

JetStream



- [Additional capabilities enabled by JetStream](#)
 - [Configuration](#)
 - [Examples](#)
 - [Goals](#)
 - [JetStream capabilities](#)
 - [Streaming: temporal decoupling between the publishers and subscribers](#)
 - [Persistent and Consistent distributed storage](#)
 - [De-coupled flow control](#)
 - [Exactly once semantics](#)
 - [Consumers](#)
 - [Key Value Store](#)
 - [Watch and History](#)
 - [Atomic updates and locking](#)
 - [Object Store](#)
 - [Legacy](#)
-

NATS has a built-in persistence engine called [JetStream](#) which enables messages to be stored and replayed at a later time. Unlike *NATS Core* which requires you to have an active subscription to process messages as they happen, JetStream allows the NATS server to capture messages and replay them to consumers as needed. This functionality enables a different quality of service for your NATS messages, and enables fault-tolerant and high-availability configurations.

JetStream is built into `nats-server`. If you have a cluster of JetStream-enabled servers you can enable data replication and thus guard against failures and service disruptions.

JetStream was created to address the problems identified with streaming technology today - complexity, fragility, and a lack of scalability. Some technologies address these better than others, but no current streaming technology is truly multi-tenant, horizontally scalable, or supports multiple deployment models. No other technology that we are aware of can scale from edge to cloud using the same security context while having complete deployment observability for operations.

Additional capabilities enabled by JetStream

The JetStream persistence layer enables additional use cases typically not found in messaging systems. Being built on top of JetStream they inherit the core capabilities of JetStream, replication, security, routing limits, and mirroring.

- [Key Value Store](#) A map (associative array) with atomic operations
- [Object Store](#) File transfer, replications and storage API. Uses chunked transfers for scalability.

Key/Value and File transfer are capabilities commonly found in in-memory databases or deployment tools. While NATS does not intend to compete with the feature set of such tools, it is our goal to provide the developer with reasonable complete set of data storage and replications features for use cases like micro service, edge deployments and server management.

Configuration

To configure a `nats-server` with JetStream refer to:

- [Configuring JetStream](#)
- [JetStream Clustering](#)

Examples

For runnable JetStream code examples, refer to [NATS by Example](#).

Goals

JetStream was developed with the following goals in mind:

- The system must be easy to configure and operate and be observable.
- The system must be secure and operate well with NATS 2.0 security models.
- The system must scale horizontally and be applicable to a high ingestion rate.
- The system must support multiple use cases.
- The system must self-heal.
- The system must allow NATS messages to be part of a stream as desired.
- The system must display payload agnostic behavior.
- The system must not have third party dependencies.

JetStream capabilities

Streaming: temporal decoupling between the publishers and subscribers

One of the tenets of basic publish/subscribe messaging is that there is a required temporal coupling between the publishers and the subscribers: subscribers only receive the messages that are published when they are actively connected to the messaging system (i.e. they do not receive messages that are published while they are not subscribing or not running or disconnected). The traditional way for messaging systems to provide temporal decoupling of the publishers and subscribers is through the 'durable subscriber' functionality or sometimes through 'queues', but neither one is perfect:

- durable subscribers need to be created *before* the messages get published
- queues are meant for workload distribution and consumption, not to be used as a mechanism for message replay.

However, in many use cases, you do not need to 'consume exactly once' functionality but rather the ability to replay messages on demand, as many times as you want. This need has led to the popularity of some 'streaming' messaging platforms.

JetStream provides *both* the ability to *consume* messages as they are published (i.e. 'queueing') as well as the ability to *replay* messages on demand (i.e. 'streaming'). See [retention policies](#) below.

Replay policies

JetStream consumers support multiple replay policies, depending on whether the consuming application wants to receive either:

- *all* of the messages currently stored in the stream, meaning a complete 'replay' and you can select the 'replay policy' (i.e. the speed of the replay) to be either:
 - *instant* (meaning the messages are delivered to the consumer as fast as it can take them).
 - *original* (meaning the messages are delivered to the consumer at the rate they were published into the stream, which can be very useful for example for staging production traffic).
- the *last* message stored in the stream, or the *last message for each subject* (as streams can capture more than one subject).
- starting from a specific *sequence number*.
- starting from a specific *start time*.

Retention policies and limits

JetStream enables new functionalities and higher qualities of service on top of the base 'Core NATS' functionality. However, practically speaking, streams can't always just keep growing 'forever' and therefore JetStream supports multiple retention policies as well as the ability to impose size limits on streams.

Limits

You can impose the following limits on a stream

- Maximum message age.
- Maximum total stream size (in bytes).
- Maximum number of messages in the stream.
- Maximum individual message size.
- You can also set limits on the number of consumers that can be defined for the stream at any given point in time.

You must also select a **discard policy** which specifies what should happen once the stream has reached one of its limits and a new message is published:

- *discard old* means that the stream will automatically delete the oldest message in the stream to make room for the new messages.
- *discard new* means that the new message is discarded (and the JetStream publish call returns an error indicating that a limit was reached).

Retention policy

You can choose what kind of retention you want for each stream:

- *limits* (the default) is to provide a replay of messages in the stream.
- *work queue* (the stream is used as a shared queue and messages are removed from it as they are consumed) is to provide the exactly-once consumption of messages in the stream.
- *interest* (messages are kept in the stream for as long as there are consumers that haven't delivered the message yet) is a variation of work queue that only retains messages if there is interest (consumers currently defined on the stream) for the message's subject.

Note that regardless of the retention policy selected, the limits (and the discard policy) *always* apply.

Subject mapping transformations

JetStream also enables the ability to apply subject mapping transformations to messages as they are ingested into a stream.

Persistent and Consistent distributed storage

You can choose the durability as well as the resilience of the message storage according to your needs.

- Memory storage.
- File storage.
- Replication (1 (none), 2, 3) between nats servers for Fault Tolerance.

JetStream uses a NATS optimized RAFT distributed quorum algorithm to distribute the persistence service between NATS servers in a cluster while maintaining immediate consistency (as opposed to [eventual consistency](#)) even in the face of failures.

For writes (publications to a stream), the formal consistency model of NATS JetStream is [Linearizable](#). On the read side (listening to or replaying messages from streams) the formal models don't really apply because JetStream does not support atomic batching of multiple operations together (so the only kind of 'transaction' is the persisting, replicating and voting of a single operation on the stream) but in essence, JetStream is [serializable](#) because messages are added to a stream in one global order (which you can control using `compare and publish`).

Do note, while we do guarantee immediate consistency when it comes to [monotonic writes](#) and [monotonic reads](#). We don't guarantee [read your writes](#) at this time, as reads through *direct get* requests may be served by followers or mirrors. More consistent results can be achieved by sending get requests to the stream leader.

JetStream can also provide encryption at rest of the messages being stored.

In JetStream the configuration for storing messages is defined separately from how they are consumed. Storage is defined in a [Stream](#) and consuming messages is defined by multiple [Consumers](#).

Stream replication factor

A stream's replication factor (R, often referred to as the number 'Replicas') determines how many places it is stored allowing you to tune to balance risk with resource usage and performance. A stream that is easily rebuilt or temporary might be memory-based with a R=1 and a stream that can tolerate some downtime might be file-based R-1.

Typical usage to operate in typical outages and balance performance would be a file-based stream with R=3. A highly resilient, but less performant and more expensive configuration is R=5, the replication factor limit.

Rather than defaulting to the maximum, we suggest selecting the best option based on the use case behind the stream. This optimizes resource usage to create a more resilient system at scale.

- Replicas=1 - Cannot operate during an outage of the server servicing the stream. Highly performant.
- Replicas=2 - No significant benefit at this time. We recommend using Replicas=3 instead.
- Replicas=3 - Can tolerate the loss of one server servicing the stream. An ideal balance between risk and performance.
- Replicas=4 - No significant benefit over Replicas=3 except marginally in a 5 node cluster.
- Replicas=5 - Can tolerate simultaneous loss of two servers servicing the stream. Mitigates risk at the expense of performance.

Mirroring and Sourcing between streams

JetStream also allows server administrators to easily mirror streams, for example between different JetStream domains in order to offer disaster recovery. You can also define a stream that 'sources' from one or more other streams.

Syncing data to disk

JetStream's file-based streams persist messages to disk. However, while JetStream does flush file writes to the OS synchronously, under the default configuration it does not immediately `fsync` data to disk. The server uses a configurable `sync_interval` option, with a default value of 2 minutes, which controls how often the server will `fsync` its data. The data will be `fsync`-ed no later than this interval. This has important consequences for durability with respect to OS failures (meaning ungraceful exit of the Operating System such as a power outage, and not just ungraceful exit or killing of the `nats-server` process itself):

In a non-replicated setup, an OS failure may result in data loss. A client might publish a message and receive an acknowledgment, but the data may not yet be safely stored to disk. As a result, after an OS failure recovery, a server may have lost recently acknowledged messages.

In a replicated setup, a published message is acknowledged after it successfully replicated to at least a quorum of servers. However, replication alone is not enough to guarantee the strongest level of durability against multiple systemic failures.

- If multiple servers fail simultaneously, all due to an OS failure, and before their data has been `fsync`-ed, the cluster may fail to recover the most recently acknowledged messages.
- If a failed server lost data locally due to an OS failure, although extremely rare, there are some combinations of events where it may rejoin the cluster and form a new majority with nodes that have never received or persisted a given message. The cluster may then proceed with incomplete data causing acknowledged messages to be lost.

Setting a lower `sync_interval` increases the frequency of disk writes, and reduces the window for potential data loss, but at the expense of performance. Additionally, setting `sync_interval: always` will make sure servers `fsync` after every message before it is acknowledged. This setting, combined with replication in different data centers or availability zones, provides the strongest durability guarantees but at the slowest performance.

The default settings have been chosen to balance performance and risk of data loss in what we consider to be a typical production deployment scenario across multiple availability zones.

For example, consider a stream with 3 replicas deployed across three separate availability zones. For the stream state to diverge across nodes would require that:

- One of the 3 servers is already offline, isolated or partitioned.
- A second server's OS needs to fail such that it loses writes of messages that were only available on 2 out of 3 nodes due to them not being `fsync`-ed.

- The stream leader that's part of the above 2 out of 3 nodes needs to go down or become isolated/partitioned.
- The first server of the original partition that didn't receive the writes recovers from the partition.
- The OS-failed server now returns and comes in contact with the first server but not with the previous stream leader.

In the end, 2 out of 3 nodes will be available, the previous stream leader with the writes will be unavailable, one server will have lost some writes due to the OS failure, and one server will have never seen these writes due to the earlier partition. The last two servers could then form a majority and accept new writes, essentially losing some of the former writes.

Importantly this is a failure condition where stream state could diverge, but in a system that is deployed across multiple availability zones, it would require multiple faults to align precisely in the right way.

A potential mitigation to a failure of this kind is not automatically bringing back a server process that was OS-failed until it is known that a majority of the remaining servers have received the new writes, or by peer-removing the crashed server and admitting it as a new and wiped peer and allowing it to recover over the network from existing healthy nodes (although this could be expensive depending on the amount of data involved).

For use cases where minimizing loss is an absolute priority, `sync_interval: always` can of course still be configured, but note that this will have a server-wide performance impact that may affect throughput or latencies. For production environments, operators should evaluate whether the default is correct for their use case, target environment, costs, and performance requirements.

Alternatively, a hybrid approach can be used where existing clusters still function under their default `sync_interval` settings but a new cluster gets added that's configured with `sync_interval: always`, and utilizes server tags. The placement of a stream can then be specified to have this stream store data on this higher durability cluster through the use of [placement tags](#).

```
# Configure a cluster that's dedicated to always sync writes.
server_tags: ["sync:always"]

jetstream {
    sync_interval: always
}
```

Create a replicated stream that's specifically placed in the cluster using `sync_interval: always`, to ensure the strongest durability only for stream writes that require this level of durability.

```
nats stream add --replicas 3 --tag sync:always
```

De-coupled flow control

JetStream provides decoupled flow control over streams, the flow control is not 'end to end' where the publisher(s) are limited to publish no faster than the slowest of all the consumers (i.e. the lowest common denominator) can receive but is instead happening individually between each client application (publishers or consumers) and the nats server.

When using the JetStream publish calls to publish to streams there is an acknowledgment mechanism between the publisher and the NATS server, and you have the choice of making synchronous or asynchronous (i.e. 'batched') JetStream publish calls.

On the subscriber side, the sending of messages from the NATS server to the client applications receiving or consuming messages from streams is also flow controlled.

Exactly once semantics

Because publications to streams using the JetStream publish calls are acknowledged by the server the base quality of service offered by streams is '*at least once*', meaning that while reliable and normally duplicate free there are some specific failure scenarios that could result in a publishing application believing (wrongly) that a message was not published successfully and therefore publishing it again, and there are failure scenarios that could result in a client application's consumption acknowledgment getting lost and therefore in the message being re-sent to the consumer by the server. Those failure scenarios while being rare and even difficult to reproduce do exist and can result in perceived 'message duplication' at the application level.

Therefore, JetStream also offers an '*exactly once*' quality of service. For the publishing side, it relies on the publishing application attaching a unique message or publication ID in a message header and on the server keeping track of those IDs for a configurable rolling period of time in order to detect the publisher publishing the same message twice. For the subscribers a *double* acknowledgment mechanism is used to avoid a message being erroneously re-sent to a subscriber by the server after some kinds of failures.

Consumers

JetStream [consumers](#) are 'views' on a stream, they are subscribed to (or pulled) by client applications to receive copies of (or to consume if the stream is set as a working queue) messages stored in the stream.

Fast push consumers

Client applications can choose to use fast un-acknowledged `push` (ordered) consumers to receive messages as fast as possible (for the selected replay policy) on a specified delivery subject or to an inbox. Those consumers are meant to be used to 'replay' rather than 'consume' the messages in a stream.

Horizontally scalable pull consumers with batching

Client applications can also use and share `pull` consumers that are demand-driven, support batching and must explicitly acknowledge message reception and processing which means that they can be used to consume (i.e. use the stream as a distributed queue) as well as process the messages in a stream.

Pull consumers can and are meant to be shared between applications (just like queue groups) in order to provide easy and transparent horizontal scalability of the processing or consumption of messages in a stream without having (for example) to worry about having to define partitions or worry about fault-tolerance.

Note: using pull consumers doesn't mean that you can't get updates (new messages published into the stream) 'pushed' in real-time to your application, as you can pass a (reasonable) timeout to the consumer's `Fetch` call and call it in a loop.

Consumer acknowledgments

While you can decide to use un-acknowledged consumers trading quality of service for the fastest possible delivery of messages, most processing is not idem-potent and requires higher qualities of service (such as the ability to automatically recover from various failure scenarios that could result in some messages not being processed or being processed more than once) and you will want to use acknowledged consumers. JetStream supports more than one kind of acknowledgment:

- Some consumers support acknowledging *all* the messages up to the sequence number of the message being acknowledged, some consumers provide the highest quality of service but require acknowledging the reception and processing of each message explicitly as well as the maximum amount of time the server will wait for an acknowledgment for a specific message before re-delivering it (to another process attached to the consumer).
- You can also send back *negative* acknowledgements.
- You can even send *in progress* acknowledgments (to indicate that you are still processing the message in question and need more time before acking or nacking it).

Key Value Store

The JetStream persistence layer enables the Key Value store: the ability to store, retrieve and delete `value` messages associated with a `key` into a `bucket`.

- [Concepts](#)
- [Walkthrough](#)
- [API and details](#)

Watch and History

You can subscribe to changes in a Key Value on the bucket or individual key level with `watch` and optionally retrieve a `history` of the values (and deletions) that have happened on a particular key.

Atomic updates and locking

The Key Value store supports atomic `create` and `update` operations. This enables pessimistic locks (by creating a key and holding on to it) and optimistic locks (using CAS - compare and set).

Object Store

The Object Store is similar to the Key Value Store. The key being replaced by a file name and value being designed to store arbitrarily large `objects` (e.g. files, even if they are very large) rather than 'values' that are message-sized (i.e. limited to 1Mb by default). This is achieved by chunking messages.

- [Concepts](#)
- [Walkthrough](#)
- [API and details](#)

Legacy

Note that JetStream completely replaces the [STAN](#) legacy NATS streaming layer.

📄 #3

📄 cocobook-sync 📄 2026-10-28 11:10:19 UTC

📄 cocobook-sync 📄 2026-10-29 01:42:33 UTC