

# Reference

- [FAQ](#)
- [NATS Protocols](#)
- [Protocol Demo](#)
- [Client Protocol](#)
- [Developing a Client](#)
- [NATS Cluster Protocol](#)
- [JetStream wire API Reference](#)
- [Roadmap](#)
- [Contributing](#)

# FAQ

## General

- [What is NATS?](#)
- [What language is NATS written in?](#)
- [Who maintains NATS?](#)
- [What clients does NATS support?](#)
- [What does the NATS acronym stand for?](#)
- [JetStream and NATS Streaming?](#)

## Technical Questions

- [What is the difference between Request\(\) and Publish\(\)?](#)
- [Can multiple subscribers receive a Request?](#)
- [How can I monitor my NATS cluster?](#)
- [Does NATS do queuing? Does NATS do load balancing?](#)
- [Can I list the subjects that exist in my NATS cluster?](#)
- [Does NATS support subject wildcards?](#)
- [What is the right kind of Stream consumer to use?](#)
- [What do 'verbose' and 'pedantic' mean when using CONNECT?](#)
- [Does NATS offer any guarantee of message ordering?](#)
- [Is there a message size limitation in NATS?](#)
- [Does NATS impose any limits on the # of subjects?](#)
- [Does NATS guarantee message delivery?](#)
- [Does NATS support replay/redelivery of historical data?](#)
- [How do I gracefully shut down an asynchronous subscriber?](#)
- [How do I create subjects?](#)
- [How many clients can connect simultaneously?](#)

# General

## What is NATS?

NATS is an open source, lightweight, high-performance cloud native infrastructure messaging system. It implements a highly scalable and elegant publish-subscribe (pub/sub) distribution model. The performant nature of NATS make it an ideal base for building modern, reliable, scalable cloud native distributed systems.

NATS is offered in two interoperable modules in a single "NATS Server" binary (often referred to as `nats-server` throughout this site):

- 'Core NATS' is the set of core NATS functionalities and qualities of service.
- '[JetStream](#)' is the (optionally enabled) built-in persistence layer that adds streaming, at-least-once and exactly-once guarantees, historical data replay, decoupled flow-control and key/value store functionalities to Core NATS.

NATS was created by Derek Collison, who has over 25 years of experience designing, building, and using publish-subscribe messaging systems. NATS is maintained by an amazing Open Source Ecosystem, find more at [GitHub](#).

## What does the NATS acronym stand for?

NATS stands for Neural Autonomic Transport System. Derek Collison conceived NATS as a messaging platform that functions like a central nervous system.

## JetStream and NATS Streaming?

As of NATS Server 2.2, NATS [JetStream](#) is the recommended option for persistence, streaming and higher message guarantees. [NATS Streaming](#) a.k.a. 'STAN' is now deprecated.

## What language is NATS written in?

The NATS server (`nats-server`) is written in Go. There is client support for a wide variety of languages. Please see the [Developing with NATS](#) page for more info.

# Who maintains NATS?

NATS is maintained by a select group of Maintainers following a Governance process as part of the [Cloud Native Computing Foundation \(CNCF\)](#). The team of engineers at [Synadia](#) in conjunction with Community Maintainers, maintain the NATS server, NATS Streaming Server, as well as the official Go, Ruby, Node.js, C, C#, Java and several other client libraries. Our very active user community also contributes client libraries and connectors for several other implementation languages. Please see the [download](#) page for the complete list, and links to the relevant source repositories and documentation.

# What client support exists for NATS?

Please see the [Developing with NATS](#) page for the latest list of Synadia and Community maintained NATS clients.

# Technical Questions

## What is the difference between Request() and Publish()?

Publish() sends a message to `nats-server` with a subject as its address, and `nats-server` delivers the message to any interested/eligible subscriber of that subject. Optionally, you may also send along a reply subject with your message, which provides a way for subscribers who have received your message(s) to send messages back to you.

Request() is simply a convenience API that does this for you in a pseudo-synchronous fashion, using a timeout supplied by you. It creates an INBOX (a type of subject that is unique to the requestor), subscribes to it, then publishes your request message with the reply address set to the inbox subject. It will then wait for a response, or the timeout period to elapse, whichever comes first.

## Can multiple subscribers receive a Request?

Yes. NATS is a publish and subscribe system that also has distributed queueing functionality on a per subscriber basis. When you publish a message, for instance at the beginning of a request, every subscriber will receive the message. If subscribers form a queue group, only one subscriber will be picked at random to receive the message. However, note that the requestor does not know or control this information. What the requestor does control is that it only wants one answer to the request, and NATS handles this very well by actively pruning the interest graph.

## How can I monitor my NATS cluster?

NATS can be deployed to have an HTTP(s) monitoring port - see the demo server here:

<https://demo.nats.io:8222/>. Alternately, there are several options available, including some from the active NATS community:

- [Prometheus NATS Exporter](#) Use Prometheus to configure metrics and Grafana to create a visual display.
- [nats-top](#) A top-like monitoring tool developed by Wally Quevedo of Synadia.
- [natsboard](#) A monitoring tool developed by Fatih Cetinkaya.
- [nats-mon](#) A monitoring tool developed by Raúl Pérez and Adrià Cidre.

A more detailed overview of monitoring is available under [NATS Server Monitoring](#).

## Does NATS do queueing? Does NATS do load balancing?

The term 'queueing' implies different things in different contexts, so we must be careful with its use. NATS implements non-persistent distributed queueing via subscriber queue groups. Subscriber queue groups offer a form of message-distribution load balancing. Subject subscriptions in NATS may be either 'individual' subscriptions or queue group subscriptions. The choice to join a queue group is made when the subscription is created, by supplying an optional queue group name. For individual subject subscribers, `nats-server` will attempt to deliver a copy of every message published to that subject to every eligible subscriber of that subject. For subscribers in a queue group, `nats-server` will attempt to deliver each successive message to exactly *one* subscriber in the group, chosen at random.

This form of distributed queueing is done in real time, and messages are not persisted to secondary storage. Further, the distribution is based on interest graphs (subscriptions), so it is not a publisher operation, but instead is controlled entirely by `nats-server`.

# Can I list the subjects that exist in my NATS cluster?

NATS maintains and constantly updates the interest graph (subjects and their subscribers) in real time. Do not think of it as a "directory" that is aggregated over time. The interest graph is dynamic, and will change constantly as publishers and subscribers come and go.

If you are determined to gather this information, it can be indirectly derived at any instant in time by polling the monitoring endpoint for `/connz` and `/routez`. See [Server Monitoring](#) for more information.

## Does NATS support subject wildcards?

Yes. The valid wildcards are as follows:

The dot character `'.'` is the token separator.

The asterisk character `'*'` is a token wildcard match.

```
e.g foo.* matches foo.bar, foo.baz, but not foo.bar.baz.
```

The greater-than symbol `'>'` is a full wildcard match.

```
e.g. foo.> matches foo.bar, foo.baz, foo.bar.baz, foo.bar.1, etc.
```

## What is the right kind of Stream consumer to use

It depends on the access pattern of the application using the stream: if you want to horizontally scale the processing of all the messages stored in a stream and/or process a high-throughput stream of messages in real-time using batching, then use a shared pull consumer (as they scale well horizontally and batching is in practice key to achieving high throughput). But if the access pattern is more like individual application instances needing their own individual replay of the messages in a stream on demand: then an 'ordered push consumer' is best. Consider the use of a durable push consumer with a queue-group for the clients if you want a scalable low latency real time processing of the messages inserted into a stream.

# What do 'verbose' and 'pedantic' mean when using CONNECT?

'Verbose' means all protocol commands will be acked with a +OK or -ERR. If verbose is off, you don't get the +OK for each command. Pedantic means the server does lots of extra checking, mostly around properly formed subjects, etc. Verbose mode is ON by default for new connections; most client implementations disable verbose mode by default in their INFO handshake during connection.

# Does NATS offer any guarantee of message ordering?

NATS implements source ordered delivery per publisher. That is to say, messages from a given single publisher will be delivered to all eligible subscribers in the order in which they were originally published. There are no guarantees of message delivery order amongst multiple publishers.

# Is there a message size limitation in NATS?

Messages have a maximum size (which is set in the server configuration with `max_payload`) that is enforced by the server and communicated to the client during connection setup. The size is set to 1 MB by default, but can be increased up to 64 MB if needed (though we recommend keeping the max message size to something more reasonable like 8 MB).

# Does NATS impose any limits on the # of subjects?

No. As of `nats-server` v0.8.0, there is no hard limit on the maximum number of subjects.

# Does NATS guarantee message delivery?

Core NATS, offers "at-most-once" delivery. This means messages are guaranteed to arrive intact, in order from a given publisher, but not across different publishers. NATS does everything required to remain available and provide a dial-tone. However, if a subscriber is problematic or goes offline it will not receive messages, as the basic NATS platform is a simple pub-sub transport system that

offers only TCP reliability.

As of NATS Server 2.2, NATS JetStream offers persistence with "at-least-once" and "exactly-once" (within a time window) delivery. See the [JetStream](#) documentation for detailed information.

## Does NATS support replay/redelivery of historical data?

NATS [JetStream](#) offers message store and replay by time or sequence.

## How do I gracefully shut down an asynchronous subscriber?

To gracefully shut down an asynchronous subscriber so that any outstanding MsgHandlers have a chance to complete outstanding work, call `sub.Unsubscribe()`. There is a Go routine per subscription. These will be cleaned up on `Unsubscribe()`, or upon connection teardown.

## How do I create subjects?

Subjects are created and pruned (deleted) dynamically based on interest (subscriptions). This means that a subject does not exist in a NATS cluster until a client subscribes to it, and the subject goes away after the last subscribing client unsubscribes from that subject.

## How many clients can connect simultaneously?

The default setting for a single server is 65,536. Although there is no specified limit to the number of connections supported by NATS, there are some environmental factors that will influence your decision as to how many connections to allow per server.

Most systems can handle several thousand NATS connections per server without any changes although some have a very low default such as OS X. You'll want to look at kernel/OS settings to increase that limit. You'll also want to look at default TCP buffer sizes to best optimize your machine for your traffic characteristics.

If you are using TLS you'll want to be sure the hardware can handle the CPU load created by TLS negotiation when there is the thundering herd of inbound connections after an outage or network

partition event. This often overlooked factor is usually the constraint limiting the number of connections a single server should support. Choosing a cipher suite that is supported by TLS acceleration can mitigate this (e.g. AES with x86). Thinking of the entire system, you'll also want to look at a range of reconnect delay times or add reconnect jitter to the NATS clients to even out the distribution of connection attempts over time and reduce CPU spikes.

All said, each server can be tuned to handle a large number of clients, and given the flexibility and scalability of NATS with clusters, superclusters, and leaf nodes one can build a NATS deployment supporting many millions of connections.

# NATS Protocols

This section of the documentation contains the reference documentation for the NATS network protocols used between clients applications and server and between nats servers interconnected in a cluster.

# Protocol Demo

The virtues of the NATS protocol manifest quickly when you experience how easy it is to use NATS. Because the NATS protocol is text-based, you can use NATS across virtually any platform or language.

In the following demo we use [Telnet](#). On the wire you can publish and subscribe using a simple [set of protocol commands](#).

## Initiate a connection

Open a terminal and initiate a connection to the NATS demo instance.

```
telnet demo.nats.io 4222
```

The expected result will roughly look like this. Note the IP address, and `INFO` payload may have different values.

```
Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^]'.
INFO
{"server_id":"NCXMJZYQEWDJFLYLSTTE745I2WUNCVG3LJJ3NRKSFJXEG6RGK7753DJ","version":"2.0.0","proto":1,"go":"go1.11.10","host":"0.0.0.0","port":4222,"max_payload":1048576,"client_id":5089}
```

## Confirm the connection

Any client establishing a connection with the server must send `CONNECT` message to confirm the connection. There are several options that can be specified to indicate the client supported features, but for the purpose of this example, we can send an empty payload.

```
CONNECT {}
```

You will see a `+OK` message in response.

# Observe the ping/pong interval

Not long after the `CONNECT` you will see a `PING` message. There is a bi-directional behavior between the client and server to check for liveness. In the case of the server, after some period of time without a `PONG`, the server will shutdown the client connection. If the client does not hear back a `PONG`, it will attempt to reconnect to a different server in a clustered setup, if available.

You can respond to the server, but simply typing `PONG` followed by a return.

```
PONG
```

## Bind a subscription

Subscribe to the wildcard subject `foo.*` with subscription ID of `90`.

```
SUB foo.* 90
```

An `+OK` message will follow, indicating a successful subscription.

## Publish a message

NATS connection are bi-directional, so we can not only subscribe to subjects, but we can publish to them as well.

We can send the `PUB` command followed by the subject, and the length of the message payload that will be followed on the second line. In this case, `hello` is the payload. Once that is typed, hit return which will send the message.

```
PUB foo.bar 5  
hello
```

An `+OK` message will follow, indicating a successful publish.

Immediately following, a `MSG` message will appear which indicates the subscription received the message that was just published.

```
MSG foo.bar 90 5  
hello
```

# Unsubscribe from the subject

You can use the `UNSUB` command to unsubscribe from a message.

Run the subscriber to unsubscribe:

```
UNSUB 90
```

# Try to publish again

Now that we unsubscribed, if we attempt to publish again, we will receive the `+OK`, but will not receive a `MSG`.

```
PUB foo.bar 7  
goodbye
```

# Close the connection

Use `ctrl+c` to close the connection, however, as noted above, if this is not done, after some period of inactivity the server will automatically close the connection.

# Client Protocol

## Client Protocol

The wire protocol used to communicate between the NATS server and clients is a simple, text-based publish/subscribe style protocol. Clients connect to and communicate with `nats-server` (the NATS server) through a regular TCP/IP socket using a small set of protocol operations that are terminated by a new line.

Unlike traditional messaging systems that use a binary message format that require an API to consume, the text-based NATS protocol makes it easy to implement clients in a wide variety of programming and scripting languages. In fact, refer to the topic [NATS Protocol Demo](#) to play with the NATS protocol for yourself using telnet.

The NATS server implements a [zero allocation byte parser](#) that is fast and efficient.

## Protocol conventions

**Control Line with Optional Content:** Each interaction between the client and server consists of a control, or protocol, line of text followed, optionally by message content. Most of the protocol messages don't require content, only `PUB`, `MSG`, `HPUB`, and `HMSG` include payloads.

**Field Delimiters:** The fields of NATS protocol messages are delimited by whitespace characters (space) or `\t` (tab). Multiple whitespace characters will be treated as a single field delimiter.

**Newlines:** NATS uses `\n` followed by `\r` (`\r\n`, `0x0D0A`) to terminate protocol messages. This newline sequence is also used to mark the end of the message payload in `PUB`, `MSG`, `HPUB`, and `HMSG` protocol messages.

**Subject names:** Subject names, including reply subject names, are case-sensitive and must be non-empty alphanumeric strings with no embedded whitespace. All UTF-8 characters except spaces/tabs and separators which are `.` and `>` are allowed. Subject names can be optionally token-delimited using the dot character (`.`), e.g.:

`F00`, `BAR`, `foo.bar`, `foo.BAR`, `F00.BAR` and `F00.BAR.BAZ` are all valid subject names

`F00. BAR`, `foo. .bar` and `foo..bar` are *not* valid subject names

A subject is comprised of 1 or more tokens. Tokens are separated by `.` and can be any non whitespace UTF-8 character. The full wildcard token `>` is only valid as the last token and matches all tokens past that point. A token wildcard, `*` matches any token in the position it was listed. Wildcard tokens should only be used in a wildcard capacity and not part of a literal token.

**Character Encoding:** Subject names should be UTF-8 compatible.

**Wildcards:** NATS supports the use of wildcards in subject subscriptions.

- The asterisk character (`*`) matches a single token at any level of the subject.
- The greater than symbol (`>`), also known as the *full wildcard*, matches one or more tokens at the tail of a subject, and must be the last token. The wildcarded subject `foo.>` will match `foo.bar` or `foo.bar.baz.1`, but not `foo`.
- Wildcards must be a separate token (`foo*.baz` or `foo.>` are syntactically valid; `foo*.bar`, `f*o.b*r` and `foo>` are not)

For example, the wildcard subscriptions `foo*.quux` and `foo.>` both match `foo.bar.quux`, but only the latter matches `foo.bar.baz`. With the full wildcard, it is also possible to express interest in every subject that may exist in NATS: `sub > 1`, limited of course by authorization settings.

# Protocol messages

The following table briefly describes the NATS protocol messages. NATS protocol operation names are case insensitive, thus `SUB foo 100` and `sub foo 100` are equivalent.

Click the name to see more detailed information, including syntax:

| OP Name                 | Sent By | Description                                                                        |
|-------------------------|---------|------------------------------------------------------------------------------------|
| <a href="#">INFO</a>    | Server  | Sent to client after initial TCP/IP connection                                     |
| <a href="#">CONNECT</a> | Client  | Sent to server to specify connection information                                   |
| <a href="#">PUB</a>     | Client  | Publish a message to a subject, with optional reply subject                        |
| <a href="#">HPUB</a>    | Client  | Publish a message to a subject including NATS headers, with optional reply subject |
| <a href="#">SUB</a>     | Client  | Subscribe to a subject (or subject wildcard)                                       |
| <a href="#">UNSUB</a>   | Client  | Unsubscribe (or auto-unsubscribe) from subject                                     |
| <a href="#">MSG</a>     | Server  | Delivers a message payload to a subscriber                                         |

| OP Name | Sent By | Description                                                  |
|---------|---------|--------------------------------------------------------------|
| HMSG    | Server  | Delivers a message payload to a subscriber with NATS headers |
| PING    | Both    | PING keep-alive message                                      |
| PONG    | Both    | PONG keep-alive response                                     |
| +OK     | Server  | Acknowledges well-formed protocol message in verbose mode    |
| -ERR    | Server  | Indicates a protocol error. May cause client disconnect.     |

The following sections explain each protocol message.

# INFO

## Description

A client will need to start as a plain TCP connection, then when the server accepts a connection from the client, it will send information about itself, the configuration and security requirements necessary for the client to successfully authenticate with the server and exchange messages.

When using the updated client protocol (see `CONNECT` below), `INFO` messages can be sent anytime by the server. This means clients with that protocol level need to be able to asynchronously handle `INFO` messages.

## Syntax

```
INFO {"option_name":option_value,...}[]
```

The valid options are as follows, encoded as JSON:

| name        | description                                           | type   | presence |
|-------------|-------------------------------------------------------|--------|----------|
| server_id   | The unique identifier of the NATS server.             | string | always   |
| server_name | The name of the NATS server.                          | string | always   |
| version     | The version of NATS.                                  | string | always   |
| go          | The version of golang the NATS server was built with. | string | always   |

| name          | description                                                                                                                                                           | type     | presence |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------|
| host          | The IP address used to start the NATS server, by default this will be 0.0.0.0 and can be configured with -client_advertise host:port.                                 | string   | always   |
| port          | The port number the NATS server is configured to listen on.                                                                                                           | int      | always   |
| headers       | Whether the server supports headers.                                                                                                                                  | bool     | always   |
| max_payload   | Maximum payload size, in bytes, that the server will accept from the client.                                                                                          | int      | always   |
| proto         | An integer indicating the protocol version of the server. The server version 1.2.0 sets this to 1 to indicate that it supports the "Echo" feature.                    | int      | always   |
| client_id     | The internal client identifier in the server. This can be used to filter client connections in monitoring, correlate with error logs, etc...                          | uint64   | optional |
| auth_required | If this is true, then the client should try to authenticate upon connect.                                                                                             | bool     | optional |
| tls_required  | If this is true, then the client must perform the TLS/1.2 handshake. Note, this used to be ssl_required and has been updated along with the protocol from SSL to TLS. | bool     | optional |
| tls_verify    | If this is true, the client must provide a valid certificate during the TLS handshake.                                                                                | bool     | optional |
| tls_available | If this is true, the client can provide a valid certificate during the TLS handshake.                                                                                 | bool     | optional |
| connect_urls  | List of server urls that a client can connect to.                                                                                                                     | [string] | optional |

| name                         | description                                                                                                                                                           | type     | presence |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------|
| <code>ws_connect_urls</code> | List of server urls that a websocket client can connect to.                                                                                                           | [string] | optional |
| <code>ldm</code>             | If the server supports <i>Lame Duck Mode</i> notifications, and the current server has transitioned to lame duck, <code>ldm</code> will be set to <code>true</code> . | bool     | optional |
| <code>git_commit</code>      | The git hash at which the NATS server was built.                                                                                                                      | string   | optional |
| <code>jetstream</code>       | Whether the server supports JetStream.                                                                                                                                | bool     | optional |
| <code>ip</code>              | The IP of the server.                                                                                                                                                 | string   | optional |
| <code>client_ip</code>       | The IP of the client.                                                                                                                                                 | string   | optional |
| <code>nonce</code>           | The nonce for use in CONNECT.                                                                                                                                         | string   | optional |
| <code>cluster</code>         | The name of the cluster.                                                                                                                                              | string   | optional |
| <code>domain</code>          | The configured NATS domain of the server.                                                                                                                             | string   | optional |

## connect\_urls

The `connect_urls` field is a list of urls the server may send when a client first connects, and when there are changes to server cluster topology. This field is considered optional, and may be omitted based on server configuration and client protocol level.

When a NATS server cluster expands, an `INFO` message is sent to the client with an updated `connect_urls` list. This cloud-friendly feature asynchronously notifies a client of known servers, allowing it to connect to servers not originally configured.

The `connect_urls` will contain a list of strings with an IP and port, looking like this:

```
"connect_urls":["10.0.0.184:4333","192.168.129.1:4333","192.168.192.1:4333"]
```

## Example

Below you can see a sample connection string from a telnet connection to the `demo.nats.io` site.

```
telnet demo.nats.io 4222
```

```
Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^]'.
INFO
{"server_id":"Zk0GQ3JBsrg3oyxCRRlE09","version":"1.2.0","proto":1,"go":"go1.10.3","host":"0.0.0.0","port":4222,"max_payload":1048576,"client_id":2392}
```

# CONNECT

## Description

The `CONNECT` message is the client version of the `INFO` message. Once the client has established a TCP/IP socket connection with the NATS server, and an `INFO` message has been received from the server, the client may send a `CONNECT` message to the NATS server to provide more information about the current connection as well as security information.

## Syntax

```
CONNECT {"option_name":option_value,...}[]
```

The valid options are as follows, encoded as JSON:

| name                      | description                                                                    | type   | required                                           |
|---------------------------|--------------------------------------------------------------------------------|--------|----------------------------------------------------|
| <code>verbose</code>      | Turns on <code>+OK</code> protocol acknowledgements.                           | bool   | true                                               |
| <code>pedantic</code>     | Turns on additional strict format checking, e.g. for properly formed subjects. | bool   | true                                               |
| <code>tls_required</code> | Indicates whether the client requires an SSL connection.                       | bool   | true                                               |
| <code>auth_token</code>   | Client authorization token.                                                    | string | if <code>auth_required</code> is <code>true</code> |
| <code>user</code>         | Connection username.                                                           | string | if <code>auth_required</code> is <code>true</code> |
| <code>pass</code>         | Connection password.                                                           | string | if <code>auth_required</code> is <code>true</code> |
| <code>name</code>         | Client name.                                                                   | string | false                                              |
| <code>lang</code>         | The implementation language of the client.                                     | string | true                                               |
| <code>version</code>      | The version of the client.                                                     | string | true                                               |

| name          | description                                                                                                                                                                                                                                                                                                                                | type   | required                       |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|--------------------------------|
| protocol      | Sending <code>0</code> (or absent) indicates client supports original protocol. Sending <code>1</code> indicates that the client supports dynamic reconfiguration of cluster topology changes by asynchronously receiving <code>INFO</code> messages with known servers it can reconnect to.                                               | int    | false                          |
| echo          | If set to <code>false</code> , the server (version 1.2.0+) will not send originating messages from this connection to its own subscriptions. Clients should set this to <code>false</code> only for server supporting this feature, which is when <code>proto</code> in the <code>INFO</code> protocol is set to at least <code>1</code> . | bool   | false                          |
| sig           | In case the server has responded with a <code>nonce</code> on <code>INFO</code> , then a NATS client must use this field to reply with the signed <code>nonce</code> .                                                                                                                                                                     | string | if <code>nonce</code> received |
| jwt           | The JWT that identifies a user permissions and account.                                                                                                                                                                                                                                                                                    | string | false                          |
| no_responders | Enable <a href="#">quick replies for cases where a request is sent to a topic with no responders</a> .                                                                                                                                                                                                                                     | bool   | false                          |
| headers       | Whether the client supports headers.                                                                                                                                                                                                                                                                                                       | bool   | false                          |
| nkey          | The public NKey to authenticate the client. This will be used to verify the signature ( <code>sig</code> ) against the <code>nonce</code> provided in the <code>INFO</code> message.                                                                                                                                                       | string | false                          |

## Example

Here is an example from the default string of the Go client:

```
CONNECT
```

```
{"verbose":false,"pedantic":false,"tls_required":false,"name":"","lang":"go","version":"1.2.2",
,"protocol":1}
```

Most clients set `verbose` to `false` by default. This means that the server should not confirm each message it receives on this connection with a [+OK](#) back to the client.

# PUB

## Description

The `PUB` message publishes the message payload to the given subject name, optionally supplying a reply subject. If a reply subject is supplied, it will be delivered to eligible subscribers along with the supplied payload. Note that the payload itself is optional. To omit the payload, set the payload size to 0, but the second CRLF is still required.

## Syntax

```
PUB <subject> [reply-to] <#bytes>[payload]
```

where:

| name                  | description                                                                                    | type   | required |
|-----------------------|------------------------------------------------------------------------------------------------|--------|----------|
| <code>subject</code>  | The destination subject to publish to.                                                         | string | true     |
| <code>reply-to</code> | The reply subject that subscribers can use to send a response back to the publisher/requestor. | string | false    |
| <code>#bytes</code>   | The payload size in bytes.                                                                     | int    | true     |
| <code>payload</code>  | The message payload data.                                                                      | string | false    |

## Example

To publish the ASCII string message payload "Hello NATS!" to subject FOO:

```
PUB F00 11Hello NATS!
```

To publish a request message "Knock Knock" to subject FRONT.DOOR with reply subject JOKE.22:

```
PUB FRONT.DOOR JOKE.22 11Knock Knock
```

To publish an empty message to subject NOTIFY:

```
PUB NOTIFY 0
```

# HPUB

## Description

The `HPUB` message is the same as `PUB` but extends the message payload to include NATS headers. Note that the payload itself is optional. To omit the payload, set the total message size equal to the size of the headers. Note that the trailing CR+LF is still required.

NATS headers are similar, in structure and semantics, to HTTP headers as `name: value` pairs including supporting multi-value headers. Headers can be mixed case and NATS will preserve case between message publisher and message receiver(s). See also [ADR-4 NATS Message Headers](#).

## Syntax

```
HPUB <subject> [reply-to] <#header bytes> <#total bytes>[headers][payload]
```

where:

| name          | description                                                                                    | type   | required |
|---------------|------------------------------------------------------------------------------------------------|--------|----------|
| subject       | The destination subject to publish to.                                                         | string | true     |
| reply-to      | The reply subject that subscribers can use to send a response back to the publisher/requestor. | string | false    |
| #header bytes | The size of the headers section in bytes including the delimiter before the payload.           | int    | true     |
| #total bytes  | The total size of headers and payload sections in bytes.                                       | int    | true     |

| name    | description                                                                                                                   | type   | required |
|---------|-------------------------------------------------------------------------------------------------------------------------------|--------|----------|
| headers | Header version <code>NATS/1.0</code> followed by one or more <code>name: value</code> pairs, each separated by <code>.</code> | string | false    |
| payload | The message payload data.                                                                                                     | string | false    |

## Example

To publish the ASCII string message payload "Hello NATS!" to subject FOO with one header Bar with value Baz:

```
HPUB FOO 22 33NATS/1.0Bar: BazHello NATS!
```

To publish a request message "Knock Knock" to subject FRONT.DOOR with reply subject JOKE.22 and two headers:

```
HPUB FRONT.DOOR JOKE.22 45 56NATS/1.0BREAKFAST: donutLUNCH: burgerKnock Knock
```

To publish an empty message to subject NOTIFY with one header Bar with value Baz:

```
HPUB NOTIFY 22 22NATS/1.0Bar: Baz
```

To publish a message to subject MORNING MENU with one header BREAKFAST having two values and payload "Yum!"

```
HPUB MORNING.MENU 47 51NATS/1.0BREAKFAST: donutBREAKFAST: eggsYum!
```

## SUB

## Description

`SUB` initiates a subscription to a subject, optionally joining a distributed queue group.

## Syntax

```
SUB <subject> [queue group] <sid>
```

where:

| name        | description                                                     | type   | required |
|-------------|-----------------------------------------------------------------|--------|----------|
| subject     | The subject name to subscribe to.                               | string | true     |
| queue_group | If specified, the subscriber will join this queue group.        | string | false    |
| sid         | A unique alphanumeric subscription ID, generated by the client. | string | true     |

# Example

To subscribe to the subject F00 with the connection-unique subscription identifier (sid) 1:

```
SUB F00 1
```

To subscribe the current connection to the subject BAR as part of distribution queue group G1 with sid 44:

```
SUB BAR G1 44
```

# UNSUB

## Description

UNSUB unsubscribes the connection from the specified subject, or auto-unsubscribes after the specified number of messages has been received.

## Syntax

```
UNSUB <sid> [max_msgs]
```

where:

| name | description                                                                 | type   | required |
|------|-----------------------------------------------------------------------------|--------|----------|
| sid  | The unique alphanumeric subscription ID of the subject to unsubscribe from. | string | true     |

| name     | description                                                          | type | required |
|----------|----------------------------------------------------------------------|------|----------|
| max_msgs | A number of messages to wait for before automatically unsubscribing. | int  | false    |

## Example

The following examples concern subject `F00` which has been assigned sid `1`. To unsubscribe from `F00`:

```
UNSUB 1
```

To auto-unsubscribe from `F00` after 5 messages have been received:

```
UNSUB 1 5
```

## MSG

## Description

The `MSG` protocol message is used to deliver an application message to the client.

## Syntax

```
MSG <subject> <sid> [reply-to] <#bytes>[payload]
```

where:

| name     | description                                                    | type   | presence |
|----------|----------------------------------------------------------------|--------|----------|
| subject  | Subject name this message was received on.                     | string | always   |
| sid      | The unique alphanumeric subscription ID of the subject.        | string | always   |
| reply-to | The subject on which the publisher is listening for responses. | string | optional |

| name    | description                   | type   | presence |
|---------|-------------------------------|--------|----------|
| #bytes  | Size of the payload in bytes. | int    | always   |
| payload | The message payload data.     | string | optional |

# Example

The following message delivers an application message from subject FOO.BAR:

```
MSG FOO.BAR 9 11Hello World
```

To deliver the same message along with a reply subject:

```
MSG FOO.BAR 9 GREETING.34 11Hello World
```

# HMSG

## Description

The HMSG message is the same as MSG, but extends the message payload with headers. See also [ADR-4 NATS Message Headers](#).

## Syntax

```
HMSG <subject> <sid> [reply-to] <#header bytes> <#total bytes>[headers][payload]
```

where:

| name     | description                                                    | type   | presence |
|----------|----------------------------------------------------------------|--------|----------|
| subject  | Subject name this message was received on.                     | string | always   |
| sid      | The unique alphanumeric subscription ID of the subject.        | string | always   |
| reply-to | The subject on which the publisher is listening for responses. | string | optional |

| name          | description                                                                                                                   | type   | presence |
|---------------|-------------------------------------------------------------------------------------------------------------------------------|--------|----------|
| #header bytes | The size of the headers section in bytes including the <code>----</code> delimiter before the payload.                        | int    | always   |
| #total bytes  | The total size of headers and payload sections in bytes.                                                                      | int    | always   |
| headers       | Header version <code>NATS/1.0</code> followed by one or more <code>name: value</code> pairs, each separated by <code>.</code> | string | optional |
| payload       | The message payload data.                                                                                                     | string | optional |

## Example

The following message delivers an application message from subject `FOO.BAR` with a header:

```
HMSG FOO.BAR 34 45NATS/1.0FoodGroup: vegetableHello World
```

To deliver the same message along with a reply subject:

```
HMSG FOO.BAR 9 BAZ.69 34 45NATS/1.0FoodGroup: vegetableHello World
```

## PING/PONG

### Description

`PING` and `PONG` implement a simple keep-alive mechanism between client and server. Once a client establishes a connection to the NATS server, the server will continuously send `PING` messages to the client at a configurable interval. If the client fails to respond with a `PONG` message within the configured response interval, the server will terminate its connection. If your connection stays idle for too long, it is cut off.

If the server sends a ping request, you can reply with a pong message to notify the server that you are still interested. You can also ping the server and will receive a pong reply. The ping/pong interval is configurable.

The server uses normal traffic as a ping/pong proxy, so a client that has messages flowing may not receive a ping from the server.

# Syntax

PING

PONG

## Example

The following example shows the demo server pinging the client and finally shutting it down.

```
telnet demo.nats.io 4222

Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^]'.
INFO
{"server_id":"Zk0GQ3JBsrg3oyxCRRlE09","version":"1.2.0","proto":1,"go":"go1.10.3","host":"0.0.0.0","port":4222,"max_payload":1048576,"client_id":2392}
PING
PING
-ERR 'Stale Connection'
Connection closed by foreign host.
```

## +OK/ERR

## Description

When the `verbose` connection option is set to `true` (the default value), the server acknowledges each well-formed protocol message from the client with a `+OK` message. Most NATS clients set the `verbose` option to `false` using the `CONNECT` message

The `-ERR` message is used by the server indicate a protocol, authorization, or other runtime connection error to the client. Most of these errors result in the server closing the connection.

Handling of these errors usually has to be done asynchronously.

## Syntax

+OK

-ERR <error message>

Some protocol errors result in the server closing the connection. Upon receiving these errors, the connection is no longer valid and the client should clean up relevant resources. These errors include:

- -ERR 'Unknown Protocol Operation': Unknown protocol error
- -ERR 'Attempted To Connect To Route Port': Client attempted to connect to a route port instead of the client port
- -ERR 'Authorization Violation': Client failed to authenticate to the server with credentials specified in the `CONNECT` message
- -ERR 'Authorization Timeout': Client took too long to authenticate to the server after establishing a connection (default 1 second)
- -ERR 'Invalid Client Protocol': Client specified an invalid protocol version in the `CONNECT` message
- -ERR 'Maximum Control Line Exceeded': Message destination subject and reply subject length exceeded the maximum control line value specified by the `max_control_line` server option. The default is 1024 bytes.
- -ERR 'Parser Error': Cannot parse the protocol message sent by the client
- -ERR 'Secure Connection - TLS Required': The server requires TLS and the client does not have TLS enabled.
- -ERR 'Stale Connection': The server hasn't received a message from the client, including a `PONG` in too long.
- -ERR 'Maximum Connections Exceeded': This error is sent by the server when creating a new connection and the server has exceeded the maximum number of connections specified by the `max_connections` server option. The default is 64k.
- -ERR 'Slow Consumer': The server pending data size for the connection has reached the maximum size (default 10MB).
- -ERR 'Maximum Payload Violation': Client attempted to publish a message with a payload size that exceeds the `max_payload` size configured on the server. This value is supplied to the client upon connection in the initial `INFO` message. The client is expected to do proper accounting of byte size to be sent to the server in order to handle this error synchronously.

Protocol error messages where the connection remains open are listed below. The client should not close the connection in these cases.

- -ERR 'Invalid Subject': Client sent a malformed subject (e.g. `sub foo. 90`)
- -ERR 'Permissions Violation for Subscription to <subject>': The user specified in the `CONNECT` message does not have permission to subscribe to the subject.
- -ERR 'Permissions Violation for Publish to <subject>': The user specified in the `CONNECT` message does not have permissions to publish to the subject.

# Developing a Client

## NATS Client Development Guide

This guide provides you with considerations for developing NATS clients, including:

- CONNECT handling
- Authorization
- Verbose (acks)
- Pedantic mode
- Ping/pong interval
- Parsing the protocol
- Deciding on a parsing strategy
- Storing and dispatching subscription callbacks
- Implementing requests/response
- Error handling, disconnecting and reconnecting
- Cluster support

Probably the best way to learn about implementing a client is to look at one of the client's maintained by the Synadia team. These clients are generally full featured, so if you can use them, that is even better, but if you have to write a client these may go beyond your needs while still capturing many of the design considerations discussed here.

- [GoLang](#)
- [Java](#)
- [C# / .NET](#)
- [Rust](#)
- [JavaScript](#)
- [Python](#)
- [C](#)

See [Developing with NATS](#) for more links and community supported languages.

## Client connection options

Clients can connect in authenticated or unauthenticated mode, as well as verbose mode which enables acknowledgements. See the [protocol documentation](#) for details.

# Client authorization

By default clients can connect to the server in unauthenticated mode. You can configure the NATS server to require password authentication to connect.

For example, using the command line:

```
nats-server -DV -m 8222 -user foo -pass bar
```

The client must then authenticate to connect to the server. For example:

```
nats.Connect("nats://foo:bar@localhost:4222")
```

# Verbose mode

When 'verbose' is enabled (via the `CONNECT` message), the NATS server will return `+OK` to acknowledge receipt of a valid protocol message. The NATS server automatically runs in verbose mode. Most client implementations disable verbose mode (set it to `false` in the `CONNECT` message) for performance reasons.

# Pedantic mode

A client may also support 'pedantic' mode. Pedantic mode indicates to the server that strict protocol enforcement is required.

# Ping/pong interval

NATS implements auto-pruning. When a client connects to the server, the server expects that client to be active. Periodically, the NATS server pings each subscriber, expecting a reply. If there is no reply within the configurable time limit, the server disconnects the client.

# Parsing the protocol

NATS provides a text-based message format. The text-based [protocol](#) makes it easy to implement NATS clients. The key consideration is deciding on a parsing strategy.

The NATS server implements a [zero allocation byte parser](#) that is fast and efficient. Off the wire, a NATS message is simply a slice of bytes. Across the wire the message is transported as an immutable string over a TCP connection. It is up to the client to implement logic to parse the message.

The NATS message structure includes the Subject string, an optional Reply string, and an optional Data field that is a byte array. The type `Msg` is a structure used by Subscribers and PublishMsg().

```
type Msg struct {  
    Subject string  
    Reply   string  
    Data    []byte  
    Sub     *Subscription  
}
```

A NATS publisher publishes the data argument to the given subject. The data argument is left untouched and needs to be correctly interpreted on the receiver. How the client parses a NATS message depends on the programming language.

## Deciding on a parsing strategy

Generally, protocol parsing for a NATS client is a string operation. In Python, for example, string operations are faster than regex. The Go and Java clients also use string operations to parse the message. But, if you look at the Ruby client, regex is used to parse the protocol because in Ruby regex is faster than string operations.

In sum, there is no magic formula for parsing—it depends on the programming language. But, you need to take into consideration how you are going to parse the message when you write a client.

## Storing and dispatching subscription callbacks

When you make a subscription to the server, you need to store and dispatch callback handlers.

On the client side, you need a hash map for this data structure. The hash map will be storing the callback that maps the subscription ID to the subscription.

The key of the hash map is the subscription ID. The key is used to look up the callback in the hash map. When you process the NATS message off the wire, you pass the parameters subject, reply subject, and the payload to the callback handler, which does its work.

Thus, you must store the mapping of subscription ID to the callback. Inside the subscription you have the callback.

## Implementing request/response

When to use pub/sub vs. req/rep depends on your use case. Run the tutorials for each to understand the differences between each style of implementation.

## Error handling, disconnecting and reconnecting

Considerations for error handling primarily include handling client disconnections and implementing retry logic.

## Cluster support

The NATS client has reconnection logic. So, if you are implementing clustering, you need to implement reconnect callbacks a priori, meaning you cannot modify it during runtime. When you start it, you need to have that information already.

# NATS Cluster Protocol

## NATS Cluster Protocol

The NATS server clustering protocol describes the protocols passed between NATS servers within a [cluster](#) to share accounts, subscriptions, forward messages, and share cluster topology regarding new servers. It is a simple text-based protocol. Servers communicate with each other through a regular TCP/IP or TLS socket using a small set of protocol operations that are terminated by newline.

The NATS server implements a [zero allocation byte parser](#) that is fast and efficient.

The NATS cluster protocol is very similar to that of the NATS client protocol. In the context of a cluster, it can be helpful to visualize a server being a proxy operating on behalf of its connected clients, subscribing, unsubscribing, sending and receiving messages.

## NATS Cluster protocol conventions

**Subject names and wildcards:** The NATS cluster protocol has the same features and restrictions as the client with respect to subject names and wildcards. Clients are bound to a single account, however the cluster protocol handles all accounts.

**Field Delimiters:** The fields of NATS protocol messages are delimited by whitespace characters ' ``'\(space\)` or `\t`` (tab). Multiple whitespace characters will be treated as a single field delimiter.

**Newlines:** Like other text-based protocols, NATS uses `CR` followed by `LF` (`CR+LF`, `\r`, `0x0D0A`) to terminate protocol messages. This newline sequence is also used to mark the beginning of the actual message payload in a `RMSG` protocol message.

## NATS Cluster protocol messages

The following table briefly describes the NATS cluster protocol messages. As in the client protocol, the NATS protocol operation names are case insensitive, thus `SUB foo 1\r` and `sub foo 1\r` are equivalent.

Click the name to see more detailed information, including syntax:

| OP Name                 | Sent By       | Description                                                                  |
|-------------------------|---------------|------------------------------------------------------------------------------|
| <a href="#">INFO</a>    | All Servers   | Sent after initial TCP/IP connection and to update cluster knowledge         |
| <a href="#">CONNECT</a> | All Servers   | Sent to establish a route                                                    |
| <a href="#">RS+</a>     | All Servers   | Subscribes to a subject for a given account on behalf of interested clients. |
| <a href="#">RS-</a>     | All Servers   | Unsubscribe (or auto-unsubscribe) from subject for a given account.          |
| <a href="#">RMSG</a>    | Origin Server | Delivers a message for a given subject and account to another server.        |
| <a href="#">PING</a>    | All Servers   | PING keep-alive message                                                      |
| <a href="#">PONG</a>    | All Servers   | PONG keep-alive response                                                     |
| <a href="#">-ERR</a>    | All Servers   | Indicates a protocol error. May cause the remote server to disconnect.       |

The following sections explain each protocol message.

# INFO

## Description

As soon as the server accepts a connection from another server, it will send information about itself and the configuration and security requirements that are necessary for the other server to successfully authenticate with the server and exchange messages.

The connecting server also sends an [INFO](#) message. The accepting server will add an [ip](#) field containing the address and port of the connecting server, and forward the new server's [INFO](#) message to all servers it is routed to.

Any servers in a cluster receiving an [INFO](#) message with an [ip](#) field will attempt to connect to the server at that address, unless already connected. This propagation of [INFO](#) messages on behalf of a connecting server provides automatic discovery of new servers joining a cluster.

## Syntax

```
INFO [{"option_name":option_value},...]
```

The valid options are as follows:

- `server_id`: The unique identifier of the NATS server
- `version`: The version of the NATS server
- `go`: The version of go lang the NATS server was built with
- `host`: The host specified in the cluster parameter/options
- `port`: The port number specified in the cluster parameter/options
- `auth_required`: If this is set, then the server should try to authenticate upon connect.
- `tls_required`: If this is set, then the server must authenticate using TLS.
- `max_payload`: Maximum payload size that the server will accept.
- `connect_urls`: A list of server urls that a client can connect to.
- `ip`: Optional route connection address of a server, `nats-route://<hostname>:<port>`

## Example

Below is an example of an `INFO` string received by a NATS server, with the `ip` field.

```
INFO
{"server_id":"KP19vTlB417XElnv8kKaC5","version":"2.0.0","go":"","host":"localhost","port":5222
,"auth_required":false,"tls_required":false,"tls_verify":false,"max_payload":1048576,"ip":"nat
s-route://127.0.0.1:5222/","connect_urls":["localhost:4222"]}
```

# CONNECT

## Description

The `CONNECT` message is analogous to the `INFO` message. Once the NATS server has established a TCP/IP socket connection with another server, and an `INFO` message has been received, the server will send a `CONNECT` message to provide more information about the current connection as well as security information.

## Syntax

```
CONNECT [{"option_name":option_value},...]
```

The valid options are as follows:

- `tls_required`: Indicates whether the server requires an SSL connection.
- `auth_token`: Authorization token
- `user`: Connection username (if `auth_required` is set)
- `pass`: Connection password (if `auth_required` is set)
- `name`: Generated Server Name
- `lang`: The implementation language of the server (go).
- `version`: The version of the server.

## Example

Here is an example from the default string from a server.

```
CONNECT {"tls_required":false,"name":"wt0vffeQyoDGMVBC2aKX0b"}\r
```

## RS+

## Description

`RS+` initiates a subscription to a subject on on a given account, optionally with a distributed queue group name and weighting factor. Note that queue subscriptions will use RS+ for increases and decreases to queue weight except when the weighting factor is 0.

## Syntax

**Subscription:** `RS+ <account> <subject>\r`

**Queue Subscription:** `RS+ <account> <subject> <queue> <weight>\r`

where:

- `account`: The account associated with the subject interest
- `subject`: The subject
- `queue`: Optional queue group name
- `weight`: Optional queue group weight representing how much interest/subscribers

## RS-

# Description

**RS-** unsubscribes from the specified subject on the given account. It is sent by a server when it no longer has interest in a given subject.

## Syntax

**Subscription:** `RS- <account> <subject>\r`

where:

- `account`: The account associated with the subject interest
- `subject`: The subject

# RMSG

## Description

The **RMSG** protocol message delivers a message to another server.

## Syntax

`RMSG <account> <subject> [reply-to] <#bytes>\r\n[payload]\r`

where:

- `account`: The account associated with the subject interest
- `subject`: Subject name this message was received on
- `reply-to`: The optional reply subject
- `#bytes`: Size of the payload in bytes
- `payload`: The message payload data

# PING/PONG

## Description

`PING` and `PONG` implement a simple keep-alive mechanism between servers. Once two servers establish a connection with each other, the NATS server will continuously send `PING` messages to other servers at a configurable interval. If another server fails to respond with a `PONG` message within the configured response interval, the server will terminate its connection. If your connection stays idle for too long, it is cut off.

If the another server sends a ping request, a server will reply with a pong message to notify the other server that it is still present.

## Syntax

`PING\r` `PONG\r`

## -ERR

## Description

The `-ERR` message is used by the server to indicate a protocol, authorization, or other runtime connection error to another server. Most of these errors result in the remote server closing the connection.

# JetStream wire API

## Reference

The normal way to use JetStream is through the NATS client libraries which expose a set of JetStream functions that you can use directly in your programs. But that is not the only way you can interact with the JetStream infrastructure programmatically. Just like core NATS has a wire protocol on top of TCP, the JetStream enabled nats-server(s) expose a set of Services over core NATS.

## Reference

All of these subjects are found as constants in the NATS Server source, so for example the subject `$JS.API.STREAM.LIST` is represented by `api.JSApiStreamList` constant in the nats-server source. Tables below will reference these constants and payload related data structures.

Note that if the resources you're trying to access have a JetStream [domain](#) associated with them, then the subject prefix will be `$JS.{domain}.API` rather than `$JS.API`.

## Error Handling

The APIs used for administrative tools all respond with standardised JSON and these include errors.

```
nats req '$JS.API.STREAM.INFO.nonexisting' ''
```

```
Published 11 bytes to $JS.API.STREAM.INFO.nonexisting
Received [_INBOX.lcWgjX2WgJLxqepU0K9pNf.mpBW9tHK] : {
  "type": "io.nats.jetstream.api.v1.stream_info_response",
  "error": {
    "code": 404,
    "description": "stream not found"
  }
}
```

```
nats req '$JS.STREAM.INFO.ORDERS' ''
```

```
Published 6 bytes to $JS.STREAM.INFO.ORDERS
Received [_INBOX.fwqdpWtG8XFHKfqhQDVA.vBecyWmF] : '{
  "type": "io.nats.jetstream.api.v1.stream_info_response",
  "config": {
    "name": "ORDERS",
    ...
  }
}
```

Here the responses include a `type` which can be used to find the JSON Schema for each response.

Non-admin APIs - like those for adding a message to the stream will respond with `-ERR` or `+OK` with an optional reason after.

## Admin API

All of the admin actions the `nats` CLI can do fall in the sections below. The API structure is kept in the `api` package in the `jsm.go` repository.

Subjects that end in `T` like `api.JSApiConsumerCreateT` are formats and would need to have the Stream Name and in some cases also the Consumer name interpolated into them. In this case `t := fmt.Sprintf(api.JSApiConsumerCreateT, streamName)` to get the final subject.

The command `nats events` will show you an audit log of all API access events which includes the full content of each admin request, use this to view the structure of messages the `nats` command sends.

The API uses JSON for inputs and outputs, all the responses are typed using a `type` field which indicates their Schema. A JSON Schema repository can be found in `nats-io/jsm.go/schemas`.

## General Info

| Subject                    | Constant                          | Description                                   | Request Payload | Response Payload                       |
|----------------------------|-----------------------------------|-----------------------------------------------|-----------------|----------------------------------------|
| <code>\$JS.API.INFO</code> | <code>api.JSApiAccountInfo</code> | Retrieves stats and limits about your account | empty payload   | <code>api.JetStreamAccountStats</code> |

## Streams

| Subject                                   | Constant                              | Description                                                                      | Request Payload                             | Response Payload                             |
|-------------------------------------------|---------------------------------------|----------------------------------------------------------------------------------|---------------------------------------------|----------------------------------------------|
| <code>\$JS.API.STREAM.LIST</code>         | <code>api.JSApiStreamList</code>      | Paged list known Streams including all their current information                 | <code>api.JSApiStreamListRequest</code>     | <code>api.JSApiStreamListResponse</code>     |
| <code>\$JS.API.STREAM.NAMES</code>        | <code>api.JSApiStreamNames</code>     | Paged list of Streams                                                            | <code>api.JSApiStreamNamesRequest</code>    | <code>api.JSApiStreamNamesResponse</code>    |
| <code>\$JS.API.STREAM.CREATE.*</code>     | <code>api.JSApiStreamCreateT</code>   | Creates a new Stream                                                             | <code>api.StreamConfig</code>               | <code>api.JSApiStreamCreateResponse</code>   |
| <code>\$JS.API.STREAM.UPDATE.*</code>     | <code>api.JSApiStreamUpdateT</code>   | Updates an existing Stream with new config                                       | <code>api.StreamConfig</code>               | <code>api.JSApiStreamUpdateResponse</code>   |
| <code>\$JS.API.STREAM.INFO.*</code>       | <code>api.JSApiStreamInfoT</code>     | Information about config and state of a Stream                                   | empty payload, Stream name in subject       | <code>api.JSApiStreamInfoResponse</code>     |
| <code>\$JS.API.STREAM.DELETE.*</code>     | <code>api.JSApiStreamDeleteT</code>   | Deletes a Stream and all its data                                                | empty payload, Stream name in subject       | <code>api.JSApiStreamDeleteResponse</code>   |
| <code>\$JS.API.STREAM.PURGE.*</code>      | <code>api.JSApiStreamPurgeT</code>    | Purges all of the data in a Stream, leaves the Stream                            | empty payload, Stream name in subject       | <code>api.JSApiStreamPurgeResponse</code>    |
| <code>\$JS.API.STREAM.MSG.DELETE.*</code> | <code>api.JSApiMsgDeleteT</code>      | Deletes a specific message in the Stream by sequence, useful for GDPR compliance | <code>api.JSApiMsgDeleteRequest</code>      | <code>api.JSApiMsgDeleteResponse</code>      |
| <code>\$JS.API.STREAM.MSG.GET.*</code>    | <code>api.JSApiMsgGetT</code>         | Retrieves a specific message from the stream                                     | <code>api.JSApiMsgGetRequest</code>         | <code>api.JSApiMsgGetResponse</code>         |
| <code>\$JS.API.STREAM.SNAPSHOT.*</code>   | <code>api.JSApiStreamSnapshotT</code> | Initiates a streaming backup of a streams data                                   | <code>api.JSApiStreamSnapshotRequest</code> | <code>api.JSApiStreamSnapshotResponse</code> |
| <code>\$JS.API.STREAM.RESTORE.*</code>    | <code>api.JSApiStreamRestoreT</code>  | Initiates a streaming restore of a stream                                        | <code>{}</code>                             | <code>api.JSApiStreamRestoreResponse</code>  |

## Consumers

| Subject                                                                       | Constant                              | Description                  | Request Payload                 | Response Payload                             |
|-------------------------------------------------------------------------------|---------------------------------------|------------------------------|---------------------------------|----------------------------------------------|
| <code>\$JS.API.CONSUMER.CREATE.&lt;stream&gt;</code>                          | <code>api.JSApiConsumerCreateT</code> | Create an ephemeral consumer | <code>api.ConsumerConfig</code> | <code>api.JSApiConsumerCreateResponse</code> |
| <code>\$JS.API.CONSUMER.DURABLE.CREATE.&lt;stream&gt;.&lt;consumer&gt;</code> | <code>api.JSApiDurableCreateT</code>  | Create a consumer            | <code>api.ConsumerConfig</code> | <code>api.JSApiConsumerCreateResponse</code> |

| Subject                                                                              | Constant                                | Description                                                                                                                                                                                                                                       | Request Payload                             | Response Payload                             |
|--------------------------------------------------------------------------------------|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|----------------------------------------------|
| <code>\$JS.API.CONSUMER.CREATE.&lt;stream&gt;.&lt;consumer&gt;.&lt;filter&gt;</code> | <code>api.JSApiConsumerCreateExT</code> | Create a consumer (server 2.9+)                                                                                                                                                                                                                   | <code>api.CreateConsumerRequest</code>      | <code>api.JSApiConsumerCreateResponse</code> |
| <code>\$JS.API.CONSUMER.LIST.&lt;stream&gt;</code>                                   | <code>api.JSApiConsumerListT</code>     | Paged list of known consumers including their current info for a given stream                                                                                                                                                                     | <code>api.JSApiConsumerListsRequest</code>  | <code>api.JSApiConsumerListsResponse</code>  |
| <code>\$JS.API.CONSUMER.NAMES.&lt;stream&gt;</code>                                  | <code>api.JSApiConsumerNamesT</code>    | Paged list of known consumer names for a given stream                                                                                                                                                                                             | <code>api.JSApiConsumerNamesRequest</code>  | <code>api.JSApiConsumerNamesResponse</code>  |
| <code>\$JS.API.CONSUMER.INFO.&lt;stream&gt;.&lt;consumer&gt;</code>                  | <code>api.JSApiConsumerInfoT</code>     | Information about a specific consumer by name                                                                                                                                                                                                     | empty payload                               | <code>api.JSApiConsumerInfoResponse</code>   |
| <code>\$JS.API.CONSUMER.DELETE.&lt;stream&gt;.&lt;consumer&gt;</code>                | <code>api.JSApiConsumerDeleteT</code>   | Deletes a Consumer                                                                                                                                                                                                                                | empty payload                               | <code>api.JSApiConsumerDeleteResponse</code> |
| <code>\$JS.API.CONSUMER.RESET.&lt;stream&gt;.&lt;consumer&gt;</code>                 | <code>api.JSApiConsumerResetT</code>    | Resets a consumer to a specific sequence number or timestamp. Or to the original starting point of the consumer if the payload was empty. The effective reset floor will never be earlier than the one set when the consumer was created.         | <code>api.JSApiConsumerResetResponse</code> |                                              |
| <code>\$JS.API.CONSUMER.PAUSE.&lt;stream&gt;.&lt;consumer&gt;</code>                 | <code>api.JSApiConsumerPauseT</code>    | Pauses or resumes a consumer. If payload is nil the consumer will be resumed. If payload is set it will contain the timestamp until which the consumers will be paused.                                                                           | empty payload                               | <code>api.JSApiConsumerPauseResponse</code>  |
| <code>\$JS.FC.&lt;stream&gt;.&gt;</code>                                             | N/A                                     | Consumer to subscriber flow control replies for <code>PUSH</code> consumer. Also used for sourcing and mirroring, which are implemented as <code>PUSH</code> consumers. If this subject is not forwarded, the consumer may stall under high load. | empty payload                               | N/A                                          |

| Subject                                   | Constant | Description                                                                                                                                                                                   | Request Payload | Response Payload |
|-------------------------------------------|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|------------------|
| <code>\$JSC.R.&lt;uid&gt;</code>          | N/A      | Reply subject used by source and mirror consumer create request                                                                                                                               | Consumer info   | N/A              |
| <code>\$JS.S.&lt;uid&gt;</code>           | N/A      | Default delivery subject for sourced streams. Can be overwritten by the <code>deliver</code> attribute in the source configuration.                                                           | Message data    | N/A              |
| <code>\$JS.M.&lt;uid&gt;</code>           | N/A      | Default delivery subject for mirrored streams. Can be overwritten by the <code>deliver</code> attribute in the source configuration.                                                          | Message data    | N/A              |
| <code>\$JS.ACK.&lt;stream&gt;.&gt;</code> | N/A      | Acknowledgments for <code>PULL</code> consumers. When this subject is not forwarded, <code>PULL</code> consumers in acknowledgment modes <code>all</code> or <code>explicit</code> will fail. | empty payload   | reply subject    |

## Stream Source and Mirror - Classic variant

Sourcing and mirroring streams use 3 inbound and 2 outbound subjects to establish and control the data flow. When setting permissions or creating export/import agreements all 5 subjects may need to be considered.

Notes:

- There are two variants to the consumer-create subject depending on the number of filters.
- In some setup a domain prefix may be present e.g.

```
$JS.<domain>.API.CONSUMER.CREATE.<stream>.>
```

| Subject | Direction | Description | Reply |
|---------|-----------|-------------|-------|
|---------|-----------|-------------|-------|

|                                                                                                          |          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                        |
|----------------------------------------------------------------------------------------------------------|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| <pre>\$JS.API.CONSUMER.CREATE.&lt;stream&gt;.&gt; and/or \$JS.API.CONSUMER.CREATE.&lt;stream&gt;</pre>   | outbound | <p>Create an ephemeral consumer to deliver pending messages. Note that this subject may be prefixed with a JetStream domain</p> <pre>\$JS.&lt;domain&gt;.API.CONSUMER.CREATE.&lt;stream&gt;.&lt;consumer&gt;.</pre> <p>The consumer create comes in 2 flavors depending on the number of filter subjects:</p> <p>*</p> <pre>\$JS.API.CONSUMER.CREATE.&lt;stream&gt;</pre> <p>- When there is no filter or there are multiple filters.</p> <p>*</p> <pre>\$JS.API.CONSUMER.CREATE.&lt;stream&gt;.&lt;consumer&gt;.&lt;filter subject&gt;</pre> <p>- When there is exactly one filter subject</p> | service request with <code>\$JSC.R.&lt;uid&gt;</code> as reply subject |
| <code>\$JS.FC.&lt;stream&gt;.&gt;</code>                                                                 | outbound | Flow control messages. Will on slow routes or when the target cannot keep up with the message flow.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | service request with <code>\$JSC.R.&lt;uid&gt;</code> as reply subject |
| <code>\$JSC.R.&lt;uid&gt;</code>                                                                         | inbound  | Reply to consumer creation request                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | reply message to service request                                       |
| <pre>\$JS.S.&lt;uid&gt; (source) OR \$JS.M.&lt;uid&gt; (mirror) OR &lt;custom delivery subject&gt;</pre> | inbound  | Message data and heartbeats                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | message stream                                                         |

## Heartbeats and Retries

The stream from which data is sourced/mirrored MAY NOT be reachable. It may not have been created yet OR the route may be down. This does not prevent the source/mirror agreement from being created.

- The target stream will try to create a consumer every 10s to 60s. (This value may change in the future or may be configurable). Note that delivery may therefore only resume after a short delay.
- For active consumers heartbeats are sent at a rate of 1/s.

## Constraints and Limitations

- Do not delete and recreate the original stream! Please flush/purge the stream instead. The target stream remembers the last sequence id to be delivered. A delete will reset the sequence ID.
- `$JS.FC.<stream>.>` - The flow control subject is NOT prefixed with a JetStream domain. This creates a limitation where identically named streams in different domains cannot be

reliably sourced/mirrored into the same account. Please create unique stream names to avoid this limitation.

# Stream Source and Mirror - With consumers

**This feature is available from server version 2.14**

Sourcing and mirroring streams can now make use of explicitly configured consumers. The consumer created in the origin stream (from which messages are being sourced/mirrored) must be of type `flow_control`. This now supports workqueue and interest-based streams.

It is strongly recommended to make the best use of this feature by pre-creating consumers. There are some situations where such consumers will be auto-created for backwards compatibility. Please avoid relying on this feature.

Sourcing and mirroring streams use 2 inbound and 2 outbound subjects to establish and control the data flow. When setting permissions or creating export/import agreements, all 4 subjects may need to be considered.

| Subject                                                              | Direction | Description                                                                                                                                                                                                           | Reply                                             |
|----------------------------------------------------------------------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| <code>\$JS.API.CONSUMER.RESET.&lt;stream&gt;.&lt;consumer&gt;</code> | outbound  | Reset the consumer to the desired state (last message seen +1). Note that this subject may be prefixed with a JetStream domain <code>\$JS.&lt;domain&gt;.API.CONSUMER.CREATE.&lt;stream&gt;.&lt;consumer&gt;</code> . | <code>\$JSC.R.&lt;uid&gt;</code> as reply subject |
| <code>\$JSC.R.&lt;uid&gt;</code>                                     | inbound   | Reply to consumer reset request                                                                                                                                                                                       | N/A                                               |
| <code>&lt;custom delivery subject&gt;</code>                         | inbound   | Message data and heartbeats                                                                                                                                                                                           | flow control and acks                             |
| <code>\$JS.FC.&lt;stream&gt;.&lt;stream&gt;.&lt;uid&gt;</code>       | outbound  | Flow control messages. Will on acknowledge messages or slow down the consumer.                                                                                                                                        | N/A                                               |

## Heartbeats and Retries

The stream from which data is sourced/mirrored MAY NOT be reachable. It may not have been created yet OR the route may be down. This does not prevent the source/mirror agreement from being created.

- The target stream will try to reset a consumer every 10s to 60s. (This value may change in the future or may be configurable). Note that delivery may therefore only resume after

a short delay.

- For active consumers, heartbeats are sent at a rate of 1/s.

## Constraints and Limitations

- Do not delete and recreate the original stream! Please flush/purge the stream instead. The target stream remembers the last sequence id to be delivered. A delete will reset the sequence ID.
- `$JS.FC.<stream>.<consumer>.*` - The flow control subject is NOT prefixed with a JetStream domain. This creates a limitation where identically named streams in different domains cannot be reliably sourced/mirrored into the same account. Please create unique stream names to avoid this limitation.

## ACLs

When using the subjects-based ACL, please note the patterns in the subjects grouped by purpose below.

### General information

```
$JS.API.INFO
```

### Stream Admin

```
$JS.API.STREAM.CREATE.<stream>
$JS.API.STREAM.UPDATE.<stream>
$JS.API.STREAM.DELETE.<stream>
$JS.API.STREAM.INFO.<stream>
$JS.API.STREAM.PURGE.<stream>
$JS.API.STREAM.LIST
$JS.API.STREAM.NAMES
$JS.API.STREAM.MSG.DELETE.<stream>
$JS.API.STREAM.MSG.GET.<stream>
$JS.API.STREAM.SNAPSHOT.<stream>
$JS.API.STREAM.RESTORE.<stream>
```

### Consumer Admin

```
$JS.API.CONSUMER.CREATE.<stream>
$JS.API.CONSUMER.DURABLE.CREATE.<stream>.<consumer>
$JS.API.CONSUMER.DELETE.<stream>.<consumer>
$JS.API.CONSUMER.RESET.<stream>.<consumer>
```

```
$JS.API.CONSUMER.PAUSE.<stream>.<consumer>  
$JS.API.CONSUMER.INFO.<stream>.<consumer>  
$JS.API.CONSUMER.LIST.<stream>  
$JS.API.CONSUMER.NAMES.<stream>
```

## Consumer message flow

```
$JS.API.CONSUMER.MSG.NEXT.<stream>.<consumer>  
$JS.SNAPSHOT.RESTORE.<stream>.<msg id>  
$JS.ACK.<stream>.<consumer>.x.x.x  
$JS.SNAPSHOT.ACK.<stream>.<msg id>  
$JS.FC.<stream>.>
```

## Optional Events and Advisories :

```
$JS.EVENT.METRIC.CONSUMER_ACK.<stream>.<consumer>  
$JS.EVENT.ADVISORY.CONSUMER.MAX_DELIVERIES.<stream>.<consumer>  
$JS.EVENT.ADVISORY.CONSUMER.MSG_TERMINATED.<stream>.<consumer>  
$JS.EVENT.ADVISORY.STREAM.CREATED.<stream>  
$JS.EVENT.ADVISORY.STREAM.DELETED.<stream>  
$JS.EVENT.ADVISORY.STREAM.UPDATED.<stream>  
$JS.EVENT.ADVISORY.CONSUMER.CREATED.<stream>.<consumer>  
$JS.EVENT.ADVISORY.CONSUMER.DELETED.<stream>.<consumer>  
$JS.EVENT.ADVISORY.STREAM.SNAPSHOT_CREATE.<stream>  
$JS.EVENT.ADVISORY.STREAM.SNAPSHOT_COMPLETE.<stream>  
$JS.EVENT.ADVISORY.STREAM.RESTORE_CREATE.<stream>  
$JS.EVENT.ADVISORY.STREAM.RESTORE_COMPLETE.<stream>  
$JS.EVENT.ADVISORY.STREAM.LEADER_ELECTED.<stream>  
$JS.EVENT.ADVISORY.STREAM.QUORUM_LOST.<stream>  
$JS.EVENT.ADVISORY.CONSUMER.LEADER_ELECTED.<stream>.<consumer>  
$JS.EVENT.ADVISORY.CONSUMER.QUORUM_LOST.<stream>.<consumer>  
$JS.EVENT.ADVISORY.API
```

This design allows you to easily create ACL rules that limit users to a specific Stream or Consumer and to specific verbs for administration purposes. For ensuring only the receiver of a message can Ack it we have response permissions ensuring you can only Publish to Response subject for messages you received.

# Acknowledging Messages

Messages that need acknowledgment will have a Reply subject set, something like `$JS.ACK.ORDERS.test.1.2.2`, this is the prefix defined in `api.JetStreamAckPre` followed by `<stream>.<consumer>.<delivered count>.<stream sequence>.<consumer sequence>.<timestamp>.<pending messages>`.

JetStream and the consumer (including sourced and mirrored streams) may exchange flow control messages. A message with the header: `NATS/1.0 100 FlowControl Request` must be replied to, otherwise the consumer may stall. The reply subjects looks like: `$JS.FC.orders.6i5h0GiQ.ep3Y`

In all of the Synadia maintained API's you can simply do `msg.Respond(nil)` (or language equivalent) which will send nil to the reply subject.

# Fetching The Next Message From a Pull-based Consumer

If you have a pull-based Consumer you can send a standard NATS Request to `$JS.API.CONSUMER.MSG.NEXT.<stream>.<consumer>`, here the format is defined in `api.JetStreamRequestNextT` and requires populating using `fmt.Sprintf()`.

```
nats req '$JS.API.CONSUMER.MSG.NEXT.ORDERS.test' '1'
```

```
Published 1 bytes to $JS.API.CONSUMER.MSG.NEXT.ORDERS.test
Received [js.1] : 'message 1'
```

Here we ask for just 1 message - `nats req` only shows 1 - but you can fetch a batch of messages by varying the argument. This combines well with the `AckAll` Ack policy.

The above request for the next message will stay in the server for as long as the client is connected and future pulls from the same client will accumulate on the server, meaning if you ask for 1 message 100 times and 1000 messages arrive you'll get sent 1000 messages not 1.

This is often not desired, pull consumers support a mode where a JSON document is sent describing the pull request.

```
{
  "expires": 7000000000,
  "batch": 10
}
```

This requests 10 messages and asks the server to keep this request for 7 seconds, this is useful when you poll the server frequently and do not want the pull requests to accumulate on the server.

Set the expire time to now + your poll frequency.

```
{
  "batch": 10,
  "no_wait": true
}
```

Here we see a second format of the Pull request that will not store the request on the queue at all but when there are no messages to deliver will send a nil bytes message with a `Status` header of `404`, this way you can know when you reached the end of the stream for example. A `409` is returned if the Consumer has reached `MaxAckPending` limits.

```
nats req '$JS.API.CONSUMER.MSG.NEXT.ORDERS.NEW' '{"no_wait": true, "batch": 10}'
```

```
13:45:30 Sending request on "$JS.API.CONSUMER.MSG.NEXT.ORDERS.NEW"
13:45:30 Received on "_INBOX.UKQGqq0W1EKl8inzXUlnaH.XJiawTRM" rtt 594.908µs
13:45:30 Status: 404
13:45:30 Description: No Messages
```

## Fetching From a Stream By Sequence

If you know the Stream sequence of a message, you can fetch it directly, this does not support acks. Do a Request() to `$JS.API.STREAM.MSG.GET.ORDERS` sending it the message sequence as payload. Here the prefix is defined in `api.JetStreamMsgBySeqT` which also requires populating using `fmt.Sprintf()`.

```
nats req '$JS.API.STREAM.MSG.GET.ORDERS' '{"seq": 1}'
```

```
Published 1 bytes to $JS.STREAM.ORDERS.MSG.BYSEQ
Received [_INBOX.cJrbzPJfZrq8NrFm1DsZuH.k91Gb4xM] : '{
  "type": "io.nats.jetstream.api.v1.stream_msg_get_response",
  "message": {
    "subject": "x",
    "seq": 1,
    "data": "aGVsbG8=",
    "time": "2020-05-06T13:18:58.115424+02:00"
  }
}
```

```
}'
```

The Subject shows where the message was received, Data is base64 encoded and Time is when it was received.

# Consumer Samples

Samples are published to a specific subject per Consumer, something like

`$JS.EVENT.METRIC.CONSUMER_ACK.<stream>.<consumer>` you can subscribe to that and get

`api.ConsumerAckMetric` messages in JSON format. The prefix is defined in

`api.JetStreamMetricConsumerAckPre`.

# Roadmap

You can find the NATS.io roadmap on the [nats.io main site](https://nats.io/main-site). See what is coming to NATS!

# Contributing

Do you want to contribute to NATS? Awesome! We'd love your help.

The best way to get started contributing to NATS is to read our [Contributor's Guide](#). Here are some examples of ways you can contribute:

- Report or fix bugs
- Add or propose new features
- Improve our documentation
- Add or update client libraries
- Host a Meetup
- Improve this website

Please [contact us](#) with any other suggestions.