

Consumer Details



- [Durable versus ephemeral consumers](#)
 - [Durable consumers](#)
 - [Ephemeral consumers](#)
 - [Push and Pull consumers](#)
 - [Push consumers](#)
 - [Pull consumers](#)
 - [Pull](#)
 - [Push](#)
 - [Ordered Consumers](#)
 - [Delivery reliability](#)
 - ["Dead Letter Queues" type functionality](#)
-

Consumers are how client applications get the messages stored in the streams. You can have many consumers on a single stream. Consumers are like a view on a stream, can filter messages and have some state (maintained by the servers) associated with them.

Consumers can be 'durable' or 'ephemeral'.

Durable versus ephemeral consumers

Durable consumer persist message delivery progress on the server side. A durable consumer can be retrieved by name and shared between client instances for load balancing. It can be made highly available through replicas.

An ephemeral consumer does not persist delivery progress and will automatically be deleted when there are no more client instances connected.

Durable consumers

Durable consumers are meant to be used by multiple instances of an application, either to distribute and scale out the processing or to persist the position of the consumer over the stream between runs of an application.

Durable consumers, as the name implies, are meant to last 'forever' and are typically created and deleted administratively rather than by the application code, which only needs to specify the durable consumer's well-known name to use it.

You create a durable consumer using the `nats consumer add` CLI tool command, or programmatically by passing a durable name option to the subscription creation call.

Ephemeral consumers

Ephemeral consumers are meant to be used by a single instance of an application (e.g. to get its own replay of the messages in the stream).

Ephemeral consumers are not meant to last 'forever', they are defined automatically at subscription time by the client library and disappear after the application disconnects.

You (automatically) create an ephemeral consumer when you call the `js.Subscribe()` function without specifying the Durable or Bind subscription options. Calling `Drain()` on that subscription automatically deletes the underlying ephemeral consumer. You can also explicitly create an ephemeral consumer by not passing a durable name option to the `jsm.AddConsumer()` call.

Ephemeral consumers otherwise have the same control over message acknowledgement and re-delivery as durable consumers.

Push and Pull consumers

Clients implement two implementations of consumers identified as 'push' or 'pull'.

Push consumers

Push consumers receive messages on a specific subject where message flow is controlled by the server. Load balancing is supported through NATS core queue groups. The messages from the stream are distributed automatically between the subscribing clients to the push consumers.

Pull consumers

Pull consumers request messages explicitly from the server in batches, giving the client full control over dispatching, flow control, pending (unacknowledged) messages and load balancing. Pull consuming clients make `fetch()` calls in a dispatch loop.

“`▯▯` We recommend using pull consumers for new projects. In particular when scalability, detailed flow control or error handling are a design focus. Most client APIs have been updated to provide convenient interfaces for consuming messages through callback handlers or iterators without the need to manage message retrieval.

`fetch()` calls can be immediate or have a defined timeout, allowing for either controlled (1 by 1) consumption or `realtime` delivery with minimal polling overhead.

Pull consumers create less CPU load on the NATS servers and therefore scale better (note that the push consumers are still quite fast and scalable, you may only notice the difference between the two if you have sustained high message rates).

Pull

Go

```
func ExampleJetStream() {
    nc, err := nats.Connect("localhost")
    if err != nil {
        log.Fatal(err)
    }

    ▯// Use the JetStream context to produce and consume messages
    ▯// that have been persisted.
    ▯js, err := nc.JetStream(nats.PublishAsyncMaxPending(256))
    ▯if err != nil {
    ▯▯log.Fatal(err)
    ▯}

    ▯js.AddStream(&nats.StreamConfig{
    ▯▯Name:      "F00",
    ▯▯Subjects: []string{"foo"},
    ▯})
}
```

```

js.Publish("foo", []byte("Hello JS!"))

// Publish messages asynchronously.
for i := 0; i < 500; i++ {
    js.PublishAsync("foo", []byte("Hello JS Async!"))
}

select {
case <-js.PublishAsyncComplete():
case <-time.After(5 * time.Second):
    fmt.Println("Did not resolve in time")
}

// Create Pull based consumer with maximum 128 inflight.
sub, _ := js.PullSubscribe("foo", "wq", nats.PullMaxWaiting(128))

ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()

for {
    select {
    case <-ctx.Done():
        return
    default:
    }

    // Fetch will return as soon as any message is available rather than wait until the
    // full batch size is available, using a batch size of more than 1 allows for higher throughput
    // when needed.
    msgs, _ := sub.Fetch(10, nats.Context(ctx))
    for _, msg := range msgs {
        msg.Ack()
    }
}

```

Java

```

package io.nats.examples.jetstream.simple;

```

```

import io.nats.client.*;
import io.nats.client.api.ConsumerConfiguration;
import io.nats.examples.jetstream.ResilientPublisher;
import java.io.IOException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;
import static io.nats.examples.jetstream.NatsJsUtils.createOrReplaceStream;

/**
 * This example will demonstrate simplified consume with a handler
 */
public class MessageConsumerExample {
    private static final String STREAM = "consume-stream";
    private static final String SUBJECT = "consume-subject";
    private static final String CONSUMER_NAME = "consume-consumer";
    private static final String MESSAGE_PREFIX = "consume";
    private static final int STOP_COUNT = 500;
    private static final int REPORT_EVERY = 100;
    private static final String SERVER = "nats://localhost:4222";

    public static void main(String[] args) {
        Options options = Options.builder().server(SERVER).build();
        try (Connection nc = Nats.connect(options)) {
            JetStreamManagement jsm = nc.jetStreamManagement();
            createOrReplaceStream(jsm, STREAM, SUBJECT);

            //Utility for filling the stream with some messages
            System.out.println("Starting publish...");
            ResilientPublisher publisher = new ResilientPublisher(nc, jsm, STREAM,
SUBJECT).basicDataPrefix(MESSAGE_PREFIX).jitter(10);
            Thread pubThread = new Thread(publisher);
            pubThread.start();

            // get stream context, create consumer and get the consumer context
            StreamContext streamContext;
            ConsumerContext consumerContext;
            CountDownLatch latch = new CountDownLatch(1);
            AtomicInteger atomicCount = new AtomicInteger();
            long start = System.nanoTime();

```

```

        streamContext = nc.getStreamContext(STREAM);

streamContext.createOrUpdateConsumer(ConsumerConfiguration.builder().durable(CONSUMER_NAME).build());

        consumerContext = streamContext.getConsumerContext(CONSUMER_NAME);

        MessageHandler handler = msg -> {
            msg.ack();
            int count = atomicCount.incrementAndGet();
            if (count % REPORT_EVERY == 0) {
                □ System.out.println("Handler" + ": Received " + count + " messages in " +
(System.nanoTime() - start) / 1_000_000 + "ms.");
            }
            if (count == STOP_COUNT) {
                latch.countDown();
            }
        };

        □ // create the consumer and install handler
        □ MessageConsumer consumer = consumerContext.consume(handler);
        □ //Waiting for the handler signalling us to stop
            latch.await();
            // When stop is called, no more pull requests will be made, but messages already
requested
            // will still come across the wire to the client.
            System.out.println("Stopping the consumer...");
            consumer.stop();
            // wait until the consumer is finished processing backlog
            while (!consumer.isFinished()) {
                Thread.sleep(10);
            }
            System.out.println("Final" + ": Received " + atomicCount.get() + " messages in " +
(System.nanoTime() - start) / 1_000_000 + "ms.");

            publisher.stop(); // otherwise the ConsumerContext background thread will complain
when the connection goes away
            pubThread.join();
        }
        catch (JetStreamApiException | IOException e) {
            // JetStreamApiException:

```

```

        //      1. the stream or consumer did not exist
        //      2. api calls under the covers theoretically this could fail, but practically
it won't.
        // IOException:
        //      likely a connection problem
        System.err.println("Exception should not handled, exiting.");
        System.exit(-1);
    }
    catch (Exception e) {
        System.err.println("Exception should not handled, exiting.");
        System.exit(-1);
    }
}
}

```

JavaScript

```

import { AckPolicy, connect, nanos } from "../../src/mod.ts";
import { nuid } from "../../nats-base-client/nuid.ts";

const nc = await connect();

const stream = nuid.next();
const subj = nuid.next();
const durable = nuid.next();

const jsm = await nc.jetstreamManager();
await jsm.streams.add({ name: stream, subjects: [subj] });

const js = nc.jetstream();
await js.publish(subj);
await js.publish(subj);
await js.publish(subj);
await js.publish(subj);

const psub = await js.pullSubscribe(subj, {
    mack: true,
    // artificially low ack_wait, to show some messages
    // not getting acked being redelivered
    config: {

```

```

    durable_name: durable,
    ack_policy: AckPolicy.Explicit,
    ack_wait: nanos(4000),
  },
});

(async () => {
  for await (const m of psub) {
    console.log(
      `[${m.seq}] ${
        m.redelivered ? '- redelivery ${m.info.redeliveryCount}' : ''
      }`
    );
    if (m.seq % 2 === 0) {
      m.ack();
    }
  }
})();

const fn = () => {
  console.log("[PULL]");
  psub.pull({ batch: 1000, expires: 10000 });
};

// do the initial pull
fn();
// and now schedule a pull every so often
const interval = setInterval(fn, 10000); // and repeat every 2s

setTimeout(() => {
  clearInterval(interval);
  nc.drain();
}, 20000);

```

Python

```

import asyncio

import nats
from nats.errors import TimeoutError

```

```

async def main():
    nc = await nats.connect("localhost")

    # Create JetStream context.
    js = nc.jetstream()

    # Persist messages on 'foo's subject.
    await js.add_stream(name="sample-stream", subjects=["foo"])

    for i in range(0, 10):
        ack = await js.publish("foo", f"hello world: {i}".encode())
        print(ack)

    # Create pull based consumer on 'foo'.
    psub = await js.pull_subscribe("foo", "psub")

    # Fetch and ack messages from consumer.
    for i in range(0, 10):
        msgs = await psub.fetch(1)
        for msg in msgs:
            print(msg)

    await nc.close()

if __name__ == '__main__':
    asyncio.run(main())

```

C#

```

// dotnet add package NATS.Net
using NATS.Net;
using NATS.Client.JetStream;
using NATS.Client.JetStream.Models;

await using var client = new NatsClient();

INatsJSContext js = client.CreateJetStreamContext();

// Create a stream

```

```

var streamConfig = new StreamConfig(name: "F00", subjects: ["foo"]);
await js.CreateStreamAsync(streamConfig);

// Publish a message
{
    PubAckResponse ack = await js.PublishAsync("foo", "Hello, JetStream!");
    ack.EnsureSuccess();
}

// Publish messages concurrently
List<NatsJSPublishConcurrentFuture> futures = new();
for (var i = 0; i < 500; i++)
{
    NatsJSPublishConcurrentFuture future
        = await js.PublishConcurrentAsync("foo", "Hello, JetStream 1!");
    futures.Add(future);
}

foreach (var future in futures)
{
    await using (future)
    {
        PubAckResponse ack = await future.GetResponseAsync();
        ack.EnsureSuccess();
    }
}

// Create a consumer with a maximum 128 inflight messages
INatsJSConsumer consumer = await js.CreateConsumerAsync("F00", new ConsumerConfig(name: "foo")
{
    MaxWaiting = 128,
});

using CancellationTokenSource cts = new(TimeSpan.FromSeconds(10));

while (cts.IsCancellationRequested == false)
{
    var opts = new NatsJSFetchOpts { MaxMsgs = 10 };
    await foreach (NatsJSMsg<string> msg in consumer.FetchAsync<string>(opts,

```

```

cancellationToken: cts.Token))
{
    await msg.AckAsync(cancellationToken: cts.Token);
}
}

```

C

```

#include "examples.h"

static const char *usage = ""\
"-gd          use global message delivery thread pool\n" \
"-sync        receive synchronously (default is asynchronous)\n" \
"-pull        use pull subscription\n" \
"-fc          enable flow control\n" \
"-count       number of expected messages\n";

static void
onMsg(natsConnection *nc, natsSubscription *sub, natsMsg *msg, void *closure)
{
    if (print)
        printf("Received msg: %s - %.*s\n",
            natsMsg_GetSubject(msg),
            natsMsg_GetDataLength(msg),
            natsMsg_GetData(msg));

    if (start == 0)
        start = nats_Now();

    // We should be using a mutex to protect those variables since
    // they are used from the subscription's delivery and the main
    // threads. For demo purposes, this is fine.
    if (++count == total)
        elapsed = nats_Now() - start;

    // Since this is auto-ack callback, we don't need to ack here.
    natsMsg_Destroy(msg);
}

static void

```

```

asyncCb(natsConnection *nc, natsSubscription *sub, natsStatus err, void *closure)
{
    printf("Async error: %u - %s\n", err, natsStatus_GetText(err));

    natsSubscription_GetDropped(sub, (int64_t*) &dropped);
}

int main(int argc, char **argv)
{
    natsConnection      *conn  = NULL;
    natsStatistics       *stats = NULL;
    natsOptions          *opts  = NULL;
    natsSubscription     *sub   = NULL;
    natsMsg              *msg   = NULL;
    jsCtx                *js    = NULL;
    jsErrCode            jerr   = 0;
    jsOptions            jsOpts;
    jsSubOptions         so;
    natsStatus           s;
    bool                 delStream = false;

    opts = parseArgs(argc, argv, usage);

    printf("Created %s subscription on '%s'.\n",
        (pull ? "pull" : (async ? "asynchronous" : "synchronous")), subj);

    s = natsOptions_SetErrorHandler(opts, asyncCb, NULL);

    if (s == NATS_OK)
        s = natsConnection_Connect(&conn, opts);

    if (s == NATS_OK)
        s = jsOptions_Init(&jsOpts);

    if (s == NATS_OK)
        s = jsSubOptions_Init(&so);
    if (s == NATS_OK)
    {
        so.Stream = stream;
        so.Consumer = durable;
    }
}

```

```

    if (flowctrl)
    {
        so.Config.FlowControl = true;
        so.Config.Heartbeat = (int64_t)1E9;
    }
}

if (s == NATS_OK)
    s = natsConnection_JetStream(&js, conn, &jsOpts);

if (s == NATS_OK)
{
    jsStreamInfo    *si = NULL;

    // First check if the stream already exists.
    s = js_GetStreamInfo(&si, js, stream, NULL, &jerr);
    if (s == NATS_NOT_FOUND)
    {
        jsStreamConfig  cfg;

        // Since we are the one creating this stream, we can delete at the end.
        delStream = true;

        // Initialize the configuration structure.
        jsStreamConfig_Init(&cfg);
        cfg.Name = stream;
        // Set the subject
        cfg.Subjects = (const char*[1]){subj};
        cfg.SubjectsLen = 1;
        // Make it a memory stream.
        cfg.Storage = js_MemoryStorage;
        // Add the stream,
        s = js_AddStream(&si, js, &cfg, NULL, &jerr);
    }
    if (s == NATS_OK)
    {
        printf("Stream %s has %" PRIu64 " messages (%" PRIu64 " bytes)\n",
            si->Config->Name, si->State.Msgs, si->State.Bytes);

        // Need to destroy the returned stream object.
    }
}

```

```

        jsStreamInfo_Destroy(si);
    }
}

if (s == NATS_OK)
{
    if (pull)
        s = js_PullSubscribe(&sub, js, subj, durable, &jsOpts, &so, &jerr);
    else if (async)
        s = js_Subscribe(&sub, js, subj, onMsg, NULL, &jsOpts, &so, &jerr);
    else
        s = js_SubscribeSync(&sub, js, subj, &jsOpts, &so, &jerr);
}

if (s == NATS_OK)
    s = natsSubscription_SetPendingLimits(sub, -1, -1);

if (s == NATS_OK)
    s = natsStatistics_Create(&stats);

if ((s == NATS_OK) && pull)
{
    natsMsgList list;
    int i;

    for (count = 0; (s == NATS_OK) && (count < total); )
    {
        s = natsSubscription_Fetch(&list, sub, 1024, 5000, &jerr);
        if (s != NATS_OK)
            break;

        if (start == 0)
            start = nats_Now();

        count += (int64_t) list.Count;
        for (i=0; (s == NATS_OK) && (i<list.Count); i++)
            s = natsMsg_Ack(list.Msgs[i], &jsOpts);

        natsMsgList_Destroy(&list);
    }
}

```

```

else if ((s == NATS_OK) && async)
{
    while (s == NATS_OK)
    {
        if (count + dropped == total)
            break;

        nats_Sleep(1000);
    }
}
else if (s == NATS_OK)
{
    for (count = 0; (s == NATS_OK) && (count < total); count++)
    {
        s = natsSubscription_NextMsg(&msg, sub, 5000);
        if (s != NATS_OK)
            break;

        if (start == 0)
            start = nats_Now();

        s = natsMsg_Ack(msg, &jsOpts);
        natsMsg_Destroy(msg);
    }
}

if (s == NATS_OK)
{
    printStats(STATS_IN|STATS_COUNT, conn, sub, stats);
    printPerf("Received");
}
if (s == NATS_OK)
{
    jsStreamInfo *si = NULL;

    // Let's report some stats after the run
    s = js_GetStreamInfo(&si, js, stream, NULL, &jerr);
    if (s == NATS_OK)
    {
        printf("\nStream %s has %" PRIu64 " messages (%" PRIu64 " bytes)\n",

```

```

        si->Config->Name, si->State.Msgs, si->State.Bytes);

    jsStreamInfo_Destroy(si);
}
if (delStream)
{
    printf("\nDeleting stream %s: ", stream);
    s = js_DeleteStream(js, stream, NULL, &jerr);
    if (s == NATS_OK)
        printf("OK!");
    printf("\n");
}
}
else
{
    printf("Error: %u - %s - jerr=%u\n", s, natsStatus_GetText(s), jerr);
    nats_PrintLastErrorStack(stderr);
}

// Destroy all our objects to avoid report of memory leak
jsCtx_Destroy(js);
natsStatistics_Destroy(stats);
natsSubscription_Destroy(sub);
natsConnection_Destroy(conn);
natsOptions_Destroy(opts);

// To silence reports of memory still in used with valgrind
nats_Close();

return 0;
}

```

A push consumer can also be used in some other use cases such as without a queue group, or with no acknowledgement or cumulative acknowledgements.

Push

Go

```

func ExampleJetStream() {
    nc, err := nats.Connect("localhost")

```

```

if err != nil {
    log.Fatal(err)
}

// Use the JetStream context to produce and consume messages
// that have been persisted.
js, err := nc.JetStream(nats.PublishAsyncMaxPending(256))
if err != nil {
    log.Fatal(err)
}

js.AddStream(&nats.StreamConfig{
    Name:      "F00",
    Subjects: []string{"foo"},
})

js.Publish("foo", []byte("Hello JS!"))

// Publish messages asynchronously.
for i := 0; i < 500; i++ {
    js.PublishAsync("foo", []byte("Hello JS Async!"))
}

select {
case <-js.PublishAsyncComplete():
case <-time.After(5 * time.Second):
    fmt.Println("Did not resolve in time")
}

// Create async consumer on subject 'foo'. Async subscribers
// ack a message once exiting the callback.
js.Subscribe("foo", func(msg *nats.Msg) {
    meta, _ := msg.Metadata()
    fmt.Printf("Stream Sequence  : %v\n", meta.Sequence.Stream)
    fmt.Printf("Consumer Sequence: %v\n", meta.Sequence.Consumer)
})

// Async subscriber with manual acks.
js.Subscribe("foo", func(msg *nats.Msg) {
    msg.Ack()
}, nats.ManualAck())

```

```

// Async queue subscription where members load balance the
// received messages together.
// If no consumer name is specified, either with nats.Bind()
// or nats.Durable() options, the queue name is used as the
// durable name (that is, as if you were passing the
// nats.Durable(<queue group name>) option.
// It is recommended to use nats.Bind() or nats.Durable()
// and preferably create the JetStream consumer beforehand
// (using js.AddConsumer) so that the JS consumer is not
// deleted on an Unsubscribe() or Drain() when the member
// that created the consumer goes away first.
// Check Godoc for the QueueSubscribe() API for more details.
js.QueueSubscribe("foo", "group", func(msg *nats.Msg) {
    msg.Ack()
}, nats.ManualAck())

// Subscribe to consume messages synchronously.
sub, _ := js.SubscribeSync("foo")
msg, _ := sub.NextMsg(2 * time.Second)
msg.Ack()

// We can add a member to the group, with this member using
// the synchronous version of the QueueSubscribe.
sub, _ = js.QueueSubscribeSync("foo", "group")
msg, _ = sub.NextMsg(2 * time.Second)
msg.Ack()

// ChanSubscribe
msgCh := make(chan *nats.Msg, 8192)
sub, _ = js.ChanSubscribe("foo", msgCh)

select {
case msg := <-msgCh:
    fmt.Println("[Received]", msg)
case <-time.After(1 * time.Second):
}
}

```

```

package io.nats.examples.jetstream;

import io.nats.client.*;
import io.nats.client.api.PublishAck;
import io.nats.examples.ExampleArgs;
import io.nats.examples.ExampleUtils;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

import static io.nats.examples.jetstream.NatsJsUtils.createStreamExitWhenExists;

/**
 * This example will demonstrate JetStream push subscribing using a durable consumer and a
 * queue
 */
public class NatsJsPushSubQueueDurable {
    static final String usageString =
        "\nUsage: java -cp <classpath> NatsJsPushSubQueueDurable [-s server] [-strm stream] [-"
    + "sub subject] [-q queue] [-dur durable] [-mcnt msgCount] [-scnt subCount]"
        + "\n\nDefault Values:"
        + "\n  [-strm stream]    qdur-stream"
        + "\n  [-sub subject]    qdur-subject"
        + "\n  [-q queue]        qdur-queue"
        + "\n  [-dur durable]    qdur-durable"
        + "\n  [-mcnt msgCount] 100"
        + "\n  [-scnt subCount] 5"
        + "\n\nUse tls:// or opentls:// to require tls, via the Default SSLContext\n"
        + "\nSet the environment variable NATS_NKEY to use challenge response
authentication by setting a file containing your private key.\n"
        + "\nSet the environment variable NATS_CREDS to use JWT/NKey authentication by
setting a file containing your user creds.\n"
        + "\nUse the URL in the -s server parameter for user/pass/token
authentication.\n";

    public static void main(String[] args) {

```

```

ExampleArgs exArgs = ExampleArgs.builder("Push Subscribe, Durable Consumer, Queue",
args, usageString)
    .defaultStream("qdur-stream")
    .defaultSubject("qdur-subject")
    .defaultQueue("qdur-queue")
    .defaultDurable("qdur-durable")
    .defaultMsgCount(100)
    .defaultSubCount(5)
    .build();

try (Connection nc = Nats.connect(ExampleUtils.createExampleOptions(exArgs.server,
true))) {

    // Create a JetStreamManagement context.
    JetStreamManagement jsm = nc.jetStreamManagement();

    // Use the utility to create a stream stored in memory.
    createStreamExitWhenExists(jsm, exArgs.stream, exArgs.subject);

    // Create our JetStream context
    JetStream js = nc.jetStream();

    System.out.println();

    // Setup the subscribers
    // - the PushSubscribeOptions can be re-used since all the subscribers are the
same
    // - use a concurrent integer to track all the messages received
    // - have a list of subscribers and threads so I can track them
    PushSubscribeOptions pso =
PushSubscribeOptions.builder().durable(exArgs.durable).build();
    AtomicInteger allReceived = new AtomicInteger();
    List<JsQueueSubscriber> subscribers = new ArrayList<>();
    List<Thread> subThreads = new ArrayList<>();
    for (int id = 1; id <= exArgs.subCount; id++) {
        // setup the subscription
        JetStreamSubscription sub = js.subscribe(exArgs.subject, exArgs.queue, pso);
        // create and track the runnable
        JsQueueSubscriber qs = new JsQueueSubscriber(id, exArgs, js, sub,
allReceived);

```

```

        subscribers.add(qs);
        // create, track and start the thread
        Thread t = new Thread(qs);
        subThreads.add(t);
        t.start();
    }
    nc.flush(Duration.ofSeconds(1)); // flush outgoing communication with/to the
server

```

```

// create and start the publishing
Thread pubThread = new Thread(new JsPublisher(js, exArgs));
pubThread.start();

```

```

// wait for all threads to finish
pubThread.join();
for (Thread t : subThreads) {
    t.join();
}

```

```

// report
for (JsQueueSubscriber qs : subscribers) {
    qs.report();
}

```

```

System.out.println();

```

```

// delete the stream since we are done with it.
jrm.deleteStream(exArgs.stream);
}
catch (Exception e) {
    e.printStackTrace();
}
}

```

```

static class JsPublisher implements Runnable {
    JetStream js;
    ExampleArgs exArgs;

    public JsPublisher(JetStream js, ExampleArgs exArgs) {
        this.js = js;
    }
}

```

```

        this.exArgs = exArgs;
    }

    @Override
    public void run() {
        for (int x = 1; x <= exArgs.msgCount; x++) {
            try {
                PublishAck pa = js.publish(exArgs.subject, ("Data # " +
x).getBytes(StandardCharsets.US_ASCII));
            } catch (IOException | JetStreamApiException e) {
                // something pretty wrong here
                e.printStackTrace();
                System.exit(-1);
            }
        }
    }
}

```

```

static class JsQueueSubscriber implements Runnable {
    int id;
    int thisReceived;
    List<String> datas;

    ExampleArgs exArgs;
    JetStream js;
    JetStreamSubscription sub;
    AtomicInteger allReceived;

    public JsQueueSubscriber(int id, ExampleArgs exArgs, JetStream js,
JetStreamSubscription sub, AtomicInteger allReceived) {
        this.id = id;
        thisReceived = 0;
        datas = new ArrayList<>();
        this.exArgs = exArgs;
        this.js = js;
        this.sub = sub;
        this.allReceived = allReceived;
    }

    public void report() {

```

```

        System.out.printf("Sub # %d handled %d messages.\n", id, thisReceived);
    }

    @Override
    public void run() {
        while (allReceived.get() < exArgs.msgCount) {
            try {
                Message msg = sub.nextMessage(Duration.ofMillis(500));
                while (msg != null) {
                    thisReceived++;
                    allReceived.incrementAndGet();
                    String data = new String(msg.getData(), StandardCharsets.US_ASCII);
                    datas.add(data);
                    System.out.printf("QS # %d message # %d %s\n", id, thisReceived,
data);

                    msg.ack();

                    msg = sub.nextMessage(Duration.ofMillis(500));
                }
            } catch (InterruptedException e) {
                // just try again
            }
        }
        System.out.printf("QS # %d completed.\n", id);
    }
}
}

```

JavaScript

```

import { AckPolicy, connect } from "../../src/mod.ts";
import { nuid } from "../../nats-base-client/nuid.ts";

const nc = await connect();

// create a regular subscription - this is plain nats
const sub = nc.subscribe("my.messages", { max: 5 });
const done = (async () => {
    for await (const m of sub) {
        console.log(m.subject);
    }
})

```

```

        m.respond();
    }
})();

const jsm = await nc.jetstreamManager();
const stream = nuid.next();
const subj = nuid.next();
await jsm.streams.add({ name: stream, subjects: [`${subj}.>`] });

// create a consumer that delivers to the subscription
await jsm.consumers.add(stream, {
    ack_policy: AckPolicy.Explicit,
    deliver_subject: "my.messages",
});

// publish some old nats messages
nc.publish(`${subj}.A`);
nc.publish(`${subj}.B`);
nc.publish(`${subj}.C`);
nc.publish(`${subj}.D.A`);
nc.publish(`${subj}.F.A.B`);

await done;
await nc.close();

```

Python

```

import asyncio

import nats
from nats.errors import TimeoutError

async def main():
    nc = await nats.connect("localhost")

    # Create JetStream context.
    js = nc.jetstream()

    # Persist messages on 'foo's subject.

```

```

await js.add_stream(name="sample-stream", subjects=["foo"])

for i in range(0, 10):
    ack = await js.publish("foo", f"hello world: {i}".encode())
    print(ack)

# Create pull based consumer on 'foo'.
psub = await js.pull_subscribe("foo", "psub")

# Fetch and ack messages from consumer.
for i in range(0, 10):
    msgs = await psub.fetch(1)
    for msg in msgs:
        print(msg)

# Create single push based subscriber that is durable across restarts.
sub = await js.subscribe("foo", durable="myapp")
msg = await sub.next_msg()
await msg.ack()

# Create deliver group that will have load balanced messages.
async def qsub_a(msg):
    print("QSUB A:", msg)
    await msg.ack()

async def qsub_b(msg):
    print("QSUB B:", msg)
    await msg.ack()
await js.subscribe("foo", "workers", cb=qsub_a)
await js.subscribe("foo", "workers", cb=qsub_b)

for i in range(0, 10):
    ack = await js.publish("foo", f"hello world: {i}".encode())
    print("\t", ack)

await nc.close()

if __name__ == '__main__':
    asyncio.run(main())

```

C#

```
// NATS .NET doesn't publicly support push consumers and treats all consumers
// as just consumers. The mechanics of the consuming messages are abstracted
// away from the applications and are handled by the library.
```

C

```
#include "examples.h"

static const char *usage = ""\
"-gd          use global message delivery thread pool\n" \
"-sync        receive synchronously (default is asynchronous)\n" \
"-pull        use pull subscription\n" \
"-fc          enable flow control\n" \
"-count       number of expected messages\n";

static void
onMsg(natsConnection *nc, natsSubscription *sub, natsMsg *msg, void *closure)
{
    if (print)
        printf("Received msg: %s - %.*s\n",
               natsMsg_GetSubject(msg),
               natsMsg_GetDataLength(msg),
               natsMsg_GetData(msg));

    if (start == 0)
        start = nats_Now();

    // We should be using a mutex to protect those variables since
    // they are used from the subscription's delivery and the main
    // threads. For demo purposes, this is fine.
    if (++count == total)
        elapsed = nats_Now() - start;

    // Since this is auto-ack callback, we don't need to ack here.
    natsMsg_Destroy(msg);
}
```

```

static void
asyncCb(natsConnection *nc, natsSubscription *sub, natsStatus err, void *closure)
{
    printf("Async error: %u - %s\n", err, natsStatus_GetText(err));

    natsSubscription_GetDropped(sub, (int64_t*) &dropped);
}

int main(int argc, char **argv)
{
    natsConnection      *conn  = NULL;
    natsStatistics       *stats = NULL;
    natsOptions          *opts  = NULL;
    natsSubscription     *sub   = NULL;
    natsMsg              *msg   = NULL;
    jsCtx                *js    = NULL;
    jsErrCode            jerr   = 0;
    jsOptions            jsOpts;
    jsSubOptions         so;
    natsStatus           s;
    bool                 delStream = false;

    opts = parseArgs(argc, argv, usage);

    printf("Created %s subscription on '%s'.\n",
        (pull ? "pull" : (async ? "asynchronous" : "synchronous")), subj);

    s = natsOptions_SetErrorHandler(opts, asyncCb, NULL);

    if (s == NATS_OK)
        s = natsConnection_Connect(&conn, opts);

    if (s == NATS_OK)
        s = jsOptions_Init(&jsOpts);

    if (s == NATS_OK)
        s = jsSubOptions_Init(&so);
    if (s == NATS_OK)
    {

```

```

so.Stream = stream;
so.Consumer = durable;
if (flowctrl)
{
    so.Config.FlowControl = true;
    so.Config.Heartbeat = (int64_t)1E9;
}
}

if (s == NATS_OK)
    s = natsConnection_JetStream(&js, conn, &jsOpts);

if (s == NATS_OK)
{
    jsStreamInfo    *si = NULL;

    // First check if the stream already exists.
    s = js_GetStreamInfo(&si, js, stream, NULL, &jerr);
    if (s == NATS_NOT_FOUND)
    {
        jsStreamConfig  cfg;

        // Since we are the one creating this stream, we can delete at the end.
        delStream = true;

        // Initialize the configuration structure.
        jsStreamConfig_Init(&cfg);
        cfg.Name = stream;
        // Set the subject
        cfg.Subjects = (const char*[1]){subj};
        cfg.SubjectsLen = 1;
        // Make it a memory stream.
        cfg.Storage = js_MemoryStorage;
        // Add the stream.
        s = js_AddStream(&si, js, &cfg, NULL, &jerr);
    }
    if (s == NATS_OK)
    {
        printf("Stream %s has %" PRIu64 " messages (%" PRIu64 " bytes)\n",

```

```

        si->Config->Name, si->State.Msgs, si->State.Bytes);

    // Need to destroy the returned stream object.
    jsStreamInfo_Destroy(si);
}

}

if (s == NATS_OK)
{
    if (pull)
        s = js_PullSubscribe(&sub, js, subj, durable, &jsOpts, &so, &jerr);
    else if (async)
        s = js_Subscribe(&sub, js, subj, onMsg, NULL, &jsOpts, &so, &jerr);
    else
        s = js_SubscribeSync(&sub, js, subj, &jsOpts, &so, &jerr);
}

if (s == NATS_OK)
    s = natsSubscription_SetPendingLimits(sub, -1, -1);

if (s == NATS_OK)
    s = natsStatistics_Create(&stats);

if ((s == NATS_OK) && pull)
{
    natsMsgList list;
    int i;

    for (count = 0; (s == NATS_OK) && (count < total); )
    {
        s = natsSubscription_Fetch(&list, sub, 1024, 5000, &jerr);
        if (s != NATS_OK)
            break;

        if (start == 0)
            start = nats_Now();

        count += (int64_t) list.Count;
        for (i=0; (s == NATS_OK) && (i<list.Count); i++)
            s = natsMsg_Ack(list.Msgs[i], &jsOpts);
    }
}

```

```

        natsMsgList_Destroy(&list);
    }
}
else if ((s == NATS_OK) && async)
{
    while (s == NATS_OK)
    {
        if (count + dropped == total)
            break;

        nats_Sleep(1000);
    }
}
else if (s == NATS_OK)
{
    for (count = 0; (s == NATS_OK) && (count < total); count++)
    {
        s = natsSubscription_NextMsg(&msg, sub, 5000);
        if (s != NATS_OK)
            break;

        if (start == 0)
            start = nats_Now();

        s = natsMsg_Ack(msg, &jsOpts);
        natsMsg_Destroy(msg);
    }
}

if (s == NATS_OK)
{
    printStats(STATS_IN|STATS_COUNT, conn, sub, stats);
    printPerf("Received");
}
if (s == NATS_OK)
{
    jsStreamInfo *si = NULL;

```

```

// Let's report some stats after the run
s = js_GetStreamInfo(&si, js, stream, NULL, &jerr);
if (s == NATS_OK)
{
    printf("\nStream %s has %" PRIu64 " messages (%" PRIu64 " bytes)\n",
        si->Config->Name, si->State.Msgs, si->State.Bytes);

    jsStreamInfo_Destroy(si);
}
if (delStream)
{
    printf("\nDeleting stream %s: ", stream);
    s = js_DeleteStream(js, stream, NULL, &jerr);
    if (s == NATS_OK)
        printf("OK!");
    printf("\n");
}
}
else
{
    printf("Error: %u - %s - jerr=%u\n", s, natsStatus_GetText(s), jerr);
    nats_PrintLastErrorStack(stderr);
}

// Destroy all our objects to avoid report of memory leak
jsCtx_Destroy(js);
natsStatistics_Destroy(stats);
natsSubscription_Destroy(sub);
natsConnection_Destroy(conn);
natsOptions_Destroy(opts);

// To silence reports of memory still in used with valgrind
nats_Close();

return 0;
}

```

Ordered Consumers

Ordered consumers are a convenient form of ephemeral push consumer for applications that want to efficiently consume a stream for data inspection or analysis.

The API consumer is guaranteed delivery of messages in sequence and without gaps.

- Always ephemeral - minimal overhead for the server
- Single threaded in sequence dispatching
- Client checks message sequence and will prevent gaps in the delivery
- Can recover from server node failure and reconnect
- Does not recover from client failure as it is ephemeral

Go

```
func ExampleJetStream() {
    nc, err := nats.Connect("localhost")
    if err != nil {
        log.Fatal(err)
    }

    // Use the JetStream context to produce and consume messages
    // that have been persisted.
    js, err := nc.JetStream(nats.PublishAsyncMaxPending(256))
    if err != nil {
        log.Fatal(err)
    }

    js.AddStream(&nats.StreamConfig{
        Name:      "F00",
        Subjects: []string{"foo"},
    })

    js.Publish("foo", []byte("Hello JS!"))

    // ordered push consumer
    js.Subscribe("foo", func(msg *nats.Msg) {
        meta, _ := msg.Metadata()
        fmt.Printf("Stream Sequence  : %v\n", meta.Sequence.Stream)
        fmt.Printf("Consumer Sequence: %v\n", meta.Sequence.Consumer)
    }, nats.OrderedConsumer())
}
```

Java

```

package io.nats.examples.jetstream;

import io.nats.client.*;
import io.nats.client.api.PublishAck;
import io.nats.client.impl.NatsMessage;
import io.nats.examples.ExampleArgs;
import io.nats.examples.ExampleUtils;

import java.nio.charset.StandardCharsets;
import java.time.Duration;
import java.time.temporal.TemporalUnit;

public class myExample {
    public static void main(String[] args) {
        final String subject = "foo";

        try (Connection nc = Nats.connect(ExampleUtils.createExampleOptions("localhost"))) {

            // Create a JetStream context. This hangs off the original connection
            // allowing us to produce data to streams and consume data from
            // JetStream consumers.
            JetStream js = nc.jetStream();

            // This example assumes there is a stream already created on subject "foo" and some
            messages already stored in that stream

            // create our message handler.
            MessageHandler handler = msg -> {

                System.out.println("\nMessage Received:");

                if (msg.hasHeaders()) {
                    System.out.println("  Headers:");
                    for (String key : msg.getHeaders().keySet()) {
                        for (String value : msg.getHeaders().get(key)) {
                            System.out.printf("    %s: %s\n", key, value);
                        }
                    }
                }
            }
        }
    }
}

```

```

    System.out.printf("  Subject: %s\n  Data: %s\n",
        msg.getSubject(), new String(msg.getData(), StandardCharsets.UTF_8));
    System.out.println("    " + msg.metaData());
};

Dispatcher dispatcher = nc.createDispatcher();
PushSubscribeOptions pso = PushSubscribeOptions.builder().ordered(true).build();
JetStreamSubscription sub = js.subscribe(subject, dispatcher, handler, false, pso);

Thread.sleep(100);

sub.drain(Duration.ofMillis(100));

nc.drain(Duration.ofMillis(100));
}
catch(Exception e)
{
    e.printStackTrace();
}
}
}

```

JavaScript

```

import { connect, consumerOpts } from "../../src/mod.ts";

const nc = await connect();
const js = nc.jetstream();

// note the consumer is not durable - so after the
// subscription ends, the server will auto destroy the
// consumer
const opts = consumerOpts();
opts.manualAck();
opts.maxMessages(2);
opts.deliverTo("xxx");
const sub = await js.subscribe("a.>", opts);
await (async () => {
    for await (const m of sub) {
        console.log(m.seq, m.subject);
    }
})();

```

```
        m.ack();
    }
}());

await nc.close();
```

Python

```
import asyncio

import nats
from nats.errors import TimeoutError

async def main():
    nc = await nats.connect("localhost")

    # Create JetStream context.
    js = nc.jetstream()

    # Create ordered consumer with flow control and heartbeats
    # that auto resumes on failures.
    osub = await js.subscribe("foo", ordered_consumer=True)
    data = bytearray()

    while True:
        try:
            msg = await osub.next_msg()
            data.extend(msg.data)
        except TimeoutError:
            break
    print("All data in stream:", len(data))

    await nc.close()

if __name__ == '__main__':
    asyncio.run(main())
```

C#

```
// dotnet add package NATS.Net
using NATS.Net;
using NATS.Client.JetStream;
using NATS.Client.JetStream.Models;

await using var client = new NatsClient();

INatsJSContext js = client.CreateJetStreamContext();

var streamConfig = new StreamConfig(name: "F00", subjects: ["foo"]);
await js.CreateStreamAsync(streamConfig);

PubAckResponse ack = await js.PublishAsync("foo", "Hello, JetStream!");
ack.EnsureSuccess();

INatsJSConsumer orderedConsumer = await js.CreateOrderedConsumerAsync("F00");

using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));

await foreach (NatsJSMsg<string> msg in
orderedConsumer.ConsumeAsync<string>(cancellation_token: cts.Token))
{
    NatsJSMsgMetadata? meta = msg.Metadata;
    Console.WriteLine($"Stream Sequence : {meta?.Sequence.Stream}");
    Console.WriteLine($"Consumer Sequence: {meta?.Sequence.Consumer}");
}
```

Delivery reliability

JetStream consumers can ensure not just the reliability of message delivery but also the reliability of the processing of the messages, even in the face of client application or downstream failures. It does so by using message level acknowledgements and message re-deliveries.

Consumers have an [Acknowledgement Policy](#) specifying the level of reliability required. In increasing order of reliability the available policies are: 'none' for no application level acknowledgements, 'all' where acknowledging a specific message also implicitly acknowledges all previous messages in the stream, and 'explicit' where each message must be individually acknowledged.

When the consumer is set to require explicit acknowledgements, client applications are able to use more than one kind of [acknowledgement](#) to indicate successful (or not) reception and processing of the messages being received from the consumer.

Applications can:

- Acknowledge the successful processing of a message (`Ack()`).
- Acknowledge the successful processing of a message and request an acknowledgement of the reception of the acknowledgement by the consumer (`AckSync()`).
- Indicate that the processing is still in progress and more time is needed (`inProgress()`).
- Negatively acknowledge a message, indicating that the client application is currently (temporarily) unable to process the message and that the consumer should attempt to re-deliver it (`Nak()`).
- Terminate a message (typically, because there is a problem with the data inside the message such that the client application is never going to be able to process it), indicating that the consumer should not attempt to re-deliver the message (`Term()`).

After a message is sent from the consumer to a subscribing client application by the server, an 'AckWait' timer is started. This timer is deleted when either a positive (`Ack()`) or a termination (`Term()`) acknowledgement is received from the client application. The timer gets reset upon reception of an in-progress (`inProgress()`) acknowledgement.

If at the end of a period of time no acknowledgement has been received from the client application, the server will attempt to re-deliver the message. If there is more than one client application instance subscribing to the consumer, there is no guarantee that the re-delivery would be to any particular client instance.

You can control the timing of re-deliveries using either the single `AckWait` duration attribute of the consumer, or as a sequence of durations in the `BackOff` attribute (which overrides `AckWait`).

You can also control the timing of re-deliveries when messages are negatively acknowledged with `Nak()`, by passing a `nakDelay()` option (or using `NakWithDelay()`), otherwise the re-delivery attempt will happen right after the reception of the `Nak()` by the server.

"Dead Letter Queues" type functionality

You can set a maximum number of delivery attempts using the consumer's `MaxDeliver` setting.

Whenever a message reaches its maximum number of delivery attempts, an advisory message is published on the `$JS.EVENT.ADVISORY.CONSUMER.MAX_DELIVERIES.<STREAM>.<CONSUMER>` subject. The advisory message's payload (use `nats schema info io.nats.jetstream.advisory.v1.max_deliver` for specific information) contains a `stream_seq` field that contains the sequence number of the message in the stream.

Similarly, whenever a client application terminates delivery attempts for the message using `AckTerm`, an advisory message is published on the `$JS.EVENT.ADVISORY.CONSUMER.MSG_TERMINATED.<STREAM>.<CONSUMER>` subject, and its payload (see `nats schema info io.nats.jetstream.advisory.v1.terminated`) contains a `stream_seq` field.

You can leverage those advisory messages to implement "Dead Letter Queue" (DLQ) types of functionalities. For example:

- If you only need to know about each time a message is 'dead' (considered un-re-deliverable by the consumer), then listening to the advisories is enough.
- If you also need to have access to the message in question then you can use the message's sequence number included in the advisory to retrieve that specific message by sequence number from the stream. If a message reaches its maximum level of delivery attempts, it will still stay in the stream until it is manually deleted or manually acknowledged.

```
##### #1
##### cocobook-sync ##### 2026-10-28 11:13:01 UTC
##### cocobook-sync ##### 2026-10-28 11:13:01 UTC
```