

Consumers



- [Dispatch type - Pull / Push](#)
 - [Ordered Consumers](#)
- [Persistence - Durable / Ephemeral](#)
- [Configuration](#)
 - [General](#)
 - [AckPolicy](#)
 - [DeliverPolicy](#)
 - [MaxAckPending](#)
 - [FilterSubjects](#)
 - [Pull-specific](#)
 - [Push-specific](#)

A consumer is a stateful **view** of a stream. It acts as an interface for clients to *consume* a subset of messages stored in a stream and will keep track of which messages were delivered and acknowledged by clients.

Unlike [Core NATS](#), which provides an at most once delivery guarantee, a consumer in JetStream can provide an at least **once delivery** guarantee.

While **Streams** are responsible for storing the published messages, the consumer is responsible for tracking the delivery and acknowledgments. This tracking ensures that if a message is not acknowledged (un-acked or 'nacked'), the consumer will automatically attempt to re-deliver it. JetStream consumers support various acknowledgment types and policies. If a message is not acknowledged within a user-specified number of delivery attempts, an advisory notification is emitted.

Dispatch type - Pull / Push

Consumers can be **push**-based where messages will be delivered to a specified subject or **pull**-based which allows clients to request batches of messages on demand. The choice of what kind of consumer to use depends on the use-case.

If there is a need to process messages in an application controlled manner and easily scale horizontally, you would use a 'pull consumer'. A simple client application that wants a replay of messages from a stream sequentially you would use an 'ordered push consumer'. An application that wants to benefit from load balancing or acknowledge messages individually will use a regular push consumer.

“ ” We recommend pull consumers for new projects. In particular when scalability, detailed flow control or error handling are a concern.

Ordered Consumers

Ordered consumers are the convenient default type of push & pull consumers designed for applications that want to efficiently consume a stream for data inspection or analysis.

- Always ephemeral
- No acknowledgements (if gap is detected, consumer is recreated)
- Automatic flow control/pull processing
- Single-threaded dispatching
- No load balancing

Persistence - Durable / Ephemeral

In addition to the choice of being push or pull, a consumer can also be **ephemeral** or **durable**. A consumer is considered *durable* when an explicit name is set on the `Durable` field when creating the consumer; otherwise it is ephemeral. Setting `InactiveThreshold` does **not** make a consumer durable. `InactiveThreshold` only controls automatic cleanup of an inactive consumer and applies to both durable and ephemeral consumers (prior to 2.9 it applied to ephemeral consumers only). Note also that the `Name` field is distinct from `Durable`: setting `Name` without `Durable` creates a named consumer that the server still treats as ephemeral.

Durables and ephemeral have the same message delivery semantics but an ephemeral consumer will not have persisted state or fault tolerance (server memory only) and will be automatically *cleaned up* (deleted) after a period of inactivity, when no subscriptions are bound to the consumer.

By default, consumers will have the same replication factor as the stream they consume, and will remain even when there are periods of inactivity (unless `InactiveThreshold` is set explicitly). Consumers can recover from server and client failure.

<https://youtu.be/334XuMma1fk> NATS JS Consumers - The ONE feature that makes NATS more powerful than Kafka, Pulsar, RabbitMQ, & redis

Configuration

Below are the set of consumer configuration options that can be defined. The `Version` column indicates the version of nats-server in which the option was introduced. The `Editable` column indicates the option can be edited after the consumer is created.

General

Field	Description	Version	Editable
Durable	If set, clients can have subscriptions bind to the consumer and <i>resume</i> until the consumer is explicitly deleted. A durable name cannot contain whitespace, <code>.</code> , <code>*</code> , <code>></code> , path separators (forward or backward slash), or non-printable characters.	2.2.0	No
FilterSubject	A subject that overlaps with the subjects bound to the stream to filter delivery to subscribers. Note: This cannot be used with the <code>FilterSubjects</code> field.	2.2.0	Yes
AckPolicy	The requirement of client acknowledgments, either <code>AckExplicit</code> , <code>AckNone</code> , <code>AckAll</code> , or <code>AckFlowControl</code> (2.14).	2.2.0	No
AckWait	The duration that the server will wait for an acknowledgment for any individual message <i>once it has been delivered to a consumer</i> . If an acknowledgment is not received in time, the message will be redelivered. This setting is only effective when <code>BackOff</code> is not configured .	2.2.0	Yes

Field	Description	Version	Editable
DeliverPolicy	The point in the stream from which to receive messages: <code>DeliverAll</code> , <code>DeliverLast</code> , <code>DeliverNew</code> , <code>DeliverByStartSequence</code> , <code>DeliverByStartTime</code> , or <code>DeliverLastPerSubject</code> .	2.2.0	No
OptStartSeq	Used with the <code>DeliverByStartSequence</code> deliver policy.	2.2.0	No
OptStartTime	Used with the <code>DeliverByStartTime</code> deliver policy.	2.2.0	No
Description	A description of the consumer. This can be particularly useful for ephemeral consumers to indicate their purpose since a durable name cannot be provided.	2.3.3	Yes
InactiveThreshold	Duration that instructs the server to clean up consumers inactive for that long. Prior to 2.9, this only applied to ephemeral consumers.	2.2.0	Yes
MaxAckPending	Defines the maximum number of messages, without acknowledgment, that can be outstanding. Once this limit is reached, message delivery will be suspended. This limit applies across <i>all</i> of the consumer's bound subscriptions. A value of -1 means there can be any number of pending acknowledgments (i.e., no flow control). The default is 1000.	2.2.0	Yes

Field	Description	Version	Editable
MaxDeliver	The maximum number of times a specific message delivery will be attempted. Applies to any message that is re-sent due to acknowledgment policy (i.e., due to a negative acknowledgment or no acknowledgment sent by the client). The default is -1 (redeliver until acknowledged). Messages that have reached the maximum delivery count will stay in the stream.	2.2.0	Yes
Backoff	A sequence of delays controlling the re-delivery of messages on acknowledgment timeout (but not on <code>nak</code>). The sequence length must be less than or equal to <code>MaxDeliver</code> . If backoff is not set, a timeout will result in immediate re-delivery. E.g., <code>MaxDeliver=5</code> <code>backoff=[5s, 30s, 300s, 3600s, 84000s]</code> will re-deliver a message 5 times over one day. When <code>MaxDeliver</code> is larger than the backoff list, the last delay in the list will apply for the remaining deliveries. Note that backoff is NOT applied to <code>nak</code> ed messages. A <code>nak</code> will result in immediate re-delivery unless <code>nakWithDelay</code> is used to set the re-delivery delay explicitly. When <code>BackOff</code> is set, it overrides <code>AckWait</code> entirely . The first value in the BackOff determines the <code>AckWait</code> value.	2.7.1	Yes

Field	Description	Version	Editable
ReplayPolicy	If the policy is <code>ReplayOriginal</code> , the messages in the stream will be pushed to the client at the same rate they were originally received, simulating the original timing. If the policy is <code>ReplayInstant</code> (default), the messages will be pushed to the client as fast as possible while adhering to the acknowledgment policy, Max Ack Pending, and the client's ability to consume those messages.	2.2.0	No
Replicas	Sets the number of replicas for the consumer's state. By default, when the value is set to zero, consumers inherit the number of replicas from the stream.	2.8.3	Yes
MemoryStorage	If set, forces the consumer state to be kept in memory rather than inherit the storage type of the stream (default is file storage). This reduces I/O from acknowledgments, useful for ephemeral consumers.	2.8.3	No
SampleFrequency	Sets the percentage of acknowledgments that should be sampled for observability, 0-100. This value is a string and allows both <code>30</code> and <code>30%</code> as valid values.	2.2.0	Yes
Metadata	A set of application-defined key-value pairs for associating metadata with the consumer.	2.10.0	Yes
FilterSubjects	A set of subjects that overlap with the subjects bound to the stream to filter delivery to subscribers. Note: This cannot be used with the <code>FilterSubject</code> field.	2.10.0	Yes

Field	Description	Version	Editable
HeadersOnly	Delivers only the headers of messages in the stream, adding a <code>Nats-Msg-Size</code> header indicating the size of the removed payload.	2.6.2	Yes

AckPolicy

The policy choices include:

- `AckExplicit`: The default policy. Each individual message must be acknowledged. Recommended for most reliability and functionality.
- `AckNone`: No acknowledgment needed; the server assumes acknowledgment on delivery.
- `AckAll`: Acknowledge only the last message received in a series; all previous messages are automatically acknowledged. Will acknowledge all pending messages for all subscribers for Pull Consumer.
- `AckFlowControl`: (2.14) Messages are acknowledged based on flow control. Primarily used for stream sourcing/mirroring using a durable consumer.

If an acknowledgment is required but not received within the `AckWait` window, the message will be redelivered.

⚠ **Warning:** The server may consider an acknowledgment arriving out of the window. For instance, in a queue situation, if a first process fails to acknowledge within the window and the message has been redelivered to another consumer, the acknowledgment from the first consumer will be considered.

DeliverPolicy

The policy choices include:

- `DeliverAll`: Default policy. Start receiving from the earliest available message in the stream.
- `DeliverLast`: Start with the last message added to the stream, or the last message matching the consumer's filter subject if defined.
- `DeliverLastPerSubject`: Start with the latest message for each filtered subject currently in the stream.
- `DeliverNew`: Start receiving messages created after the consumer was created.
- `DeliverByStartSequence`: Start at the first message with the specified sequence number. The consumer must specify `OptStartSeq` defining the sequence number.
- `DeliverByStartTime`: Start with messages on or after the specified time. The consumer must specify `OptStartTime` defining the start time.

MaxAckPending

The `MaxAckPending` capability provides flow control and applies to both push and pull consumers. For push consumers, `MaxAckPending` is the only form of flow control. For pull consumers, client-driven message delivery creates implicit one-to-one flow control with subscribers.

For high throughput, set `MaxAckPending` to a high value. For applications with high latency due to external services, use a lower value and adjust `AckWait` to avoid re-deliveries.

FilterSubjects

A filter subject provides server-side filtering of messages before delivery to clients.

For example, a stream `factory-events` with subject `factory-events.*.*` can have a consumer `factory-A` with a filter `factory-events.A.*` to deliver only events for factory `A`.

A consumer can have a singular `FilterSubject` or plural `FilterSubjects`. Multiple filters can be applied, such as `[factory-events.A.*, factory-events.B.*]` or specific event types `[factory-events.*.item_produced, factory-events.*.item_packaged]`.

“ **Warning:** For granular consumer permissions, a single filter uses `$JS.API.CONSUMER.CREATE.{stream}.{consumer}.{filter}` to restrict users to specific filters. Multiple filters use the general `$JS.API.CONSUMER.DURABLE.CREATE.{stream}.{consumer}`, which does not include the `{filter}` token. Use a different strategy for granular permissions.

Pull-specific

These options apply only to pull consumers. For configuration examples, see [NATS by Example](#).

Field	Description	Version	Editable
MaxWaiting	The maximum number of waiting pull requests.	2.2.0	No
MaxRequestExpires	The maximum duration a single pull request will wait for messages to be available to pull.	2.7.0	Yes

Field	Description	Version	Editable
MaxRequestBatch	The maximum batch size a single pull request can make. When set with <code>MaxRequestMaxBytes</code> , the batch size will be constrained by whichever limit is hit first.	2.7.0	Yes
MaxRequestMaxBytes	The maximum total bytes that can be requested in a given batch. When set with <code>MaxRequestBatch</code> , the batch size will be constrained by whichever limit is hit first.	2.8.3	Yes

Push-specific

These options apply only to push consumers. For configuration examples, see [NATS by Example](#).

Field	Description	Version	Editable
DeliverSubject	The subject to deliver messages to. Setting this field decides whether the consumer is push or pull-based. With a deliver subject, the server will <i>push</i> messages to clients subscribed to this subject.	2.2.0	No
DeliverGroup	The queue group name used to distribute messages among subscribers. Analogous to a queue group in core NATS.	2.2.0	Yes
FlowControl	Enables per-subscription flow control using a sliding-window protocol. This protocol relies on the server and client exchanging messages to regulate when and how many messages are pushed to the client. This one-to-one flow control mechanism works in tandem with the one-to-many flow control imposed by <code>MaxAckPending</code> across all subscriptions bound to a consumer.	2.2.0	Yes

Field	Description	Version	Editable
IdleHeartbeat	If set, the server will regularly send a status message to the client during inactivity, indicating that the JetStream service is up and running. The status message will have a code of 100 and no reply address. Note: This mechanism is handled transparently by supported clients.	2.2.0	Yes
RateLimit	Throttles the delivery of messages to the consumer, in bits per second.	2.2.0	Yes

- #2
- cocobook-sync

2026-10-28 11:10:24 UTC
- cocobook-sync

2026-10-28 11:15:49 UTC