in case of connection problem.

#### Publishing Rate

It is possible to limit the publishing rate with the `--rate` option:

```
java -jar stream-perf-test.jar --rate 10000
```

RabbitMQ Stream can easily saturate the resources of the hardware, it can especially max out the storage IO. Reasoning when a system is under severe constraints can be difficult, so setting a low publishing rate can be a good idea to get familiar with the performance tool and the semantics of streams.

#### Number of Producers and Consumers

You can set the number of producers and consumers with the `--producers` and `--consumers` options, respectively:

```
java -jar stream-perf-test.jar --producers 5 --consumers 5
```

With the previous command, you should see a higher consuming rate than publishing rate. It is because the 5 producers publish as fast as they can and each consumer consume the messages from the 5 publishers. In theory the consumer rate should be 5 times the publishing rate, but as stated previously, the performance tool may put the broker under severe constraints, so the numbers may not add up.

You can set a low publishing rate to verify this theory:

```
java -jar stream-perf-test.jar --producers 5 --consumers 5 --rate 10000
```

With the previous command, each publisher should publish 10,000 messages per second, that is 50,000 messages per second overall. As each consumer consumes each published messages, the consuming rate should be 5 times the publishing rate, that is 250,000 messages per second. Using a small publishing rate should let plenty of resources to the system, so the rates should tend towards those values.

#### Streams

The performance tool uses a `stream` stream by default, the `--streams` option allows specifying streams that the tool will try to create. Note producer and consumer counts must be set accordingly, as they are not spread across the stream automatically. The following command will run a test with 3 streams, with a producer and a consumer on each of them:

```
java -jar stream-perf-test.jar --streams stream1,stream2,stream3 \
                               --producers 3 --consumers 3
```

The stream creation process has the following semantics:

- the tool always tries to create streams.
- if the target streams already exist and have the exact same properties as the ones the tool uses (see retention below), the run will start normally as stream creation is idempotent.
- if the target streams already exist but do not have the exact same properties as the ones the tool uses, the run will start normally as well, the tool will output a warning.
- for any other errors during creation, the run will stop.
- the streams are not deleted after the run.
- if you want the tool to delete the streams after a run, use the `--delete-streams` flag.

Specifying streams one by one can become tedious as their number grows, so the `--stream-count` option can be combined with the `--streams` option to specify a number or a range and a stream name pattern, respectively. The following table shows the usage of these 2 options and the resulting exercised streams. Do not forget to also specify the appropriate number of producers and consumers if you want all the declared streams to be used.

| Options | Computed Streams | Details |
|---|---|---|
| `--stream-count 5 --streams stream` | `stream-1,stream-2,stream-3,stream-4,stream-5` | Stream count starts at 1. |
| `--stream-count 5 --streams stream-%d` | `stream-1,stream-2,stream-3,stream-4,stream-5` | Possible to specify a Java printf-style format string. |
| `--stream-count 10 --streams stream-%d` | `stream-1,stream-2,stream-3,⋯, stream-10` | Not bad, but not correctly sorted alphabetically. |
| `--stream-count 10 --streams stream-%02d` | `stream-01,stream-02,stream-03,⋯, stream-10` | Better for sorting. |
| `--stream-count 10 --streams stream` | `stream-01,stream-02,stream-03,⋯, stream-10` | The default format string handles the sorting issue. |
| `--stream-count 50-500 --streams stream-%03d` | `stream-050,stream-051,stream-052,⋯, stream-500` | Ranges are accepted. |
| `--stream-count 50-500` | `stream-050,stream-051,stream-052,⋯, stream-500` | Default format string. |

#### Publishing Batch Size

The default publishing batch size is 100, that is a publishing frame is sent every 100 messages. The following command sets the batch size to 50 with the `--batch-size` option:

```
java -jar stream-perf-test.jar --batch-size 50
```

There is no ideal batch size, it is a tradeoff between throughput and latency. High batch size values should increase throughput (usually good) and latency (usually not so good), whereas low batch size should decrease throughput (usually not good) and latency (usually good).

#### Unconfirmed Messages

A publisher can have at most 10,000 unconfirmed messages at some point. If it reaches this value, it has to wait until the broker confirms some messages. This avoids fast publishers overwhelming the broker. The `--confirms` option allows changing the default value:

```
java -jar stream-perf-test.jar --confirms 20000
```

High values should increase throughput at the cost of consuming more memory, whereas low values should decrease throughput and memory consumption.

#### Message Size

The default size of a message is 10 bytes, which is rather small. The `--size` option lets you specify a different size, usually higher, to have a value close to your use case. The next command sets a size of 1 KB:

```
java -jar stream-perf-test.jar --size 1024
```

Note the message body size cannot be smaller that 8 bytes, as the performance tool stores a long in each message to calculate the latency. Note also the actual size of a message will be slightly higher, as the body is wrapped in an AMQP 1.0 message.

#### Connection Pooling

The performance tool does not use connection pooling by default: each producer and consumer has its own connection. This can be appropriate to reach maximum throughput in performance test runs, as producers and consumers do not share connections. But it may not always reflect what applications do, as they may have slow producers and not-so-busy consumers, so sharing connections becomes interesting to save some resources.

It is possible to configure connection pooling with the `--producers-by-connection` and `--consumers-by-connection` options. They accept a value between 1 and 255, the default being 1 (no connection pooling).

In the following example we use 10 streams with 1 producer and 1 consumer on each of them. As the rate is low, we can re-use connections:

```
java -jar stream-perf-test.jar --producers 10 --consumers 10 --stream-count 10 \
                               --rate 1000 \
                               --producers-by-connection 50 --consumers-by-connection
50
```

We end up using 2 connections for the producers and consumers with connection pooling, instead of 20.

## Advanced Usage

### Retention

If you run performance tests for a long time, you might be interested in setting a retention strategy for the streams the performance tool creates for a run. This would typically avoid saturating the storage devices of your servers. The default values are 20 GB for the maximum size of a stream and 500 MB for each segment files that composes a stream. You can change these values with the `--max-length-bytes` and `--stream-max-segment-size-bytes` options:

```
java -jar stream-perf-test.jar --max-length-bytes 10gb \
                               --stream-max-segment-size-bytes 250mb
```

Both options accept units (`kb`, `mb`, `gb`, `tb`), as well as no unit to specify a number of bytes.

It is also possible to use the time-based retention strategy with the `--max-age` option. This can be less predictable than `--max-length-bytes` in the context of performance tests though. The following command shows how to set the maximum age of segments to 5 minutes with a maximum segment size of 250 MB:

```
java -jar stream-perf-test.jar --max-age PT5M \
                               --stream-max-segment-size-bytes 250mb
```

The `--max-age` option uses the ISO 8601 duration format.

### Offset (Consumer)

Consumers start by default at the very end of a stream (offset `next`). It is possible to specify an offset to start from with the `--offset` option, if you have existing streams, and you want to consume from them at a specific offset. The following command sets the consumer to start consuming at the beginning of a stream:

```
java -jar stream-perf-test.jar --offset first
```

The accepted values for `--offset` are `first`, `last`, `next` (the default), an unsigned long for a given offset, and an ISO 8601 formatted timestamp (eg. `2020-06-03T07:45:54Z`).

### Offset Tracking (Consumer)

A consumer can track the point it has reached in a stream to be able to restart where it left off in a new incarnation. The performance tool has the `--store-every` option to tell consumers to store the offset every `x` messages to be able to measure the impact of offset tracking in terms of throughput and storage. This feature is disabled by default. The following command shows how to store the offset every 100,000 messages:

```
java -jar stream-perf-test.jar --store-every 100000
```

#### Consumer Names

When using `--store-every` (see above) for offset tracking, the performance tool uses a default name using the pattern `{stream-name}-{consumer-number}`. So the default name of a single tracking consumer consuming from `stream` will be `stream-1`.

The consumer names pattern can be set with the `--consumer-names` option, which uses the Java printf-style format string. The stream name and the consumer number are injected as arguments, in this order.

The following table illustrates some examples for the `--consumer-names` option for a `s1` stream and a second consumer:

| Option | Computed Name | Details |
|---|---|---|
| `%s-%d` | `s1-2` | Default pattern. |
| `stream-%s-consumer-%d` | `stream-s1-consumer-2` | |
| `consumer-%2$d-on-stream-%1$s` | `consumer-2-on-stream-s1` | The argument indexes (`1$` for the stream, `2$` for the consumer number) must be used as the pattern uses the consumer number first, which is not the pre-defined order of arguments. |
| `uuid` | `7cc75659-ea67-4874-96ef-151a505e1a55` | Random UUID that changes for every run. |

Note you can use `--consumer-names uuid` to change the consumer names for every run. This can be useful when you want to use tracking consumers in different runs but you want to force the offset they start consuming from. With consumer names that do not change between runs, tracking consumers would ignore the specified offset and would start where they left off (this is the purpose of offset tracking).

#### Producer Names

You can use the `--producer-names` option to set the producer names pattern and therefore enable message deduplication (using the default publishing sequence starting at 0 and incremented for each message). The same naming options apply as above in consumer names with the only difference that the default pattern is empty (i.e. no deduplication).

Here is an example of the usage of the `--producer-names` option:

```
java -jar stream-perf-test.jar --producer-names %s-%d
```

The run will start one producer and will use the `stream-1` producer reference (default stream is `stream` and the number of the producer is 1.)

#### Load Balancer in Front of the Cluster

A load balancer can misguide the performance tool when it tries to connect to nodes that host stream leaders and replicas. The "Connecting to Streams" blog post covers why client applications must connect to the appropriate nodes in a cluster.

Use the `--load-balancer` flag to make sure the performance tool always goes through the load balancer that sits in front of your cluster:

```
java -jar stream-perf-test.jar --uris rabbitmq-stream://my-load-balancer:5552 \
                               --load-balancer
```

The same blog post covers why a load balancer can make things more complicated for client applications like the performance tool and how they can mitigate these issues.

#### Single Active Consumer

If the `--single-active-consumer` flag is set, the performance tool will create single active consumer instances. This means that if there are more consumers than streams, there will be only one active consumer at a time on a stream, *if they share the same name*. Note offset tracking gets enabled automatically if it's not with `--single-active-consumer` (using 10,000 for `--store-every`). Let's see a couple of examples.

In the following command we have 1 producer publishing to 1 stream and 3 consumers on this stream. As `--single-active-consumer` is used, only one of these consumers will be active at a time.

```
java -jar stream-perf-test.jar --producers 1 --consumers 3 --single-active-consumer \
                               --consumer-names my-app
```

Note we use a fixed value for the consumer names: if they don't have the same name, the broker will not consider them as a group of consumers, so they will all get messages, like regular consumers.

In the following example we have 2 producers for 2 streams and 6 consumers overall (3 for each stream). Note the consumers have the same name on their streams with the use of `--consumer-names my-app-%s`, as `%s` is a placeholder for the stream name.

```
java -jar stream-perf-test.jar --producers 2 --consumers 6 --stream-count 2 \
                               --single-active-consumer --consumer-names my-app-%s
```

#### Super Streams

The performance tool has a `--super-streams` flag to enable super streams on the publisher and consumer sides. This support is meant to be used with the `--single-active-consumer` flag, to benefit from both features. We recommend reading the appropriate sections of the documentation to understand the semantics of the flags before using them. Let's see some examples.

The example below creates 1 producer and 3 consumers on the default `stream`, which is now a *super stream* because of the `--super-streams` flag:

```
java -jar stream-perf-test.jar --producers 1 --consumers 3 --single-active-consumer \
                               --super-streams --consumer-names my-app
```

The performance tool creates 3 individual streams by default, they are the partitions of the super stream. They are named `stream-0`, `stream-1`, and `stream-2`, after the name of the super stream, `stream`. The producer will publish to each of them, using a hash-based routing strategy.

A consumer is *composite* with `--super-streams`: it creates a consumer instance for each partition. This is 9 consumer instances overall – 3 composite consumers and 3 partitions – spread evenly across the partitions, but with only one active at a time on a given stream.

Note we use a fixed consumer name so that the broker considers the consumers belong to the same group and enforce the single active consumer behavior.

The next example is more convoluted. We are going to work with 2 super streams (`--stream-count 2` and `--super-streams`). Each super stream will have 5 partitions (`--super-stream-partitions 5`), so this is 10 streams overall (`stream-1-0` to `stream-1-4` and `stream-2-0` to `stream-2-4`). Here is the command line:

```
java -jar stream-perf-test.jar --producers 2 --consumers 6 --stream-count 2 \
                               --super-streams --super-stream-partitions 5 \
                               --single-active-consumer \
                               --consumer-names my-app-%s
```

We see also that each super stream has 1 producer (`--producers 2`) and 3 consumers (`--consumers 6`). The composite consumers will spread their consumer instances across the partitions. Each partition will have 3 consumers but only 1 active at a time with `--single-active-consumer` and `--consumer-names my-app-%s` (the consumers on a given stream have the same name, so the broker make sure only one consumes at a time).

Note the performance tool does not use connection pooling by default. The command above opens a significant number of connections – 30 just for consumers – and may not reflect exactly how applications are deployed in the real world. Don't hesitate to use the `--producers-by-connection` and `--consumers-by-connection` options to make the runs as close to your workloads as possible.

#### Monitoring

The tool can expose some runtime information on HTTP. The default port is 8080. The following options are available:

- `--monitoring`: add a `threaddump` endpoint to display a thread dump of the process. This can be useful to inspect threads if the tool seems blocked.
- `--prometheus`: add a `metrics` endpoint to expose metrics using the Prometheus format. The endpoint can then be declared in a Prometheus instance to scrape the metrics.

- `--monitoring-port`: set the port to use for the web server.

**Synchronizing Several Instances**

> **NOTE** This feature is available only on Java 11 or more.

Instances of the performance tool can synchronize to start at the same time. This can prove useful when you apply different workloads and want to compare them on the same monitoring graphics. The `--id` flag identifies the group of instances that need to synchronize and the `--expected-instances` flag sets the size of the group.

Let's start a couple of instances to compare the impact of message size. The first instance uses 100-byte message:

```
java -jar stream-perf-test.jar --id msg-size-comparison --expected-instances 2 \
                               --size 100
```

The instance will wait until the second one is ready:

```
java -jar stream-perf-test.jar --id msg-size-comparison --expected-instances 2 \
                               --size 200
```

Both instances *must* share the same `--id` if they want to communicate to synchronize.

The default synchronization timeout is 10 minutes. This can be changed with the `--instance-sync-timeout` flag, using a value in seconds.

> **NOTE** Instance synchronization is compatible with PerfTest, the AMQP 0.9.1 performance tool for RabbitMQ: instances of both tools can synchronize with each other. The 2 tools use the same flags for this feature.

Instance synchronization requires IP multicast to be available. IP multicast is not necessary when the performance tool runs on Kubernetes pods. In this case, the tool asks Kubernetes for a list of pod IPs. The performance tool instances are expected to run in the same namespace, and the namespace must be available in the `MY_POD_NAMESPACE` environment variable or provided with the `--instance-sync-namespace` flag. As soon as the namespace information is available, the tool will prefer listing pod IPs over using IP multicast. Here is an example of using instance synchronization on Kubernetes by providing the namespace explicitly:

```
java -jar stream-perf-test.jar --id workload-1 --expected-instances 2 \
                               --instance-sync-namespace qa
```

> **NOTE** The performance tool needs permission to ask Kubernetes for a list of pod IPs. This is done by creating various policies e.g. with YAML. See the Kubernetes discovery protocol for JGroups page for more information.

#### Using Environment Variables as Options

Environment variables can sometimes be easier to work with than command line options. This is especially true when using a manifest file for configuration (with Docker Compose or Kubernetes) and the number of options used grows.

The performance tool automatically uses environment variables that match the snake case version of its long options. E.g. it automatically picks up the value of the `BATCH_SIZE` environment variable for the `--batch-size` option, but only if the environment variable is defined.

You can list the environment variables that the tool picks up with the following command:

```
java -jar stream-perf-test.jar --environment-variables
```

The short version of the option is `-env`.

To avoid collisions with environment variables that already exist, it is possible to specify a prefix for the environment variables that the tool looks up. This prefix is defined with the `RABBITMQ_STREAM_PERF_TEST_ENV_PREFIX` environment variable, e.g.:

```
RABBITMQ_STREAM_PERF_TEST_ENV_PREFIX="STREAM_PERF_TEST_"
```

With `RABBITMQ_STREAM_PERF_TEST_ENV_PREFIX="STREAM_PERF_TEST_"` defined, the tool looks for the `STREAM_PERF_TEST_BATCH_SIZE` environment variable, not `BATCH_SIZE`.

#### Logging

The performance tool binary uses Logback with an internal configuration file. The default log level is `warn` with a console appender.

It is possible to define loggers directly from the command line, this is useful for quick debugging. Use the `rabbitmq.streamperftest.loggers` system property with `name=level` pairs, e.g.:

```
java -Drabbitmq.streamperftest.loggers=com.rabbitmq.stream=debug -jar stream-perf-
test.jar
```

It is possible to define several loggers by separating them with commas, e.g. `-Drabbitmq.streamperftest.loggers=com.rabbitmq.stream=debug,com.rabbitmq.stream.perf=info`.

It is also possible to use an environment variable:

```
export RABBITMQ_STREAM_PERF_TEST_LOGGERS=com.rabbitmq.stream=debug
```

The system property takes precedence over the environment variable.

Use the environment variable with the Docker image:

```
docker run -it --rm --network host \
    --env RABBITMQ_STREAM_PERF_TEST_LOGGERS=com.rabbitmq.stream=debug \
    pivotalrabbitmq/stream-perf-test
```

## Building the Performance Tool

To build the uber JAR:

```
./mvnw clean package -Dmaven.test.skip -P performance-tool
```

Then run the tool:

```
java -jar target/stream-perf-test.jar
```