NATS Typed Messages

Metadata	Value
Date	2020-05-06
Author	@ripienaar
Status	Implemented
Tags	jetstream, server, client

Context

NATS Server has a number of JSON based messages - monitoring, JetStream API and more. These are consumed, and in the case of the API produced, by 3rd party systems in many languages. To assist with standardization of data validation, variable names and more we want to create JSON Schema documents for all our outward facing JSON based communication. Specifically this is not for server to server communication protocols.

This effort is ultimately not for our own use - though libraries like `jsm.go` will use these to do validation of inputs - this is about easing interoperability with other systems and to eventually create a Schema Registry.

There are a number of emerging formats for describing message content:

- JSON Schema - transport agnostic way of describing the shape of JSON documents
- AsyncAPI - middleware specific API description that uses JSON Schema for payload descriptions
- CloudEvents - standard for wrapping system specific events in a generic, routable, package. Supported by all major Public Clouds and many event gateways. Can reference JSON Schema.
- Swagger / OpenAPI - standard for describing web services that uses JSON Schema for payload descriptions

In all of these many of the actual detail like how to label types of event or how to version them are left up to individual projects to solve. This ADR describes how we are approaching this.

Decision

Overview

We will start by documenting our data types using JSON Schema Draft 7. AsyncAPI and Swagger can both reference these documents using remote references so this, as a starting point, gives us most flexibility and interoperability to later create API and Transport specific schemas that reference these.

We define 2 major type of typed message:

- `Message` - any message with a compatible `type` hint embedded in it
- `Event` - a specialized `message` that has timestamps and event IDs, suitable for transformation to Cloud Events. Typically, published unsolicited.

Today NATS Server do not support publishing Cloud Events natively however a bridge can be created to publish those to other cloud systems using the `jsm.go` package that supports converting `events` into Cloud Event format.

Message Types

There is no standard way to indicate the schema of a specific message. We looked at a lot of prior art from CNCF projects, public clouds and more but found very little commonality. The nearest standard is the Uniform Resource Name which still leaves most of the details up to the project and does not conventionally support versioning.

We chose a message type like `io.nats.jetstream.api.v1.consumer_delete_response`, `io.nats.server.advisory.v1.client_connect` or `io.nats.unknown_message`.

`io.nats.unknown_message` is a special type returned for anything without valid type hints. In go that implies `map[string]interface{}`.

The structure is as follows: io.nats.`<source>`.`<catagory>`.v`<version>`.`<name>`

Source

The project is the overall originator of a message and should be short but descriptive, today we have 2 - `server` and `jetstream` - as we continue to build systems around Stream Processing and more we'd add more of these types. I anticipate for example adding a few to Surveyor for publishing significant lifecycle events.

Generated Cloud Events messages has the `source` set to `urn:nats:<source>`.

Project	Description
`server`	The core NATS Server excluding JetStream related messages
`jetstream`	Any JetStream related message

Category

The `category` groups messages by related sub-groups of the `source`, often this also appears in the subjects these messages get published to.

This is a bit undefined, examples in use now are `api`, `advisory`, `metric`. Where possible try to fit in with existing chosen ones, if none suits update this table with your choice and try to pick generic category names.

Category	Description
`api`	Typically these are `messages` used in synchronous request response APIs
`advisory`	These are `events` that describe a significant event that happened like a client connecting or disconnecting
`metric`	These are `events` that relate to monitoring - how long did it take a message to be acknowledged

Versioning

The ideal outcome is that we never need to version any message and maintain future compatibility.

We think we can do that with the JetStream API. Monitoring, Observability and black box management is emerging, and we know less about how that will look in the long run, so we think we will need to version those.

The philosophy has to be that we only add fields and do not significantly change the meaning of existing ones, this means the messages stay `v1`, but major changes will require bumps. So all message types includes a single digit version.

Message Name

Just a string identifying what this message is about - `client_connect`, `client_disconnect`, `api_audit` etc.

Examples

Messages

At minimum a typed message must include a `type` string:

```
{
  "type": "io.nats.jetstream.api.v1.stream_configuration"
}
```

Rest of the document is up to the specific use case

Advisories

Advisories must include additional fields:

```
{
  "type": "io.nats.jetstream.advisory.v1.api_audit",
  "id": "uafvZ1UEDIW5FZV6kvLgWA",
  "timestamp": "2020-04-23T16:51:18.516363Z"
}
```

- `timestamp` - RFC 3339 format in UTC timezone, with sub-second precision added if present
- `id` - Any sufficiently unique ID such as those produced by `nuid`

Errors

Any `message` can have an optional `error` property if needed and can be specified in the JSON Schema, they are not a key part of the type hint system which this ADR focus on.

In JetStream ADR 0001 we define an error message as this:

```
{
  "error": {
    "description": "Server Error",
    "code": 500
  }
}
```

Where error codes follow basic HTTP standards. This `error` object is not included on success and so acceptable error codes are between `300` and `599`.

It'll be advantageous to standardise around this structure, today only JetStream API has this and we have not evaluated if this will suit all our needs.

Schema Storage

Schemas will eventually be kept in some form of formal Schema registry. In the near future they will all be placed as fully dereferenced JSON files at `http://nats.io/schemas`.

The temporary source for these can be found in the `nats-io/jetstream` repository including tools to dereference the source files.

Usage

Internally the `jsm.go` package use these Schemas to validate all requests to the JetStream API. This is not required as the server does its own validation too - but it's nice to fail fast and give extended errors like a JSON validator will give.

Once we add JetStream API support to other languages it would be good if those languages use the same Schemas for validation to create a unified validation strategy.

Eventually these Schemas could be used to generate the API structure.

The `nats` utility has a `nats events` command that can display any `event`. It will display any it finds, special formatting can be added using Golang templates in its source. Consider adding support to it whenever a new `event` is added.

Status

While this is marked `accepted`, we're still learning and exploring their usage so changes should be anticipated.

Consequences

Many more aspects of the Server move into the realm of being controlled and versioned where previously we took a much more relaxed approach to modifications to the data produced by `/varz` and more.