RabbitMQ Stream .Net Client

Table of Contents

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.
- Stream compression interface: `RabbitMQ.Stream.Client.ICompressionCodec` and `RabbitMQ.Stream.Client.StreamCompressionCodecs` are used to implement custom compression codecs.

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
```

① `Microsoft.Extensions.Logging` is used to log the events. (not shipped with the client)

② `RabbitMQ.Stream.Client` is the main package to use the client

③ `RabbitMQ.Stream.Client.Reliable` contains the `Producer` and `Consumer` implementations

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);
}


await
confirmationTaskCompletionSource.Task.WaitAsync(TimeSpan.FromSeconds(5)).ConfigureAwai
t(false); 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);
```

```
await
consumerTaskCompletionSource.Task.WaitAsync(TimeSpan.FromSeconds(2)).ConfigureAwait(fa
lse); 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 client is shipped with only with `Microsoft.Extensions.Logging.Abstractions` and you can use any logger you want.

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. `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 three 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 certificates from PEM files

Creating an StreamSystem that uses TLS and external authentication

```
private static async Task CreateTlsExternal()
{
    var ssl = new SslOption() ①
    {
        Enabled = true,
        ServerName = "server_name",
        CertPath = "certs/client/keycert.p12",
        CertPassphrase = null, // in case there is no password
        CertificateValidationCallback = (sender, certificate, chain, errors) => true,
    };

    var config = new StreamSystemConfig()
    {
        UserName = "user_does_not_exist",
        Password = "password_does_not_exist",
        Ssl = ssl,
        Endpoints = new List<EndPoint>(new List<EndPoint>()
        {
            new DnsEndPoint("server_name", 5551)
        }),

        AuthMechanism = AuthMechanism.External, ②
    };

    var streamSystem = await StreamSystem.Create(config).ConfigureAwait(false);

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

}
```

① Enable TLS and configure the certificates

② Set the external authentication mechanism

Note: you need the `rabbitmq_auth_mechanism_ssl` plugin enabled on the server side to use external authentication. `AuthMechanism.External` can be used from RabbitMQ server 3.11.19 and RabbitMQ 3.12.1 onwards.

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()
            {
                Enabled = true,
                AcceptablePolicyErrors = SslPolicyErrors.RemoteCertificateNotAvailable
| ①
                                         SslPolicyErrors.RemoteCertificateChainErrors
|
                                         SslPolicyErrors.RemoteCertificateNameMismatch
            }
        }
    ).ConfigureAwait(false);
    await streamSystem.Close().ConfigureAwait(false);
}
```

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`
`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`

Parameter Name	Description	Default
`ConnectionPoolConfig`	Define the connection pool policies. See Connection Pool for more details.	1 producer/consumer per connection

Connection pool

Introduced on version 1.8.0. With the connection pool you can define how many producers and consumers can be created on a single connection and the `ConnectionCloseConfig`

```
ConnectionPoolConfig = new ConnectionPoolConfig()
{
   ProducersPerConnection = 2,
   ConsumersPerConnection = 3,
   ConnectionCloseConfig = new ConnectionCloseConfig()
   {
       Policy = ConnectionClosePolicy.CloseWhenEmpty,
   }
}
```

By default, the connection pool is set to 1 producer and 1 consumer per connection. The maximum number of producers and consumers per connection is 200.

A high value can reduce the number of connections to the server but it could reduce the performance of the producer and the consumer.

A low value can increase the number of connections to the server but it could increase the performance of the producer and the consumer.

The consumers share the same handler, so if you have a high number of consumers per connection, the handler could be a bottleneck. It means that if there is a slow consumer all the other consumers could be slow.

TIP: You can use different StreamSystemConfig like:

```
streamSystemToReduceTheConnections = new StreamSystemConfig{
        ConnectionPoolConfig = new ConnectionPoolConfig() {
                ConsumersPerConnection = 50, // high value
                ProducersPerConnection = 50,  // high value
        }
}

streamSystemToIncreaseThePerformances = new StreamSystemConfig{
        ConnectionPoolConfig = new ConnectionPoolConfig() {
                ConsumersPerConnection = 1, // low value
                ProducersPerConnection = 1,  // low value
        }
}
```

There is not a magic number, you have to test and evaluate the best value for your use case.

The `ConnectionCloseConfig` defines the policy to close the connection when the last producer or consumer is closed. - `CloseWhenEmpty` the connection is closed when the last producer or consumer is closed. - `CloseWhenEmptyAndIdle` the connection is closed when the last producer or consumer is closed and the connection is idle for a certain amount of time.

The policy `CloseWhenEmpty` covers the standard use cases when the producers or consumers have long life running.

The policy `CloseWhenEmptyAndIdle` is useful when producers or consumers have short live and the pool has to be fast to create a new entity. The parameter `IdleTime` defines the time to wait before closing the connection when the last producer or consumer is closed. The parameter `CheckIdleTime` defines the time to check if the connection is idle.

```
new ConnectionCloseConfig()
   {
       Policy = ConnectionClosePolicy.CloseWhenEmptyAndIdle,
       IdleTime = TimeSpan.FromMilliseconds(1000),
       CheckIdleTime = TimeSpan.FromMilliseconds(500)
   });
```

Note: You can't close the stream systems if there are producers or consumers still running with the `CloseWhenEmptyAndIdle` policy.

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);
}
```

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 `StreamSystem.CreateStream(..)` 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);
}
```

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

Parameter Name	Description	Default
`TimeoutMessageAfter`	Time to wait before considering a message as not confirmed.	3 seconds
`SuperStreamConfig`	The super stream configuration.	`null` (no super stream)
`StatusChanged`	The callback invoked when the producer status changes. See Producer Status for more details.	`null`

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)`.

The following 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

Algorithm	Overview	Implementation used
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")
    {
        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.

Staring from the 1.3.0 version, the `Consumer#MessageHandler` API runs in a separated `Task` and it is possible to use `async`/`await` in the handler.

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.

Parameter Name	Description	Default
`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`
`ICrc32`	The CRC32 implementation to use to validate the chunk server crc32 .	`StreamCrc32()`
`StatusChanged`	The callback invoked when the consumer status changes. See Consumer Status for more details.	`null`

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.

Check the CRC on Delivery

RabbitMQ Stream provides a CRC32 checksum on each chunk. The client library can check the checksum before parsing the chunk. By default the CRC32 checksum enabled (with the class `StreamCrc32()`), to disable it you need to set null to `ICrc32` interface in the `ConsumerConfig`:

Checking the CRC32 checksum on the chunk

```
Crc32 = new StreamCrc32() ①
{
    FailAction = (consumerInstance) => ChunkAction.Skip ②
},
OffsetSpec = new OffsetTypeTimestamp(),
```

① An implementation of the `ICrc32` interface. You are free to use any implementation you want. By default the client library uses `StreamCrc32` which is a simple implementation of the `ICrc32` interface.

② The function FailAction is called when the CRC32 checksum is not valid.

`FailAction` return types:

Parameter Name	Description
`ChunkAction.Skip`	Skip the chunk and continue consuming the next chunk.
`ChunkAction.TryToProcess`	Try to process the chunk even if the CRC32 checksum is not valid.

If `FailAction` is `null` the library will use `ChunkAction.Skip` as default value.

`FailAction` has `sourceConsumer` as parameter, which is the consumer that failed to validate the CRC32 checksum. the code in the `FailAction` should be short, fast and safe, as it is executed in the consumer thread.

It is recommended to leave it enabled, as it allows the client library to detect corrupted chunks and avoid parsing them. Disabling the CRC32 *could* improve performance, but it is not recommended as it can lead to parsing corrupted chunks and unexpected behavior.

Flow Control

This section covers how a consumer can tell the broker when to send more messages. By default, the broker keeps sending messages as soon as a chunk is received. This strategy works fine if message processing is fast enough.

With `ConsumerFlowStrategy` it is possible to control when the broker sends more messages to the consumer.

Setting a consumer flow control strategy

```
var consumerConfig = new ConsumerConfig(streamSystem, "MyStream")
{
    FlowControl = new FlowControl() ①
    {
        Strategy = ConsumerFlowStrategy.CreditsAfterParseChunk, ②
    },
};
```

① Set the flow control strategy

② Set the flow control strategy to `ConsumerFlowStrategy.CreditAfterParseChunk` as soon as the chunk is parsed. See `ConsumerFlowStrategy` for more details. If message processing takes longer, one can be tempted to process messages in parallel with `Tasks`. This will make the handle method return immediately and the broker will keep sending messages, potentially overflowing the consumer.

What we miss in the parallel processing case is a way to tell the library we are done processing a message and that we are ready at some point to handle more messages. This is the goal of the `consumer.RequestCredits()` method. With `consumer.RequestCredits()` the consumer can tell the broker it is ready to receive more messages. It is possible also the `Pause` the consumer by not asking for credits.

Setting a consumer manual control strategy

```
var consumerConfig = new ConsumerConfig(streamSystem, "MyStream")
{
    FlowControl = new FlowControl() ①
    {
        Strategy = ConsumerFlowStrategy.ConsumerCredits, ②
    },
    // here we simulate a manual flow control
    // when the consumer has consumed 10 messages, it will request more credits
    MessageHandler = (_, rawConsumer, _, _) => consumed++ % 10 == 0 ?
rawConsumer.Credits() : ③
        Task.CompletedTask
};
```

① Set the flow control strategy

② Set the flow control strategy to `ConsumerFlowStrategy.ConsumerCredits` to control when the broker sends more messages.

③ Request credits when the consumer is ready to receive more messages.

`ConsumerFlowStrategy` values:

Parameter Name	Description
`CreditsBeforeParseChunk`	The broker sends messages to the consumer before parsing the chunk. (Default)
`CreditsAfterParseChunk`	The broker sends messages to the consumer after parsing the chunk.
`ConsumerCredits`	The broker sends messages to the consumer only when the consumer requests credits.

NOTE

Manual flow control with `ConsumerCredits`

With manual flow control, the consumer can control when the broker sends more messages. It gives all the control user side. If the server doesn't receive a request for credits, it will not send more messages. So you could end up with a consumer not consuming anything.

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.

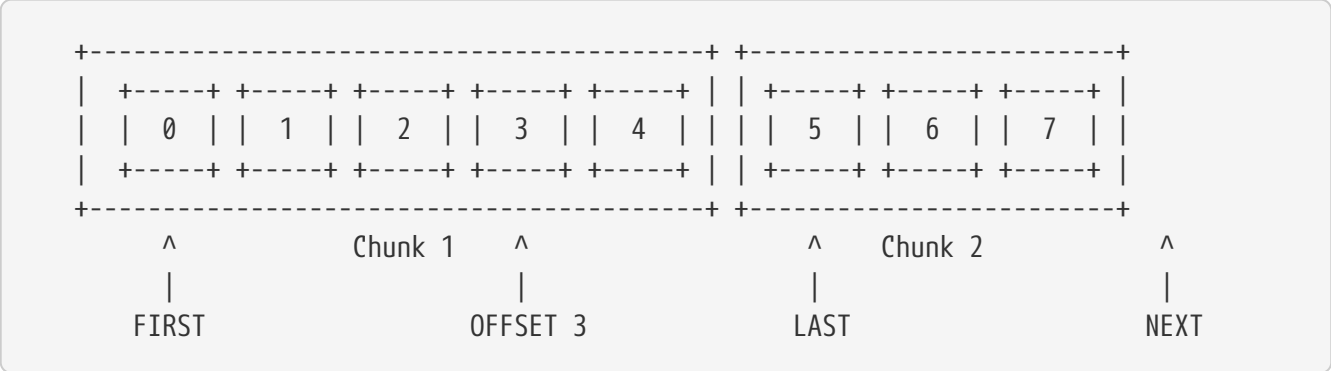
NOTE

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 **reference**. 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);
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. It is possible to store the offset in a more generic way with `StreamSystem.StoreOffset(reference,stream, offsetValue)`

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.

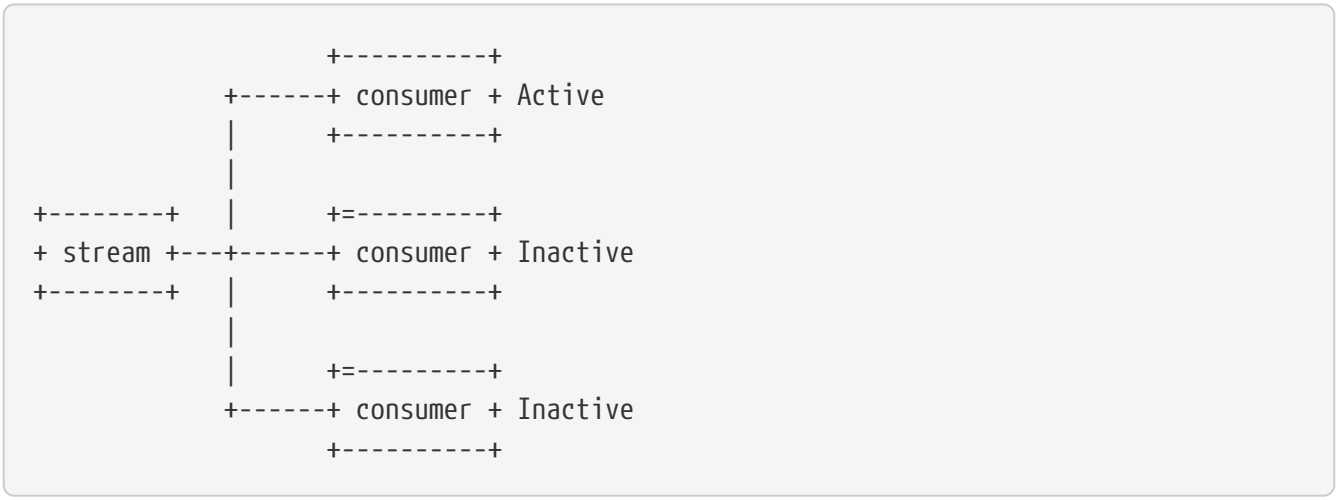
A typical sequence of events would be the following:

- 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.

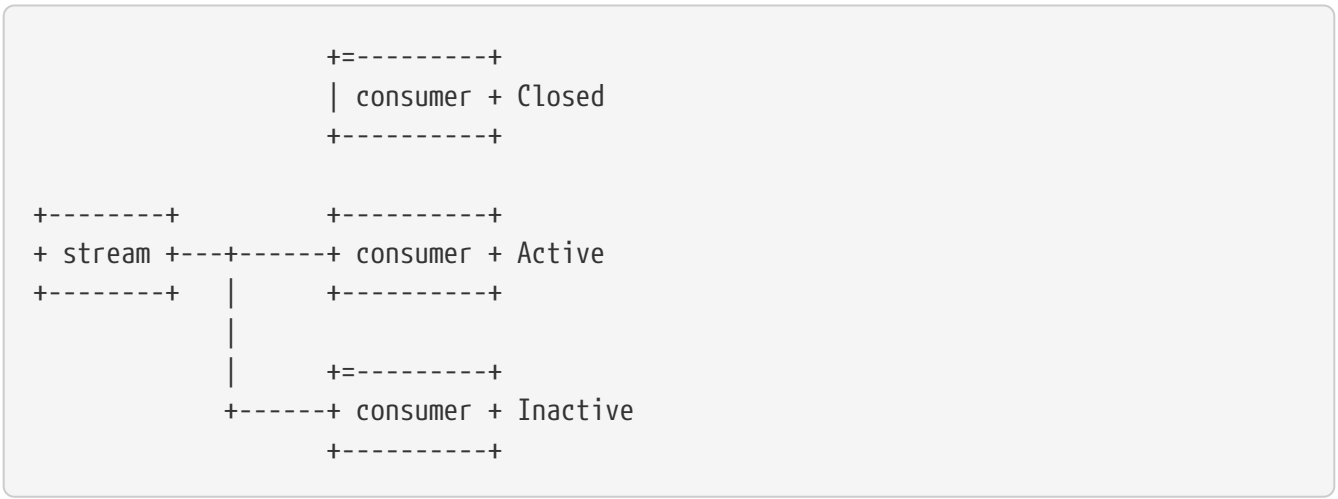
The next figures illustrates this mechanism. There can be only one active consumer:

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

Producer/Consumer change status callback

`Producer` and `Consumer` classes provide a callback to handle the status change. It is possible to configure the event using the configuration `StatusChanged` property.

like the following snippet:

```
var conf = new ConsumerConfig(system, stream)
{
    StatusChanged = (statusInfo) =>
    {
        Console.WriteLine($"Consumer status changed to {statusInfo}");
    }
};
var consumer = Consumer.Create(conf);
```

the `statusInfo` contains the following information:

Parameter Name	Description
`From`	The previous status
`To`	The new status
`Stream`	The stream where the Producer or the Consumer is connected
`Identifier`	The identifier of the Producer or the Consumer
`Partition`	The partition in case of super stream
`Reason`	The reason of the status change. See Status Reason for more details.

`statusInfo.Reason` values:

Parameter Name	Description
`None`	No reason, default value
`UnexpectedlyDisconnected`	The client was unexpectedly disconnected from the server
`MetaDataUpdate`	The server has updated the metadata of the stream. See this presentation about metadata update for more details
`ClosedByUser`	The client was closed by the user
`ClosedByStrategyPolicy`	The Producer or Consumer was closed by the strategy policy
`BoolFailure`	The Producer or Consumer has failed to connect to the server.

A full example of the status change callback can be found in here.

Low Level and High Level classes

NET stream client provides two types of classes:

- Low-level classes
- High-level classes

Low-level classes

- `RabbitMQ.Stream.Client.RawProducer` - Low-level producer class
- `RabbitMQ.Stream.Client.RawConsumer` - Low-level consumer class
- `RabbitMQ.Stream.Client.RawSuperStreamProducer` - Low-level super-stream producer class
- `RabbitMQ.Stream.Client.RawSuperStreamConsumer` - Low-level super-stream consumer class

The Classes are used to interact with the stream server in a low level way. They are used to create streams, publish messages, consume messages, etc. They give you all the callbacks to manually handle events like:

- `Disconnection`
- `Metadata update`

Creating a Raw Producer

```
var rawProducer = await streamSystem.CreateRawProducer( ①
    new RawProducerConfig("my-stream")
    {
        ConnectionClosedHandler = async reason => ②
        {
            Console.WriteLine($"Connection closed with reason: {reason}");
            await Task.CompletedTask.ConfigureAwait(false);
        },
```

```
        MetadataHandler = update => ③
        {
            Console.WriteLine($"Metadata Stream updated: {update.Stream}");
            return Task.CompletedTask;
        },
        ConfirmHandler = confirmation => ④
        {
            Console.WriteLine(confirmation.Code == ResponseCode.Ok
                ? $"Message confirmed: {confirmation.PublishingId}"
                : $"Message: {confirmation.PublishingId} not confirmed with error:
{confirmation.Code}");
        }
    }
```

① Create a `RawProducer` instance

② Event in case of disconnection

③ Event in case of MetadataHandler update. This event is triggered by the server when a stream changes topology like deleted or added/removed mirrors

④ ConfirmHandler event. This event is triggered when a `PublishingId` message is confirmed by the server with or without an error.

Like the `RawProducer` class, the `Raw*` classes have the same events to handle the disconnection and metadata update.

It is up to the user to handle the disconnection and metadata update events.

WARNING

Be careful when using the `Raw*` classes.

They are low-level classes and you need to handle the disconnection and metadata update events. If you don't handle them, you will end up with a disconnected client and you will not be able to reconnect to the server.

"RawProducer:send" is not thread-safe. You need to synchronize access to it. "RawProducer" does not handle the timeout/error confirmation messages. You need to handle it yourself.

High-level classes

- `Producer` - High-level producer class
- `Consumer` - High-level consumer class

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

The `Producer` traces the sent and received messages to give back to the user the original message sent to the server and also handle the message timeout. See [confirmation-status] for more details.

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

This is a full example how to deal with disconnections and metadata updates.

Query Stream/SuperStream

The `StreamSytem` class expose methods to query a stream or super stream. The following methods are available:

Method	Description	Valid for
`QuerySequence(string reference, string stream)`	Retrieves the last publishing id for given a producer Reference and stream. Useful for a producer wants to know the last published id.	Stream
`QueryOffset(string reference, string stream)`	Retrieves the last consumer offset stored for a given consumer Reference and stream. Useful for as consumer wants to know the last stored offset.	Stream
`TryQueryOffset(string reference, string stream)`	Like `QueryOffset` but returns `null` if the offset was not found without throwing `OffsetNotFoundException` exception .	Stream
`QueryPartition(string superStream)`	Returns the list of stream partitions for a given super stream.	SuperStream

Method	Description	Valid for
`StreamStats(string stream)`	Returns the stream statistics: - `FirstOffset()`: first offset in the stream - `CommittedChunkId()`: the ID (offset) of the committed chunk (block of messages) in the stream. It is the offset of the first message in the last chunk confirmed by a quorum of the stream cluster members (leader and replicas). The committed chunk ID is a good indication of what the last offset of a stream can be at a given time. The value can be stale as soon as the application reads it though, as the committed chunk ID for a stream that is published to changes all the time. return committed offset in this stream	Stream

Super Streams (Partitioned Streams)

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

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

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

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

WARNING *Super streams do not deprecate streams*

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

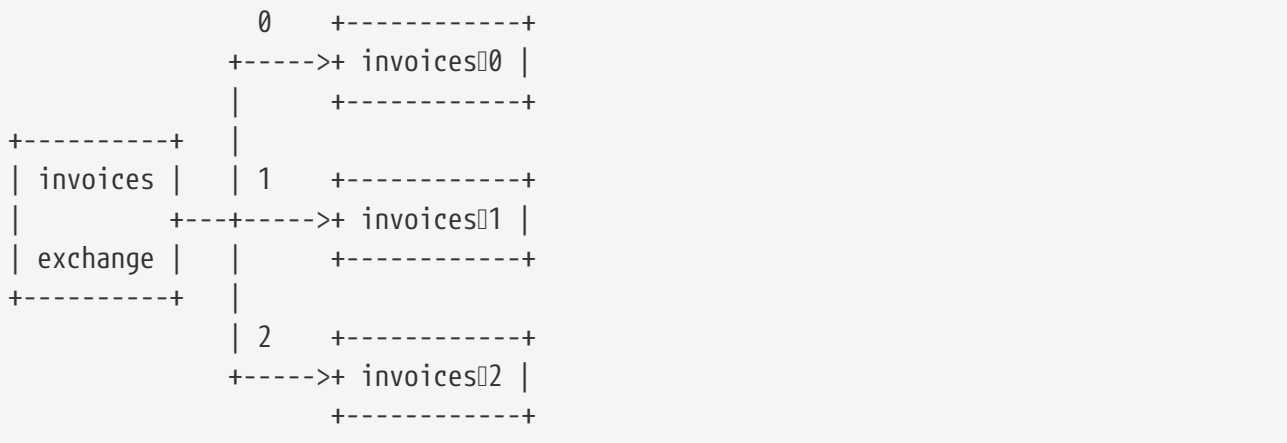
Topology

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

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

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

The topology of a super stream is defined with bindings between an exchange and queues



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

Super Stream Creation

It is possible to create the topology of a super stream with any AMQP 0.9.1 library or with the

management plugin, but the `rabbitmq-streams add_super_stream` command is a handy shortcut. Here is how to create an invoices super stream with 3 partitions:

Creating a super stream from the CLI

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

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

Publishing to a Super Stream

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

Creating a Producer for a Super Stream

```
var producer = await Producer.Create(
    new ProducerConfig(system,
            // Costants.StreamName is the Exchange name
            // invoices
            Costants.StreamName) ①
        {
            SuperStreamConfig = new SuperStreamConfig() ②
            {
                // The super stream is enable and we define the routing hashing
algorithm
                Routing = msg => msg.Properties.MessageId.ToString() ③
            }
        }, logger).ConfigureAwait(false);
const int NumberOfMessages = 1_000_000;
for (var i = 0; i < NumberOfMessages; i++)
{
    var message = new Message(Encoding.Default.GetBytes($"my_invoice_number{i}")) ④
    {
        Properties = new Properties() {MessageId = $"id_{i}"}
    };
    await producer.Send(message).ConfigureAwait(false);
```

① Configure the `Producer` with the super stream name

② Enable the Super Stream mode

③ Provide the logic to get the routing key from a message

④ Send the messages to the super stream

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

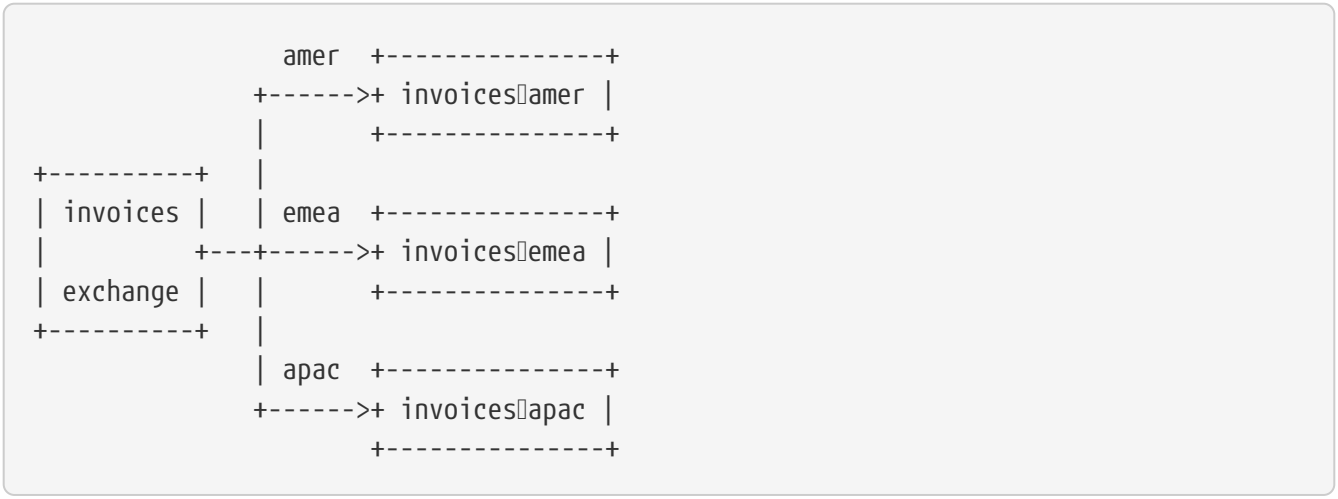
The client uses 32-bit MurmurHash3 by default to hash the routing key. This hash function provides good uniformity and it is compatible with the other clients.

Resolving Routes with Bindings

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

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

A super stream with a partition for a region in a world



To create this topology:

```
rabbitmq-streams add_super_stream invoices  --routing-keys apac,emea,amer
```

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

Enabling the "key" routing strategy

```
var producer = await Producer.Create(
    new ProducerConfig(system,
            // Costants.StreamName is the Exchange name
            // invoices
            Costants.StreamNameC)
        {
            SuperStreamConfig = new SuperStreamConfig()
            {
                // The super stream is enable and we define the routing hashing
algorithm
                Routing = msg => msg.Properties.MessageId.ToString(), ①
                RoutingStrategyType = RoutingStrategyType.Key ②
            }
        }, logger).ConfigureAwait(false);
const int NumberOfMessages = 1_000_000;
```

```
for (var i = 0; i < NumberOfMessages; i++)
{
    var key = keys[i % 3];

    var message = new Message(Encoding.Default.GetBytes($"hello{i}"))
    {
        Properties = new Properties() {MessageId = $"{key}"}
    };
    await producer.Send(message).ConfigureAwait(false);
```

① Extract the routing key

② Enable the "key" routing strategy

Internally the client will query the broker to resolve the destination streams for a given routing key, making the routing logic from any exchange type available to streams.

If there is no binding for a routing key, the client will raise an exception `RouteNotFoundException`.

`RouteNotFoundException` the message is not routed to any stream.

Deduplication

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

Creating a DeduplicatingProducer for a Super Stream

```
var producer = await DeduplicatingProducer.Create(
    new DeduplicatingProducerConfig(system,
            // Costants.StreamName is the Exchange name
            // invoices
            Costants.StreamName,
            "my-deduplication-producer" ①
        ) ①
        {
            SuperStreamConfig = new SuperStreamConfig() ②
            {
                // The super stream is enable and we define the routing hashing
algorithm
                Routing = msg => msg.Properties.MessageId.ToString() ③
            }
        }).ConfigureAwait(false);
const int NumberOfMessages = 1_000_000;
for (var i = 0; i < NumberOfMessages; i++)
{
    var message = new Message(Encoding.Default.GetBytes($"hello{i}")) ④
    {
        Properties = new Properties() {MessageId = $"hello{i}"}
    };
```

```
    await producer.Send(1, message).ConfigureAwait(false);
```

① Configure the `DeduplicatingProducer` with the super stream name and the reference.

② Enable the Super Stream mode

③ Provide the logic to get the routing key from a message. Send the messages providing the publishing ID.

Consuming From a Super Stream

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

Super Stream Consumer in Practice

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

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

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

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

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

Declaring a Super Stream Consumer

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

Declaring a super stream consumer

```
var consumer = await Consumer.Create(new ConsumerConfig(system, Costants.StreamName)
{
    IsSuperStream = true, // Mandatory for enabling the super stream ①
    // this is mandatory for super stream single active consumer
    // must have the same ReferenceName for all the consumers
    Reference = "MyApp",
```

```
    OffsetSpec = new OffsetTypeFirst(),
    MessageHandler = async (stream, consumerSource, context, message) => ②
    {
        loggerMain.LogInformation("Consumer Name {ConsumerName} " +
                                  "-Received message id: {PropertiesMessageId} body:
{S}, Stream {Stream}, Offset {Offset}",
            consumerName, message.Properties.MessageId,
Encoding.UTF8.GetString(message.Data.Contents),
            stream, context.Offset);
```

① Set the super stream name

② Close the consumer when it is no longer necessary

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

Offset Tracking

The semantic of offset tracking for a super stream consumer are roughly the same as for an individual stream consumer. There are still some subtle differences, so a good understanding of offset tracking.

The offset tracking is per stream.

Single Active Consumer Support

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

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

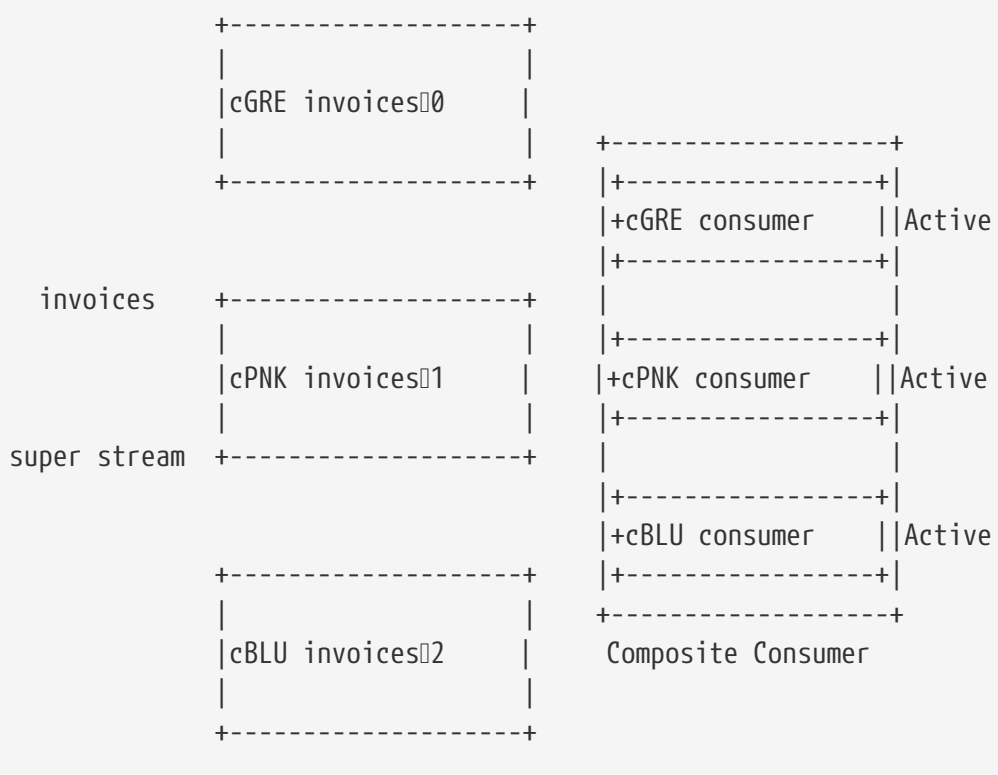
Let's take an example with a 3-partition super stream:

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

The following figure illustrates how the client library supports the combination of the super stream and single active consumer features. It uses a composite consumer that creates an individual

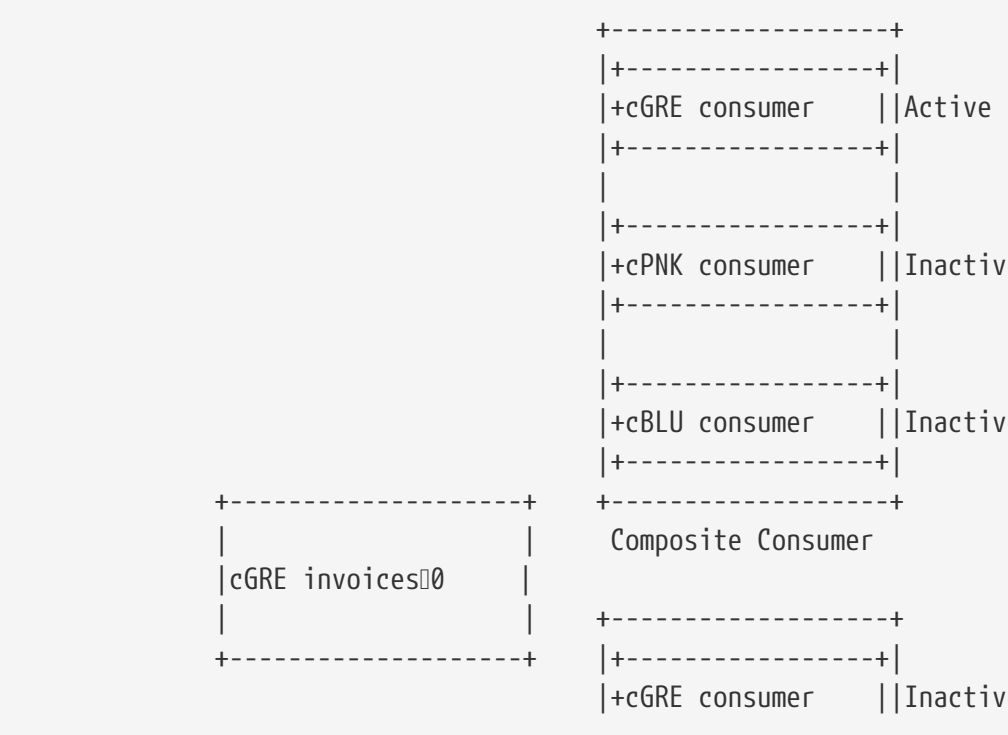
consumer for each partition of the super stream. If there is only one single active consumer instance with a given name for a super stream, each individual consumer is active.

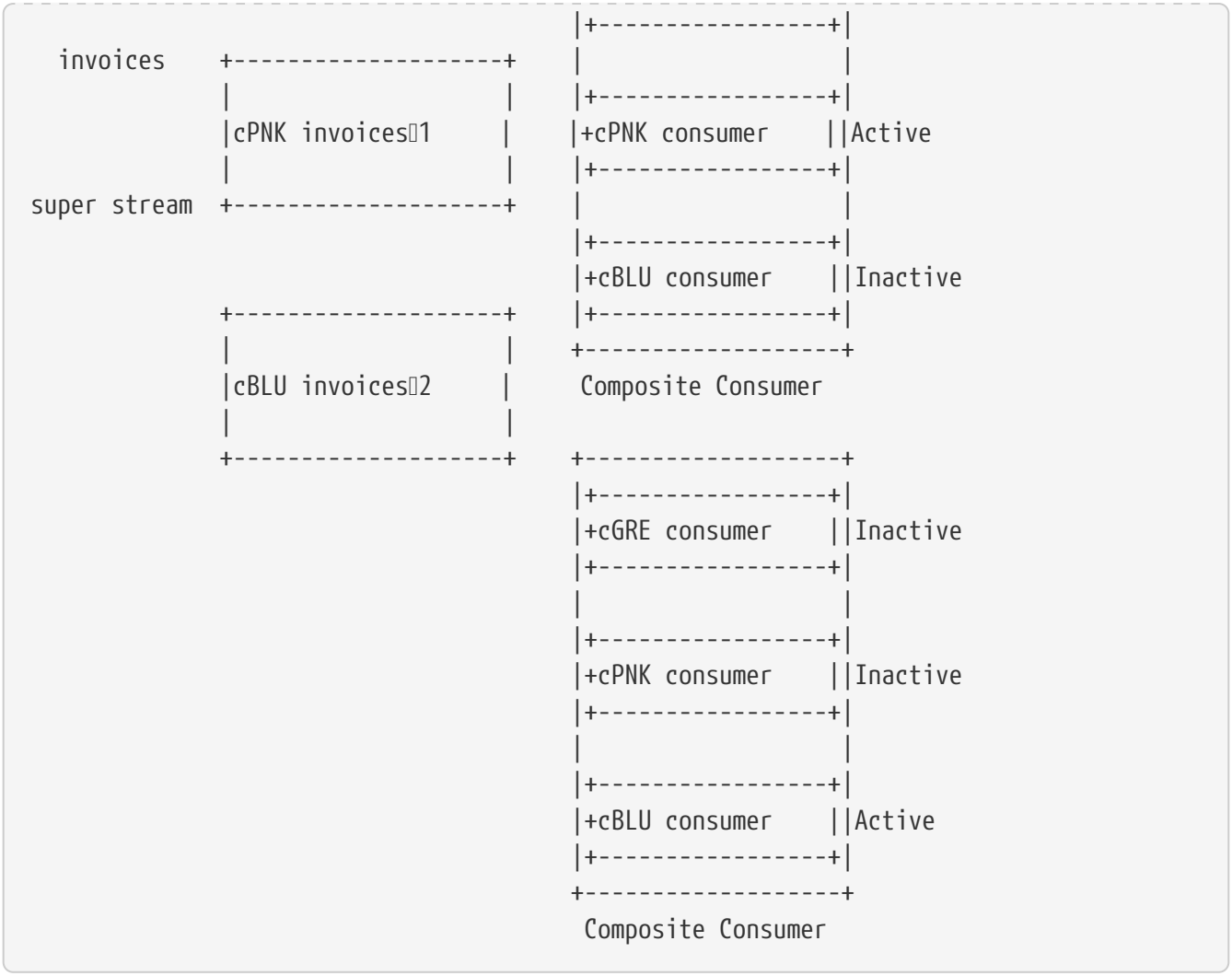
A single active consumer on a super stream is a composite consumer that creates an individual consumer for each partition



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

Consumer instances spread across the super stream partitions and are activated accordingly





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

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

The example above uses by default manual offset tracking. It looks up the latest stored offset when a consumer becomes active to start consuming at the appropriate offset and it stores the last dispatched offset when a consumer becomes inactive.

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

Using manual offset tracking for a super stream single active consumer

```
    await consumerSource.StoreOffset(context.Offset).ConfigureAwait(false); ①
    await Task.CompletedTask.ConfigureAwait(false);
},
IsSingleActiveConsumer = true, // mandatory for enabling the Single Active Consumer ②
ConsumerUpdateListener = async (reference, stream, isActive) => ③
{
```

```
loggerMain.LogInformation($"*****************************************************");
    loggerMain.LogInformation("reference {Reference} stream {Stream} is active:
{IsActive}", reference,
        stream, isActive);

    ulong offset = 0;
    try
    {
        offset = await system.QueryOffset(reference, stream).ConfigureAwait(false);
    }
    catch (OffsetNotFoundException e)
    {
        loggerMain.LogInformation("OffsetNotFoundException {Message}, will use
OffsetTypeNext", e.Message);
        return new OffsetTypeNext();
    }

    if (isActive)
    {
        loggerMain.LogInformation("Restart Offset {Offset}", offset);
    }


loggerMain.LogInformation($"*****************************************************");
    await Task.CompletedTask.ConfigureAwait(false);
    return new OffsetTypeOffset(offset + 1); ④
},
```

① Store manually the offset

② Enable single active consumer

③ Set `ConsumerUpdateListener`

④ Return stored offset + 1 or default when consumer becomes active

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

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

Note the `ConsumerUpdateListener` is called for a *partition*, that is an individual stream.

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

Super Stream with Single Active Consumer Example

You can follow the README on the link to run the Super Stream example with the single active consumer feature.

Advanced Topics

Filtering

WARNING	Filtering requires **RabbitMQ 3.13** or more and the stream_filter feature flag enabled.

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

The filtering feature works as follows:

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

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

Filtering on the Publishing Side

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

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

```
// This is mandatory for enabling the filter
Filter = new ProducerFilter()
{
    FilterValue = message => message.ApplicationProperties["state"].ToString(), ①
}
```

① Get filter value from `state` application property

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

Filtering on the Consuming Side

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

Declaring a consumer with a filter value and filtering logic

```
var consumedMessages = 0;
var consumer = await Consumer.Create(new ConsumerConfig(system, streamName)
{
    OffsetSpec = new OffsetTypeFirst(),

    // This is mandatory for enabling the filter
    Filter = new ConsumerFilter()
    {
        Values = new List<string>() {"Alabama"}, ①
        PostFilter = message => message.ApplicationProperties["state"].Equals("
Alabama"), ②
        MatchUnfiltered = true
    },
    MessageHandler = (_, _, _, message) =>
    {
        consumerLogger.LogInformation("Received message with state {State} - consumed
{Consumed}",
            message.ApplicationProperties["state"], ++consumedMessages);
        return Task.CompletedTask;
    }
```

① Set filter value

② Set filtering logic

The filter logic is a `Predicate<Message>`. It must return `true` if a message is accepted.

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

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

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

Considerations on Filtering

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

define to avoid processing unwanted messages.

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

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

Deal with broker disconnections, reconnections and metadata update events

The classes `Producer` and `Consumer` automatically handle broker disconnections and reconnections.

It is important to know what happens when a broker is disconnected and reconnected. See this presentation about that.

The `Producer` and `Consumer` classes also handle metadata update events. When a stream is deleted, the `Producer` and `Consumer` classes automatically close the underlying `Raw*Producer` and `Raw*Consumer` objects.

The classes provides two interfaces:

- ResourceAvailableReconnectStrategy (checks when the resource is available)
- ReconnectStrategy (reconnects to the broker)

That by default implement the reconnection strategy using a back off algorithm. You can provide your own implementation of these interfaces to customize the reconnection strategy. Most of the time, the default implementation is enough.

You can find a full example here

Update Secret

To update the secret, you can use:

```
await system.UpdateSecret(await NewAccessToken()).ConfigureAwait(false);
```

You can see a full example with a Keycloak integration here