

RabbitMQ Stream Java Client

The RabbitMQ Stream Java Client is a Java library to communicate with the [RabbitMQ Stream Plugin](#). It allows to create and delete streams, as well as to publish to and consume from these streams.

What is a RabbitMQ Stream?

A RabbitMQ stream is an append-only, FIFO structure. 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 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?

A stream abstraction is useful when one or several consumer applications require the whole history of data ("replay").

Streams can also be useful when a higher throughput than with classical RabbitMQ queues is required. RabbitMQ streams use various optimization techniques and a custom network protocol to achieve performances that are not possible with the other protocols supported in RabbitMQ (AMQP, STOMP, MQTT). This does not make the stream protocol better than these protocols, they just all serve different purposes.

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 queue* with the AMQP 0-9-1. A *stream queue* is a special type of queue that has been adapted to provide the stream semantics (mainly non-destructive reading).

Using a stream 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 a stream queue, 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 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.

The Stream Java Client

The library requires Java 8 or more.

Setting up RabbitMQ

A RabbitMQ node with the stream plugin enabled is required. The easiest way to get up and running is to use Docker. It is also possible to use the generic Unix package.

With Docker

The following command creates a one-time Docker container to run RabbitMQ with the stream plugin enabled:

Running the stream plugin with Docker

```
docker run -it --rm --name rabbitmq -p 5555:5555 pivotalrabbitmq/rabbitmq-stream
```

The previous command exposes only the stream port (5555), you can expose ports for other protocols:

Exposing the AMQP 0.9.1 and management ports:

```
docker run -it --rm --name rabbitmq -p 5555:5555 -p 5672:5672 -p 15672:15672 \
  pivotalrabbitmq/rabbitmq-stream
```

Refer to the official [RabbitMQ Docker image web page](#) to find out more about its usage. Make sure to use the `pivotalrabbitmq/rabbitmq-stream` image in the command line.

The `pivotalrabbitmq/rabbitmq-stream` Docker image is meant for development usage only. It does not support all the features of the official Docker image, like TLS.

With the Generic Unix Package

The generic Unix package requires [Erlang](#) to be installed.

- Download the [latest generic Unix alpha from Bintray](#).
- Follow the [instructions to install the generic Unix package](#).
- Enable the plugin `./rabbitmq-plugins enable rabbitmq_stream`.
- Start the broker `./rabbitmq-server -detached`. This starts the stream listener on port 5555.

Dependencies

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

Note the client uses the [Apache QPid Proton-J](#) library for [AMQP 1.0 message encoding and decoding](#).

Maven

pom.xml

```
<dependencies>

  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>stream-client</artifactId>
    <version>{project-version}</version>
  </dependency>

  <dependency>
    <groupId>org.apache.qpid</groupId>
    <artifactId>proton-j</artifactId>
    <version>{protonj-version}</version>
  </dependency>

</dependencies>

<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>
```

Gradle

```
dependencies {
    compile "com.rabbitmq:stream-client:{project-version}"
    compile "org.apache.qpid:proton-j:{protonj-version}"
}

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 [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().addData(String.valueOf(i).getBytes()).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)
    .messageHandler((offset, message) -> { ②
        sum.addAndGet(Long.parseLong(new String(message.getBodyAsBinary()))); ③
        consumeLatch.countDown(); ④
    })
    .build();

consumeLatch.await(10, TimeUnit.SECONDS); ⑤

System.out.println("Sum: " + sum.get()); ⑥

consumer.close(); ⑦
```

- ① Create the `Consumer` with `Environment#consumerBuilder`
- ② Set up the logic to handle messages
- ③ Add the value in the message body to the sum
- ④ Count down on each message
- ⑤ Wait for all messages to have arrived
- ⑥ Output the sum
- ⑦ 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:5555

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:5555/%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` and TLS is not supported.

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`s` and `Consumer`s`. 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(
        "rabbitmq-stream://host1:5555", ①
        "rabbitmq-stream://host2:5555",
        "rabbitmq-stream://host3:5555")
    )
    .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.

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

Parameter Name	Description	Default
<code>uri</code>	The URI of the node to connect to (single node).	<code>rabbitmq-stream://guest:guest@localhost:5555/%2f</code>
<code>uris</code>	The URI of the nodes to try to connect to (cluster).	<code>rabbitmq-stream://guest:guest@localhost:5555/%2f</code> singleton list
<code>host</code>	Host to connect to.	<code>localhost</code>
<code>port</code>	Port to use.	<code>5555</code>
<code>username</code>	Username to use to connect.	<code>guest</code>
<code>password</code>	Password to use to connect.	<code>guest</code>
<code>virtualHost</code>	Virtual host to connect to.	<code>/</code>

Parameter Name	Description	Default
<code>recoveryBackOffDelayPolicy</code>	Delay policy to use for backoff on connection recovery.	Fixed delay of 5 seconds

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

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 retention policy: once the stream reaches a given size, 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.

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.

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
<code>batchSize</code>	The maximum number of messages to accumulate before sending them to the broker.	100

Parameter Name	Description	Default
<code>subEntrySize</code>	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.	1 (meaning no use of sub-entry batching)
<code>maxUnconfirmedMessages</code>	The maximum number of unconfirmed outbound messages. <code>Producer#send</code> will start blocking when the limit is reached.	10,000
<code>batchPublishingDelay</code>	Period to send a batch of messages.	100 ms

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.

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.

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

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.

NOTE

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.

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.

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.

Client Reference (deprecated)

Connecting to RabbitMQ Stream

The main API of the stream Java client is the `com.rabbitmq.stream.Client` class. To connect to a local instance of the RabbitMQ Stream plugin with the default user and password, just create an `Client` instance

Connecting with all the defaults

```
Client client = new Client(); ①
```

① Connect to localhost:5555 with guest / guest

A `Client` constructor can take a `ClientParameters` to set parameters like host, port, virtual host, username, and password:

Connecting with parameters

```
Client client = new Client(new Client.ClientParameters() ①
    .host("my-rabbitmq-stream")
    .port(1234)
    .virtualHost("my-virtual-host")
    .username("stream-user")
    .password("stream-password")
);
```

① Use `ClientParameters` to set connection parameters

Managing Streams

Streams can be created with the `Client#create(String)` method:

Creating a stream

```
Client.Response response = client.create("my-stream"); ①
if (!response.isOk()) { ②
    response.getResponseCode(); ③
}
```

① Create the **my-stream** stream

② Check if creation is OK

③ Check response code

The **Constants** class can help to make sense of response codes.

The deletion of a stream follows the same pattern with the **Client#delete(String)** method:

Deleting a stream

```
Client.Response response = client.delete("my-stream"); ①
if (!response.isOk()) { ②
    response.getResponseCode(); ③
}
```

① Delete the **my-stream** stream

② Check if deletion is OK

③ Check response code

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). Therefore you should consider streams as long-lived objects.

Limiting the Size of a Stream

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 retention policy: once the stream reaches a given size, 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

```
client.create("my-stream", new Client.StreamParametersBuilder()
    .maxLengthGb(10) ①
    .maxSegmentSizeMb(500) ②
    .build()
);
```

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.

You can also use the `ByteCapacity` class to set up the retention policy, it is just a matter of API preference:

Using `ByteCapacity` to set up the retention policy

```
client.create("my-stream", new Client.StreamParametersBuilder()
    .maxLengthBytes(ByteCapacity.GB(10)) ①
    .maxSegmentSizeBytes(ByteCapacity.MB(500)) ②
    .build()
);
```

① Set the maximum size to 10 GB

② Set the segment size to 500 MB

Publishing Messages

The simplest way to publish something to a stream is the `Client#publish(Stream, byte[])` method:

Publishing a message

```
byte[] messagePayload = "hello".getBytes(StandardCharsets.UTF_8); ①
List<Long> publishingIds = client.publish(
    "my-stream", ②

    Collections.singletonList(client.messageBuilder().addData(messagePayload).build()) ③
);
```

① The payload of a message is an array of bytes

② The stream to publish to

③ The content to publish

The `publish` method returns a long: it is the publishing identifier for the message. This identifier is

used to correlate the message with a publishing confirmation that the broker sends asynchronously. This allows making sure a message made it to the broker and provides at-least-once guarantees.

NOTE

The publishing identifier is local to the client instance connection. It is neither a global identifier for the message, nor the offset of the message in the stream.

Each `Client` instance can register a callback to react to publish confirms when the instance is created:

Handling publish confirms

```
Client client = new Client(new Client.ClientParameters()
    .publishConfirmListener(publishingId -> { ❶
        // map publishing ID with initial outbound message
    })
);
```

❶ Set publish confirm callback on client creation

The broker also sends notification for messages that it could not take into account, for example when the targeted stream does not exist or is not available. Just like with publish confirms, set up the listener when creating the client instance:

Handling publishing errors

```
Client client = new Client(new Client.ClientParameters()
    .publishErrorListener((publishingId, errorCode) -> { ❶
        // map publishing ID with initial outbound message
    })
);
```

❶ Set publish error callback on client creation

The `PublishErrorListener` callback provides the publishing ID as well as an error code to figure out the reason of the publishing error.

It is also possible to publish messages in batch, with the `publishBinary(List<byte[]>)` method. The method returns a list of publishing identifiers to handle publish confirms. The following snippet shows how to use publish messages in batch:

```
List<Message> messages = IntStream.range(0, 9)
    .mapToObj(i -> ("hello" + i).getBytes(StandardCharsets.UTF_8))
    .map(body -> client.messageBuilder().addData(body).build())
    .collect(Collectors.toList()); ❶

List<Long> publishingIds = client.publish(
    "my-stream", ❷
    messages ❸
);
```

❶ Messages are a list of byte arrays

❷ The stream to publish to

❸ The messages to publish

TIP Use batch publishing to improve throughput.

You may wonder if messages are only made of a byte array payload. They are not. They can have fixed and arbitrary headers. We will see how to create complex messages in a dedicated section.

Consuming Messages

The `Client#subscribe` method allows subscribing to a stream:

Subscribing to a stream

```
Client.Response response = client.subscribe(
    1, ❶
    "my-stream", ❷
    OffsetSpecification.first(), ❸
    10 ❹
);
if (!response.isOk()) { ❺
    response.getResponseCode(); ❻
}
```

❶ The identifier of the subscription

❷ The stream to consume from

❸ The offset to start consuming from

❹ The initial number of credits

❺ Check if response is OK

❻ Check response code

Once the subscription is done, the broker will start sending messages. We will see how to handle these messages asynchronously in a callback in a moment. Let's have a further look at the parameters of `subscribe`:

- Identifier of the subscription: this is an arbitrary number to correlate the subscription with the inbound messages. The value must be unique within the connection.
- Stream: this is the stream to consume from. It obviously must exist.
- Offset: this is the offset to start consuming from. There are several ways to specify the offset:
 - `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.
- Initial number of credits: the number of chunks of messages the broker can send immediately after the subscription. This is a way to throttle delivery in case the broker send messages faster than the client can process them. We will see more about credits later.

The broker delivers messages in chunks (batches). The size of a chunk is not fixed, it can vary from a few messages to thousands of them in the same chunk. Delivering messages in chunks instead of one by one contributes to the higher throughput RabbitMQ Stream can achieve compared to other protocols like AMQP 0-9-1 (which do use batch at all).

The `Client` provides 2 callbacks to handle inbound messages: one for each chunk and another one for each message within a chunk. Here is more information about each callback:

- A `ChunkListener`: this callback is merely technical, it gives the application the occasion to ask for more chunks of messages or to compute metrics about chunks (number of messages, size in bytes).
- A `MessageListener`: this callback should contain the application code to process a message.

The 2 callbacks are provided when creating the `Client` instance:

```
Client consumer = new Client(new Client.ClientParameters()  
    .chunkListener((client, subscriptionId, offset, messageCount, dataSize) -> {  
        ①  
        client.credit(subscriptionId, 1); ②  
    })  
    .messageListener((subscriptionId, offset, message) -> { ③  
        message.getBodyAsBinary(); ④  
    }));
```

- ① Set chunk listener
- ② Ask for a new chunk after each chunk delivery
- ③ Set message listener
- ④ Get and process byte array payload

Asking for a new credit on each chunk is usually a good default, so the `ChunkListener` implementation above can be taken as-is for most applications.

What's the deal with credits?

The broker cannot keep sending messages to a client if the latter cannot keep up. There must be a way for the client and the server to agree on the delivery rate, not too high, but also not too low (to keep the client busy). Credits are way to achieve this.

When subscribing, the client specifies an initial number of credits. For every chunk the broker delivers, it removes a credit from the initial count. Once the credit count reaches 0 for the subscription, the broker stops sending chunks of messages. The client needs to ask for more credits to see more chunks of messages getting sent. This way a client can make sure it is not overwhelmed by the flow of incoming message.

NOTE

But the client must be careful in the way it asks for credits. It can choose to ask for more chunks when its credit count reaches 0, but this is usually not a good strategy: during the time the credit request arrives to the broker and a new chunk is received, the client remains idle, wasting processing time. This is why it is usually preferable to ask for a reasonable amount of initial credits (a few, not dozens, remember a chunk can contain thousands of messages) and ask for a new credit on every chunk.

Inappropriate credit values can be the cause of an unstable consuming rate. The recommendation given above is usually a good rule of thumb, but may not be appropriate for all workloads.

Working with Complex Messages

The publishing and consuming examples showed that messages are made of a byte array payload, but they 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 `Client#messageBuilder` method. To publish a `Message`, use the `Client#publish(String,Message)`:

Creating a `Message` instance

```
Message message = client.messageBuilder() ①
    .properties() ②
        .messageId(UUID.randomUUID())
        .correlationId(UUID.randomUUID())
        .contentType("text/plain")
    .messageBuilder() ③
        .addData("hello".getBytes(StandardCharsets.UTF_8)) ④
    .build(); ⑤
List<Long> publishingId = client.publish("my-stream",
Collections.singletonList(message)); ⑥
```

- ① Get the message builder from the client
- 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.

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
```

The Performance Tool

The library contains also a performance tool to test the RabbitMQ Stream plugin. It is [downloadable from Bintray](#) as an uber JAR and can be built separately as well.

Using the Performance Tool

To launch a run:

```
$ java -jar stream-perf-test-{version}.jar
10:11:54.324 [main] INFO  c.r.stream.perf.StreamPerfTest - Created stream stream1
10:11:54.385 [main] INFO  c.r.stream.perf.StreamPerfTest - Producer will stream
stream1
10:11:54.387 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting consuming on
stream1
10:11:54.390 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting producer
1, published 155230 msg/s, confirmed 147824 msg/s, consumed 124487 msg/s, latency
min/median/75th/95th/99th 1121/8225/17647/62468/73991 µs, chunk size 109
2, published 359193 msg/s, confirmed 336535 msg/s, consumed 306748 msg/s, latency
min/median/75th/95th/99th 1398/56590/80607/127818/135925 µs, chunk size 345
3, published 523429 msg/s, confirmed 509044 msg/s, consumed 478710 msg/s, latency
min/median/75th/95th/99th 1478/29996/69536/111946/135079 µs, chunk size 529
4, published 599735 msg/s, confirmed 594707 msg/s, consumed 568315 msg/s, latency
min/median/75th/95th/99th 964/21032/52977/98643/133399 µs, chunk size 548
5, published 632114 msg/s, confirmed 609804 msg/s, consumed 591426 msg/s, latency
min/median/75th/95th/99th 964/34303/74318/110684/127440 µs, chunk size 588
6, published 619328 msg/s, confirmed 618229 msg/s, consumed 598410 msg/s, latency
min/median/75th/95th/99th 964/45918/86391/114714/138207 µs, chunk size 657
^C
Summary: published 641792 msg/s, confirmed 635240 msg/s, consumed 636256 msg/s,
latency 95th 112730 µs, chunk size 711
```

The previous command will start publishing to and consuming from a stream created only for the

test. 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-HH:mm:ss.txt` when the run ends.

To see the options:

```
java -jar stream-perf-test-{version}.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.

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
```