JetStream JSON API Design

Metadata	Value
Date	2020-04-30
Author	@ripienaar
Status	Implemented
Tags	jetstream, client, server

Context

Several JetStream APIs exist focussed on message submission, administration and message retrieval. Where it makes sense these APIs are based on JSON documents and have Schemas and Schema type identifiers. In some cases this is not possible or would present a large overhead, so a more traditional text approach is taken for those APIs.

This document outlines the basic approach - rather than detail of every single API - it should give a good introduction to those wishing to write new clients or understand the underlying behaviors.

Overview

The API is built using NATS Request-Reply pattern, a Request to `$JS.API.STREAM.INFO.ORDERS` requests the information for the `ORDERS` stream, the response a JSON document that has type `io.nats.jetstream.api.v1.stream_info_response`.

In this case the API has no request input, the server will accept a nil payload or `{}` to indicate an empty request body, in cases where there is an optional input the optional JSON document can be added when needed.

In this example we accessed the subject `$JS.API.STREAM.INFO.<STREAM NAME>`, every API has a unique subject and generally the subjects include tokens indicating the item being accessed. This is to assist in generating ACLs giving people access to either subsets of API or even down to a single Stream or Consumer.

Errors are either in the form of JSON documents or a `-ERR` string style, more on this in the dedicated section.

One can observe the API in action using the `nats` CLI by adding the `--trace` option, any API interaction to and from JetStream is then logged showing Subjects and Bodies unmodified. This is an invaluable way to observe the interaction model.

Accessing

Accessing the API is via the Request-Reply system.

```
$ nats req '$JS.API.STREAM.NAMES' '{}'
14:18:14 Sending request on "$JS.API.STREAM.NAMES"
14:18:14 Received on "_INBOX.vrP0URcbRWXaMrqtFIDAm6.DiACQMKp" rtt 1.036883ms
```

```
{"type":"io.nats.jetstream.api.v1.stream_names_response","total":1,"offset":0,"lim
it":1024,"streams":["ORDERS"]}
```

Here the request in question had an empty payload, the server accepts nil, empty string or `{}` as valid payloads in that case.

Access to the API is via a unique subject per API, some of the subjects can be seen here, this is not an exhaustive list:

```
$JS.API.STREAM.CREATE.%s
$JS.API.STREAM.UPDATE.%s
$JS.API.STREAM.NAMES
$JS.API.STREAM.LIST
$JS.API.STREAM.INFO.%s
$JS.API.STREAM.DELETE.%s
$JS.API.STREAM.PURGE.%s
$JS.API.STREAM.MSG.DELETE.%s
$JS.API.STREAM.MSG.GET.%s
$JS.API.STREAM.SNAPSHOT.%s
$JS.API.STREAM.RESTORE.%s
$JS.API.STREAM.PEER.REMOVE.%s
$JS.API.STREAM.LEADER.STEPDOWN.%s
```

As you can see these are all STREAM related, the placeholder would be the name of the stream being accessed.

Paging

Some APIs like the `$JS.API.STREAM.NAMES` one above is paged, this is indicated by the presence of the `total`, `offset` and `limit` fields in the reply.

These APIs take a request parameter `offset` to move through pages, in other words:

```
$ nats req '$JS.API.STREAM.NAMES' '{"offset": 1024}'
```

Will list the stream names starting at number 1024.

Anatomy of a Response

Generally we have a few forms of response with some representative ones shown here:

Good response

```
{
  "type": "io.nats.jetstream.api.v1.stream_names_response",
  "total": 1,
```

```
  "offset": 0,
  "limit": 1024,
  "streams": [
    "KV_NATS"
  ]
}
```

This is not an error response (no `error` field), it's paged (has `total`, `offset` and `limit`) and is of the type `io.nats.jetstream.api.v1.stream_names_response` which indicates its schema (see below).

Error Response

```
{
  "type": "io.nats.jetstream.api.v1.consumer_info_response",
  "error": {
    "code": 404,
    "err_code": 10059,
    "description": "stream not found"
  }
}
```

This is a response to a consumer info request, it's type is `io.nats.jetstream.api.v1.consumer_info_response` which indicates its schema (see below). This is an error response, where the `description` field is variable and can change it's content between server versions. The fields of a healthy response are not shown for an error.

Schemas

All requests and responses that are in JSON format have JSON Schema Draft 7 documents describing them.

For example the request to create a stream has the type `io.nats.jetstream.api.v1.stream_create_request` and the response is a `io.nats.jetstream.api.v1.stream_create_response`. We have additional Schemas that describe some subsets of information for example `io.nats.jetstream.api.v1.stream_configuration` describes a valid Stream Configuration.

Generally requests do not expect the kind - it's inferred from which subject is accessed - but replies all have type hints.

The `nats` CLI can list, view and validate documents against these schemas:

```
$ nats schema list stream_create
Matched Schemas:

  io.nats.jetstream.api.v1.stream_create_request
  io.nats.jetstream.api.v1.stream_create_response
```

The document can be viewed, pass `--yaml` to view it in YAML format that's easier for reading:

```
$ nats schema show io.nats.jetstream.api.v1.stream_create_request
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://nats.io/schemas/jetstream/api/v1/stream_create_request.json",
  "description": "A request to the JetStream $JS.API.STREAM.CREATE API",
...
```

Finally, if you're developing an agent it could be useful to capture the JSON documents and validate them, the CLI can help with that:

```
$ nats schema validate io.nats.jetstream.api.v1.stream_create_request x.json
Validation errors in x.json:

  (root): max_consumers is required
  (root): max_msgs is required
```

Here we validated the `x.json` file and found a number of errors.

If you're writing a client and have JSON Schema validators to hand you can access the schemas in our schema repository

Data Types

The JetStream API has many data types, generally the JSON Schema tries to point this out, though we have some gaps in the data for sure. This section explores a few of the types in details.

Sequence related unsigned 64bit integers

JetStream can store up to the limit of an unsigned 64bit integer messages, this means status data, start points and more all have these kinds of unsigned 64 bit data types in them.

The type is particularly problematic near the top end of the scale since it exceeds what is possible with JSON, languages might need to have custom JSON parsers to handle this data. Though in practise this will only become a problem after many many years of creating data at full theoretical limit of message ingest.

Today all these fields are flagged in the schema, here's an example of one:

```
"opt_start_seq": {
  "minimum": 0,
  "$comment": "unsigned 64 bit integer",
  "type": "integer",
  "maximum": 18446744073709551615
}
```

Variable size integers

Some of the fields are limited by the architecture of the server - for example in practise on a 32bit system the number of consumers are limited to 32bit unsigned int, on a 64bit server it would be double.

Some fields are thus going to be dynamically capped to the server architecture, in practise you have to assume they are 64bit integer. These are in the schema with an example:

```
"max_deliver": {
  "description": "The number of times a message will be redelivered to consumers
if not acknowledged in time",
  "$comment": "integer with a dynamic bit size depending on the platform the
cluster runs on, can be up to 64bit",
  "type": "integer",
  "maximum": 9223372036854775807,
  "minimum": -9223372036854775807
}
```

Note while this is an unsigned integer, schema documents may list a different minimum than the minimum for the type.

32bit Integers

Similar to the 64bit integers, we just have a 32bit one, care should be taken not to overflow this number when sending data to the server, here's an example.

```
"max_msg_size": {
  "description": "The largest message that will be accepted by the Stream. -1 for
unlimited.",
  "minimum": -1,
  "default": -1,
  "$comment": "signed 32 bit integer",
  "type": "integer",
  "maximum": 2147483647
}
```

Note while this is an unsigned integer, schema documents may list a different minimum than the minimum for the type. Here the minimum is `-1`.

Time Durations

Some fields, like the maximum age of a message, are expressed as durations like `1 day`.

When sending this to the server you have to do it as a nanoseconds.

Here's a helper to turn some go durations into nanoseconds and confirm your calculations - play tool. Here change the `300h` into your time, this tool supports `1ms`, `1m`, `1h` and will help you turn those times into nanoseconds.

Time stamps

Specific time stamps should usually be expressed as UTC time and when sending times to the API this should be in UTC time also, but the server is flexible in handling times with zones and may at times give you back times as zones and will also accept them.

These times are in JSON as quoted strings in RFC 3339 format, with sub-second precision, here are some examples: `2021-07-22T15:42:12.580770412Z`, `2021-07-22T23:48:48.27104904+08:00`.

Error Handling

JSON Based

A JetStream API error looks like this:

```
{
  "type": "io.nats.jetstream.api.v1.consumer_info_response",
  "error": {
    "code": 404,
    "err_code": 10059,
    "description": "stream not found"
  }
}
```

The `error` key will only be present for error condition requests, the absense of a `error` usually indicates success.

Here we tried to get the status of a Consumer using a `io.nats.jetstream.api.v1.consumer_info_request` type message and we got a `404`. As with HTTP `404` means something is not found, it could be the Stream or the Consumer.

To avoid parsing - and treating as API - the error description we have an additional code `10059`. This is a unique NATS error:

```
$ nats error show 10059
NATS Error Code: 10059

        Description: stream not found
          HTTP Code: 404
  Go Index Constant: JSStreamNotFoundErr

No further information available
```

Looking at the error code `10059` we can tell that in this case the Stream was not found vs `10014` if the Consumer was not found.

The `nats` CLI has tools to view, search, list and more all the error code JetStream produce. These codes are static and will not change, however more might be added in time. For example today you might get a generic

code indicating invalid configuration but in future you might get a specific code indicating exactly what about your configuration was wrong.

More details about the Error Codes system can be found in ADR-7.

Text Based

Some APIs will respond with a text based error message, these will be in the form `-ERR <reason>`, these are very uncommon now in the API and will likely be entirely removed in time.