RabbitMQ Stream .Net Client

Table of Contents

What is a RabbitMQ Stream? . . . 2
When to Use RabbitMQ Stream? . . . 2
Other Way to Use Streams in RabbitMQ . . . 2
Guarantees . . . 3
Stream Client Overview . . . 3
Stability of Programming Interfaces . . . 3
The Stream .NET Client . . . 4
- Setting up RabbitMQ . . . 4
  - With Docker . . . 4
    - With Docker Bridge Network Driver . . . 4
    - With Docker Host Network Driver . . . 5
  - With a RabbitMQ Package Running on the Host . . . 5
- Dependencies . . . 5
- Sample Application . . . 5
- RabbitMQ Stream .NET API . . . 10
  - Overview . . . 10
  - StreamSystem . . . 10
    - Creating the StreamSystem . . . 10
    - Understanding Connection Logic . . . 11
    - Enabling TLS . . . 11
    - Configuring the Stream System . . . 12
    - When a Load Balancer is in Use . . . 13
    - Managing Streams . . . 14
  - Producer . . . 16
    - Creating a Producer . . . 16
    - Sending Messages . . . 17
    - Working with Complex Messages . . . 20
    - Message Deduplication . . . 21
      - Use DeduplicationProducer . . . 22
      - Understanding Publishing ID . . . 23
      - Restarting a Producer Where It Left Off . . . 23
    - Sub-Entry Batching and Compression . . . 24
  - Consumer . . . 27
    - Creating a Consumer . . . 27
    - Specifying an Offset . . . 28
    - Tracking the Offset for a Consumer . . . 29

Manual Offset Tracking . . . 30
Considerations On Offset Tracking . . . 31
Single Active Consumer . . . 31
Enabling Single Active Consumer . . . 33
Offset Tracking . . . 33
Reacting to Consumer State Change . . . 33
Low Level and High Level classes . . . 34

The RabbitMQ Stream .Net Client is a .Net 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 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.

See also stream-core stream-plugin comparison

Guarantees

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

Message deduplication is also supported on the publisher side.

Stream Client Overview

The RabbitMQ Stream .NET 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 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.
- *let the client handle network failure* thanks to automatic connection recovery and automatic re-subscription for consumers.

Stability of Programming Interfaces

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 `RabbitMQ.Stream.Client.Reliable` package (e.g. `Producer`, `Consumer`, `Message`). These API constitute the main programming model of the client and will be kept as stable as possible.

The Stream .NET Client

The library requires .NET 6 or .NET 7.

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* 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:3.11
```

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:3.11-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:3.11
```

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 3.9 or more.
- 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

The client is distributed via NuGet.

Sample Application

This section covers the basics of the RabbitMQ Stream .NET API by building a small publish/consume application. This is a good way to get an overview of the API.

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

```
using System.Net;
using Microsoft.Extensions.Logging; ①
using RabbitMQ.Stream.Client; ②
using RabbitMQ.Stream.Client.Reliable; ③
```

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

Creating the environment

```
var streamSystem = await StreamSystem.Create( ①
    new StreamSystemConfig() ②
    {
        UserName = "guest",
        Password = "guest",
        Endpoints = new List<EndPoint>() {new IPEndPoint(IPAddress.Loopback, 5552)}
    },
    streamLogger ③
).ConfigureAwait(false);

// Create a stream

const string StreamName = "my-stream";
await streamSystem.CreateStream(
    new StreamSpec(StreamName) ④
    {
        MaxSegmentSizeBytes = 20_000_000 ⑤
    }).ConfigureAwait(false);
```

① Use `StreamSystem.Create(..)` to create the environment

② Define the connection configuration

③ Add the logger. (Not mandatory it is very useful to understand what is going on)

④ Create the stream

⑤ Define the retention policy

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.

Publishing messages

```
var confirmationTaskCompletionSource = new TaskCompletionSource<int>();
var confirmationCount = 0;
```

```
const int MessageCount = 100;
var producer = await Producer.Create( ①
        new ProducerConfig(streamSystem, StreamName)
        {
            ConfirmationHandler = async confirmation => ②
            {
                Interlocked.Increment(ref confirmationCount);

                // here you can handle the confirmation
                switch (confirmation.Status)
                {
                    case ConfirmationStatus.Confirmed: ③
                        // all the messages received here are confirmed
                        if (confirmationCount == MessageCount)
                        {
                            Console.WriteLine("*********************************");
                            Console.WriteLine($"All the {MessageCount} messages are
confirmed");
                            Console.WriteLine("*********************************");
                        }

                        break;

                    case ConfirmationStatus.StreamNotAvailable:
                    case ConfirmationStatus.InternalError:
                    case ConfirmationStatus.AccessRefused:
                    case ConfirmationStatus.PreconditionFailed:
                    case ConfirmationStatus.PublisherDoesNotExist:
                    case ConfirmationStatus.UndefinedError:
                    case ConfirmationStatus.ClientTimeoutError:
④
                        Console.WriteLine(
                            $"Message {confirmation.PublishingId} failed with
{confirmation.Status}");
                        break;
                    default:
                        throw new ArgumentOutOfRangeException();
                }

                if (confirmationCount == MessageCount)
                {
                    confirmationTaskCompletionSource.SetResult(MessageCount);
                }

                await Task.CompletedTask.ConfigureAwait(false);
            }
        },
        producerLogger ⑤
    )
    .ConfigureAwait(false);
```

```
// Send 100 messages
Console.WriteLine("Starting publishing...");
for (var i = 0; i < MessageCount; i++)
{
    await producer.Send( ⑥
        new Message(Encoding.ASCII.GetBytes($"{i}"))
    ).ConfigureAwait(false);
}


confirmationTaskCompletionSource.Task.Wait(); ⑦
await producer.Close().ConfigureAwait(false); ⑧
```

① Create the `Producer` with `Producer.Create`

② Define the `ConfirmationHandler` where the messages are confirmed or not

③ Message is confirmed from the server

④ Message not confirmed

⑤ Add the logger. (Not mandatory it is very useful to understand what is going on)

⑥ Send messages with `producer.Send(Message)`

⑦ Wait for messages confirmation

⑧ Close the producer

It is now time to consume the messages. The `Consumer.Create` 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

```
Console.WriteLine("Starting consuming...");
var consumer = await Consumer.Create( ①
        new ConsumerConfig(streamSystem, StreamName)
        {
            OffsetSpec = new OffsetTypeFirst(), ②
            MessageHandler = async (sourceStream, consumer, messageContext, message)
=> ③
            {
                if (Interlocked.Increment(ref consumerCount) == MessageCount)
                {
                    Console.WriteLine("**********************************");
                    Console.WriteLine($"All the {MessageCount} messages are received"
);
                    Console.WriteLine("**********************************");
                    consumerTaskCompletionSource.SetResult(MessageCount);
                }
                await Task.CompletedTask.ConfigureAwait(false);
            }
        },
```

```
        consumerLogger ④
    )
    .ConfigureAwait(false);
consumerTaskCompletionSource.Task.Wait(); ⑤
await consumer.Close().ConfigureAwait(false); ⑥
```

① Create the `Consumer` with `Consumer.Create`

② Start consuming from the beginning of the stream

③ Set up the logic to handle message

④ Add the logger. (Not mandatory it is very useful to understand what is going on)

⑤ Wait for all the messages are consumed

⑥ Close the consumer

Cleaning before terminating

```
await streamSystem.DeleteStream(StreamName).ConfigureAwait(false); ①
await streamSystem.Close().ConfigureAwait(false); ②
```

① Delete the stream

② Close the stream system

About logging

```
var factory = LoggerFactory.Create(builder =>
{
    builder.AddSimpleConsole();
    builder.AddFilter("RabbitMQ.Stream", LogLevel.Information);
});

// Define the logger for the StreamSystem and the Producer/Consumer
var producerLogger = factory.CreateLogger<Producer>(); ①
var consumerLogger = factory.CreateLogger<Consumer>(); ②
var streamLogger = factory.CreateLogger<StreamSystem>(); ③
```

① Define the logger for the producer

② Define the logger for the consumer

③ Define the logger for the stream system

The logger is not mandatory but it is highly recommended to configure it to understand what is happening. In this example, we are using `Microsoft.Extensions.Logging.Console` to log to the console, but you can use any logger you want. `Microsoft.Extensions.Logging.Console` is not shipped with the client.

Run the sample application

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

```
$  dotnet run --gs
Starting publishing...
********************************
All the 100 messages are confirmed
********************************
Starting consuming...
********************************
All the 100 messages are received
********************************
```

RabbitMQ Stream .NET API

Overview

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

- `RabbitMQ.Stream.Client` for connecting to a node and optionally managing streams.
- `RabbitMQ.Stream.Client.Reliable.Producer` to publish messages.
- `RabbitMQ.Stream.Client.Reliable.Consumer` to consume messages.

StreamSystem

Creating the StreamSystem

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

Creating an environment with all the defaults

```
private static async Task CreateSimple()
{
    var streamSystem = await StreamSystem.Create( ①
        new StreamSystemConfig()
    ).ConfigureAwait(false);


    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Create an environment that will connect to localhost:5552

② Close the environment after usage

Note the streamSystem 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

`StreamSystem` instance when it starts up and close it when it exits.

It is possible to use a multiple end-points to connect to a cluster of nodes. The:

Creating an streamSystem with multiple end-points

```
private static async Task CreateMultiEndPoints()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
        {
            UserName = "guest",
            Password = "guest",
            Endpoints = new List<EndPoint> ①
            {
                new IPEndPoint(IPAddress.Parse("192.168.5.12"), 5552),
                new IPEndPoint(IPAddress.Parse("192.168.5.18"), 5552),
            }
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Define the end-points to connect to

By specifying several endpoints, the system will try to connect to the first one, and will pick a new endpoint randomly in case of disconnection.

Understanding Connection Logic

Creating the StreamSystem 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

The default TLS port is 5551.

Creating an StreamSystem that uses TLS

```
private static async Task CreateTls()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
        {
            UserName = "guest",
            Password = "guest",
```

```
            Ssl = new SslOption() ①
            {
                Enabled = true,
                ServerName = "rabbitmq-stream",
                CertPath = "/path/to/cert.pem", ②
                CertPassphrase = "Password",
            }
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Enable TLS

② Load certificate authority (CA) certificate from PEM file

Creating a TLS environment that trusts all server certificates for development

```
private static async Task CreateTlsTrust()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
        {
            UserName = "guest",
            Password = "guest",
            Ssl = new SslOption() ①
            {
                Enabled = true,
                AcceptablePolicyErrors = SslPolicyErrors.RemoteCertificateNotAvailable
| ①
                                         SslPolicyErrors.RemoteCertificateChainErrors
|
                                         SslPolicyErrors.RemoteCertificateNameMismatch
            }
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Trust all server certificates

Configuring the Stream System

The following table sums up the main settings to create an `StreamSystem` using the `StreamSystemConfig`:

Parameter Name	Description	Default
`UserName`	User name to use to connect.	`guest`
`Password`	Password to use to connect.	`guest`

Parameter Name	Description	Default
`VirtualHost`	Virtual host to connect to.	`/`
`ClientProvidedName`	To identify the client in the management UI.	`dotnet-stream-locator`
`Heartbeat`	Time between heartbeats.	`1 minute`
`Endpoints`	The list of endpoints to connect to.	`localhost:5552`
`addressResolver`	Contract to change resolved node address to connect to.	Pass-through (no-op)
`Ssl`	Configuration helper for TLS.	`null`

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 `StreamSystemConfig#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

```
private static async Task CreateAddressResolver()
{
    var addressResolver = new AddressResolver(new IPEndPoint(IPAddress.Parse(
"xxx.xxx.xxx"), 5552)); ①

    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
        {
            UserName = "myuser",
            Password = "mypassword",
            AddressResolver = addressResolver, ②
            Endpoints = new List<EndPoint> {addressResolver.EndPoint} ③
        }
    ).ConfigureAwait(false);


    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Set the load balancer address

② Use load balancer address for initial connection

③ Set the endpoints based on `AddressResolver`

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 `StreamSystem`. This comes in handy for development and testing purposes.

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

Creating a stream

```
private static async Task CreateStream()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
    ).ConfigureAwait(false);

    await streamSystem.CreateStream( ①
        new StreamSpec("my-stream")
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false); ②
}
```

① Create the `my-stream` stream

`StreamSystem.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 `StreamSystem.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 `StreamSystem#Delete(String)` method:

Deleting a stream

```
private static async Task DeleteStream()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
    ).ConfigureAwait(false);

    await streamSystem.DeleteStream("my-stream").ConfigureAwait(false); ①
    await streamSystem.Close().ConfigureAwait(false); ②
}
```

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

```
private static async Task CreateStreamRetentionLen()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
    ).ConfigureAwait(false);

    await streamSystem.CreateStream(
        new StreamSpec("my-stream")
        {
            MaxLengthBytes = 10_737_418_240, ①
            MaxSegmentSizeBytes = 524_288_000 ②
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false);
}
```

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

```
private static async Task CreateStreamRetentionAge()
{
    var streamSystem = await StreamSystem.Create(
        new StreamSystemConfig()
    ).ConfigureAwait(false);

    await streamSystem.CreateStream(
        new StreamSpec("my-stream")
        {
            MaxAge = TimeSpan.FromHours(6),  ①
            MaxSegmentSizeBytes = 524_288_000 ②
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false);
}
```

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 with `Producer.Create`. The only mandatory setting to specify is the stream to publish to:

Creating a producer from the environment

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var producer = await Producer.Create( ①
    new ProducerConfig(
        streamSystem,
        "my-stream") ②
).ConfigureAwait(false);

await producer.Close().ConfigureAwait(false); ③
await streamSystem.Close().ConfigureAwait(false);
```

① Use `Producer.Create` to define the producer

② Specify the stream to publish to

③ 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 when `Reference` is not set. Starting from version 1.2.0 the `Reference` field is deprecated. `Reference` is needed for deduplication see the Deduplication section for more details.

Internally, the `StreamSystem` 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 `ProducerConfig`:

Parameter Name	Description	Default
`StreamSystem`	The StreamSystem to use to create the producer.	No default, mandatory setting.
`stream`	The stream to publish to.	No default, mandatory setting.
`Reference`	The logical name of the producer. Specify a name to enable message deduplication.	`null` (no deduplication) - Deprecated in version 1.2.0
`ConfirmationHandler`	The confirmation handler where received the messages confirmations.	`null` (no confirmation handler)
ClientProvidedName	The TCP connection name to identify the client.	`dotnet-stream-producer`
`MaxInFlight`	The maximum number of messages that can be in flight at any given time. Messages sent - Messages confirmed. To avoid to flood the broker with messages.	1000
`ReconnectStrategy`	The strategy to use when the connection to the broker is lost.	`BackOffReconnectStrategy`
`MessagesBufferSize`	Number of the messages sent for each frame-send. This value is valid only for the `Send(Message)` method.	100
`TimeoutMessageAfter`	Time to wait before considering a message as not confirmed.	3 seconds
`SuperStreamConfig`	The super stream configuration.	`null` (no super stream)

Sending Messages

Once a `Producer` has been created, it is possible to send a message with: - `Producer#send(Message)`, - `Producer#send(List<Message>)` - `Producer#send(List<Message> messages, CompressionType compressionType)`. snippet shows how to publish a message with a byte array payload:

Sending a message

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var producer = await Producer.Create(
    new ProducerConfig(
        streamSystem,
        "my-stream")
    {
        ConfirmationHandler = async confirmation => ⑤
        {
            switch (confirmation.Status)
            {
                case ConfirmationStatus.Confirmed:
                    Console.WriteLine("Message confirmed");
                    break;
                case ConfirmationStatus.ClientTimeoutError:
                case ConfirmationStatus.StreamNotAvailable:
                case ConfirmationStatus.InternalError:
                case ConfirmationStatus.AccessRefused:
                case ConfirmationStatus.PreconditionFailed:
                case ConfirmationStatus.PublisherDoesNotExist:
                case ConfirmationStatus.UndefinedError:
                    Console.WriteLine("Message not confirmed with error: {0}",
confirmation.Status);
                    break;

                default:
                    throw new ArgumentOutOfRangeException();
            }


            await Task.CompletedTask.ConfigureAwait(false);
        }
    }
).ConfigureAwait(false);

var message = new Message(Encoding.UTF8.GetBytes("hello")); ①
await producer.Send(message).ConfigureAwait(false); ②
var list = new List<Message> {message};
await producer.Send(list).ConfigureAwait(false); ③
await producer.Send(list, CompressionType.Gzip).ConfigureAwait(false); ④

await producer.Close().ConfigureAwait(false);
await streamSystem.Close().ConfigureAwait(false);
```

① The payload of a message is an array of bytes. 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.

② Send the message. The method is asynchronous, internally the messages are buffered and sent in batch. Most of the time you can use this method.

③ Batch send is synchronous, there is not additional buffering. The messages are sent immediately. This method is useful when you want to control the number of the messages to sent is a single frame. Can be useful in case you need low latency.

④ Sub entry batching see Sub Entry Batching for more details.

⑤ 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.

`MessagesConfirmation` contains the following information:

Parameter Name	Description
`Stream`	The stream the message was published to.
`PublishingId`	The publishing id of the message.
`Status`	The confirmation status of the message. See Confirmation Status for more details.
`Messages`	The list of messages that have been confirmed or not.

`confirmation.Status` values:

Parameter Name	Description	Source
`ConfirmationStatus.Confirmed`	The message has been confirmed by the broker.	Server
`ConfirmationStatus.Timeout`	Client gave up waiting for the message	Client
StreamNotAvailable	The stream is not available.	Server
InternalError	The broker encountered an internal error.	Server
AccessRefused	Provided credentials are invalid or you lack permissions for specific vhost/etc.	Server
PreconditionFailed	Catch-all for validation on server (eg. requested to create stream with different parameters but same name).	Server
PublisherDoesNotExist	The publisher does not exist.	Server
UndefinedError	Catch-all for any new status that is not yet handled in the library.	Server

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 `StreamSystem`, `Producer`, `Consumer` instances sluggish or even block them. Any long processing should be done in a separate thread (e.g. with an asynchronous `Task.Run(⋯.)`).

NOTE

Mixing different send methods

You can mix different send methods. For example you can send a message with `send(Message)` and then send a batch of messages with `send(List<Message>)`. Avoid to sent the `Refence` property in the `ProducerConfig` it enables the deduplication and you could have unexpected results.

`Reference` is deprecated in the version `1.2.0` see deduplication section for more details.

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 NET client uses the `Message` class to represent a message.

Creating a message with properties

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var producer = await Producer.Create(
    new ProducerConfig(
        streamSystem,
        "my-stream") { }
).ConfigureAwait(false);

var message = new Message(Encoding.UTF8.GetBytes("hello")) ①
{
    ApplicationProperties = new ApplicationProperties() ②
    {
        {"key1", "value1"}, {"key2", "value2"}
```

```
    },
    Properties = new Properties() ③
    {
        MessageId = "message-id",
        CorrelationId = "correlation-id",
        ContentType = "application/json",
        ContentEncoding = "utf-8",
    }
};

await producer.Send(message).ConfigureAwait(false);
await streamSystem.Close().ConfigureAwait(false);
```

① Get the message

② Set the Application properties

③ Set the message Properties. You usually don't need to set the properties.

The `Message` contains also the following read-only properties:

- `MessageHeader`
- `Annotations`
- `AmqpValue`

These values are only for compatibility with the AMQP 1.0 message format.

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.

The client provides a specific class to handle deduplication: `DeduplicationProducer`.

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.

Use DeduplicationProducer

The `DeduplicationProducer` requires the `Reference` as mandatory parameter. This parameter enables deduplication:

Naming a producer to enable message deduplication

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var deduplicatingProducer = await DeduplicatingProducer.Create( ①
    new DeduplicatingProducerConfig(
        streamSystem,
        "my-stream", "my_producer_reference") { }
).ConfigureAwait(false);

var message = new Message(Encoding.UTF8.GetBytes("hello")); ②
await deduplicatingProducer.Send(1, message).ConfigureAwait(false); ③
await deduplicatingProducer.Send(2, message).ConfigureAwait(false);
await deduplicatingProducer.Send(3, message).ConfigureAwait(false);

// deduplication is enabled, so this message will be skipped
await deduplicatingProducer.Send(1, message).ConfigureAwait(false); ④


await streamSystem.Close().ConfigureAwait(false);
```

① Define the `DeduplicatingProducer` class with `Reference` property.

② Get a message

③ Send three messages specifying the `publishingid` and the `Message`.

④ Send again the same message with the same `publishingid` and the `Message` will be skipped by the broker since the `publishingid` is already present with the `Reference` "my_producer_reference" .

Thanks to the name, the broker will be able to track the messages it has persisted on a given stream for this producer.

Consider the `Reference` 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 `Reference` is only one part of the deduplication mechanism, the other part is the *message publishing ID*. The publishing ID is a strictly increasing sequence, starting at 0 and incremented for each message.

- 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`).

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 `DeduplicatingProducer#GetLastPublishedId()` 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 `DeduplicatingProducer#GetLastPublishedId()`:

Setting a producer where it left off

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var deduplicatingProducer = await DeduplicatingProducer.Create( ①
```

```
    new DeduplicatingProducerConfig(
        streamSystem,
        "my-stream", "my_producer_reference") { }
).ConfigureAwait(false);

var lastid = await deduplicatingProducer.GetLastPublishedId().ConfigureAwait(false);
②
var message = new Message(Encoding.UTF8.GetBytes("hello"));

await deduplicatingProducer.Send(lastid + 1, message).ConfigureAwait(false); ③
await deduplicatingProducer.Send(lastid + 2, message).ConfigureAwait(false);
await deduplicatingProducer.Send(lastid + 3, message).ConfigureAwait(false);
await streamSystem.Close().ConfigureAwait(false);
```

① Get a `DeduplicatingProducer` instance

② Query last publishing ID for this producer

③ Use the lastid and increment it

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.

The following snippet shows how to enable sub-entry batching:

Enabling sub-entry batching

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var producer = await Producer.Create( ①
    new ProducerConfig(
        streamSystem,
        "my-stream") ②
).ConfigureAwait(false);

var message = new Message(Encoding.UTF8.GetBytes("hello"));
var list = new List<Message> {message, message, message}; ①
await producer.Send(list, CompressionType.Gzip).ConfigureAwait(false); ②

await producer.Close().ConfigureAwait(false);
```

```
await streamSystem.Close().ConfigureAwait(false);
```

① Define a list of messages to compress

② Send the list of messages to the broker in tis case 3 messages compressed with GZIP

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.

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

Algorithm	Overview	Implementation used
CompressionType.None	No compression.	None
CompressionType.Gzip	Has a high compression ratio but is slow compared to other algorithms.	.Net Implementation
Snappy	Aims for reasonable compression ratio and very high speeds.	Not shipped with the client library, but can be added with the `ICompressionCodec` interface.
LZ4	Aims for good trade-off between speed and compression ratio.	Not shipped with the client library, but can be added with the `ICompressionCodec` interface.
zstd (Zstandard)	Aims for high compression ratio and high speed, especially for decompression.	Not shipped with the client library, but can be added with the `ICompressionCodec` interface.

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

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 Net client.

You can add a compression algorithm to the client library by implementing the `ICompressionCodec` interface and registering it with the `StreamCompressionCodecs` class.

The following snippet shows how to add a compression algorithm to the client library:

Adding a compression algorithm

```
class StreamLz4Codec : ICompressionCodec ①
```

```
{



    private ReadOnlySequence<byte> _compressedReadOnlySequence;
    public void Compress(List<Message> messages)
    {
        MessagesCount = messages.Count;
        UnCompressedSize = messages.Sum(msg => 4 + msg.Size);
        var messagesSource = new Span<byte>(new byte[UnCompressedSize]);
        var offset = 0;
        foreach (var msg in messages)
        {
            offset += WriteUInt32(messagesSource.Slice(offset), (uint)msg.Size);
            offset += msg.Write(messagesSource.Slice(offset));
        }

        using var source = new MemoryStream(messagesSource.ToArray());
        using var destination = new MemoryStream();
        var settings = new LZ4EncoderSettings {ChainBlocks = false};
        using (var target = LZ4Stream.Encode(destination, settings, false))
        {
            source.CopyTo(target);
        }

        _compressedReadOnlySequence = new ReadOnlySequence<byte>(destination.ToArray(
));
    }

    public ReadOnlySequence<byte> UnCompress(ReadOnlySequence<byte> source, uint
dataLen, uint unCompressedDataSize)
    {
        using var target = new MemoryStream();
        using (var sourceDecode = LZ4Stream.Decode(new MemoryStream(source.ToArray())
))
        {
            sourceDecode.CopyTo(target);
        }
        return new ReadOnlySequence<byte>(target.ToArray());
    }
```

① Implement the `ICompressionCodec` interface with all the required methods

The following snippet shows how to register the compression algorithm with the `StreamCompressionCodecs` class:

Registering a compression algorithm

```
StreamCompressionCodecs.RegisterCodec<StreamLz4Codec>(CompressionType.Lz4); ①

var producer = await Producer.Create(
```

```
    new ProducerConfig(
        streamSystem,
        "my-stream") { }
).ConfigureAwait(false);

var message = new Message(Encoding.UTF8.GetBytes("hello"));
var list = new List<Message> {message, message, message};
await producer.Send(list, CompressionType.Lz4).ConfigureAwait(false); ②
```

① Register the compression algorithm with the `StreamCompressionCodecs` class

② Use the compression algorithm in the `producer.Send(list, CompressionType.Lz4)`

Consumer

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

Creating a Consumer

A `Consumer` instance is created with `Consumer.Create(..)`. 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

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var consumer = await Consumer.Create( ①
    new ConsumerConfig( ②
        streamSystem,
        "my-stream")
    {
        // Reference =
        OffsetSpec = new OffsetTypeTimestamp(), ③
        MessageHandler = async (stream, consumer, context, message) => ④
        {
            Console.WriteLine($"Received message:
{Encoding.UTF8.GetString(message.Data.Contents)}");
            await Task.CompletedTask.ConfigureAwait(false);
        }
    }
).ConfigureAwait(false);

await consumer.Close().ConfigureAwait(false); ⑤
await streamSystem.Close().ConfigureAwait(false);
```

① Use `Consumer.Create()` to define the consumer

② Specify `ConsumerConfig` to configure the consumer behavior with the `streamSystem` and `streamName`

to consume from

③ Specify where to start consuming from

④ Handle the messages

⑤ Close consumer after usage

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

WARNING

Keep the message processing callback as short as possible

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

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

Parameter Name	Description	Default
`StreamSystem`	The StreamSystem to use.	No default, mandatory setting.
`Stream`	The stream to consume from.	No default, mandatory setting.
`OffsetSpec`	The offset to start consuming from.	`OffsetTypeNext()`
`MessageHandler`	The callback for inbound messages.	No default.
`Reference`	The consumer name (for offset tracking.)	`null` (no offset tracking)
`ReconnectStrategy`	The strategy to use when the connection to the broker is lost.	`BackOffReconnectStrategy`
`ClientProvidedName`	To identify the client in the management UI	`dotnet-stream-conusmer`
`IsSingleActiveConsumer`	Enable the Single Active Consumer feature	`false`
`IsSuperStream`	Enable the Super Stream feature	`false`

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.* 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:

- `OffsetTypeFirst()`: starting from the first available offset. If the stream has not been truncated, this means the beginning of the stream (offset 0).
- `OffsetTypeLast()`: starting from the end of the stream and returning the last chunk of messages immediately (if the stream is not empty).
- `OffsetTypeNext()`: starting from the next offset to be written. Contrary to `OffsetTypeLat()`, consuming with `OffsetTypeNext()` 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.
- `OffsetTypeOffset(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.
- `OffsetTypeTimestamp(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.

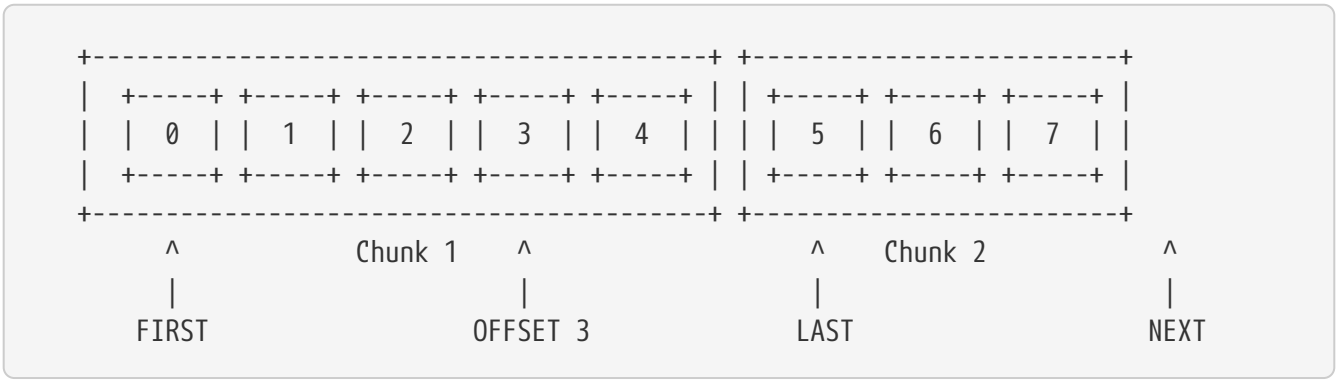
NOTE

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:

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 `ConsumerConfig#Reference`. 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` .NET 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.

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

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

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var consumed = 0;
var consumer = await Consumer.Create(
    new ConsumerConfig(
        streamSystem,
        "my-stream")
    {
        Reference = "my-reference", ①
        MessageHandler = async (stream, consumer, context, message) =>
        {
            if (consumed++ % 10000 == 0)
            {
                await consumer.StoreOffset(context.Offset).ConfigureAwait(false); ②
            }

            Console.WriteLine($"Received message:
{Encoding.UTF8.GetString(message.Data.Contents)}");
            await Task.CompletedTask.ConfigureAwait(false);
        }
    }
).ConfigureAwait(false);

await consumer.Close().ConfigureAwait(false); ⑤
await streamSystem.Close().ConfigureAwait(false);
```

① Set the consumer Reference (mandatory for offset tracking)

② Store the current offset on some condition

The snippet above uses `consumer.StoreOffset(context.Offset)` to store at the offset of the current message.

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.

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.

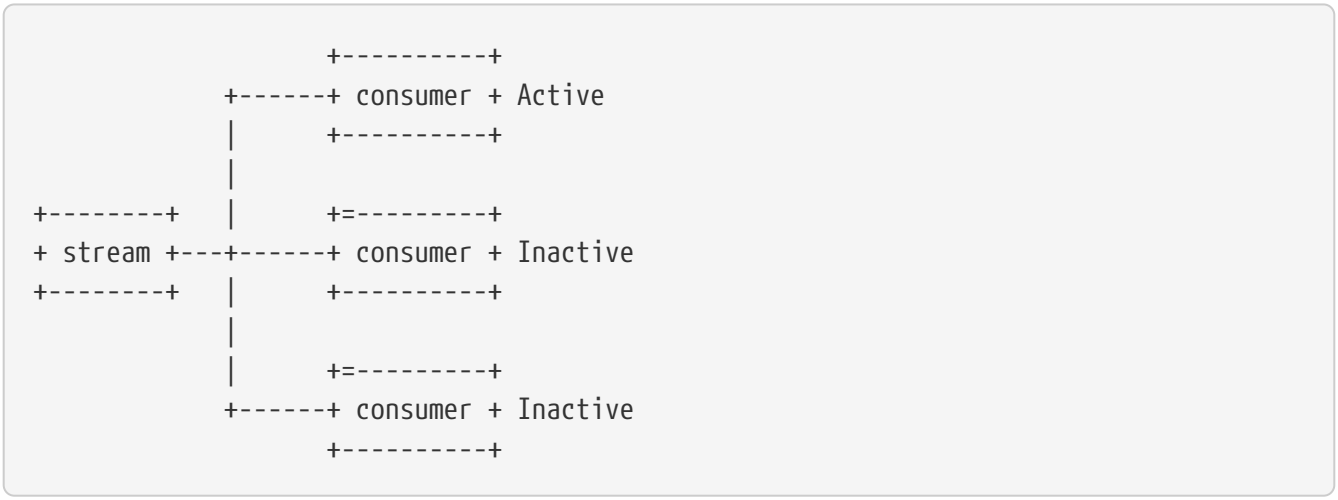
A typical sequence of events would be the following:

- 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.

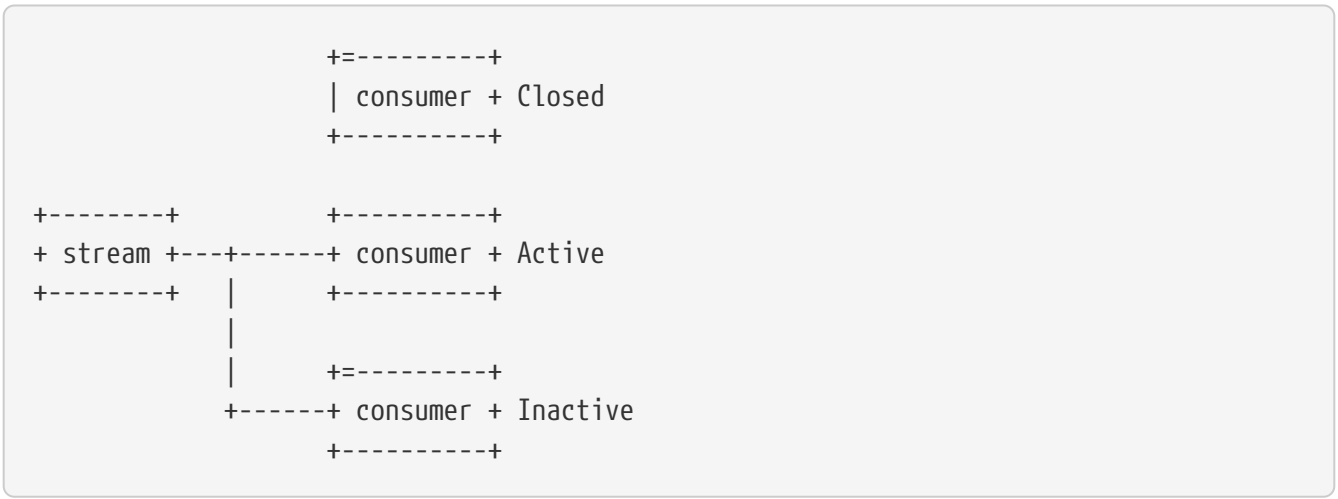
The next figures illustrates this mechanism. There can be only one active consumer:

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:

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

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var consumer = await Consumer.Create(
    new ConsumerConfig(
        streamSystem,
        "my-stream")
    {
        Reference = "my-reference", ①
        IsSingleActiveConsumer = true, ②
```

① Set the `Reference` 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 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 `ConsumerConfig#ConsumerUpdateListener()` method like demonstrated in the following snippet:

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

```
var streamSystem = await StreamSystem.Create(
    new StreamSystemConfig()
).ConfigureAwait(false);

var consumer = await Consumer.Create(
    new ConsumerConfig(
        streamSystem,
```

```
        "my-stream")
    {
        Reference = "my-reference", ①
        IsSingleActiveConsumer = true, ②
        ConsumerUpdateListener = async (consumerRef, stream, isActive) => ③
        {
           var offset =  await streamSystem.QueryOffset(consumerRef, stream)
.ConfigureAwait(false);
           return new OffsetTypeOffset(offset);
        },
```

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

② Enable single active consumer

③ Handle `ConsumerUpdateListener` callback

Low Level and High Level classes

.NET stream client provides two types of classes:

- Low-level classes
- High-level classes

High-level classes (`Producer` and `Consumer`) are a wrapper around the Low-Level classes (`Raw*Producer` and `Raw*Consumer`)

`Producer` and `Consumer` classes handle auto-reconnection, metadata updates, super-stream and some low-level client behaviour.

It would be best to use `Producer` and `Consumer` classes unless you need to handle the low-level details.