Service API

Metadata	Value
Date	2022-11-23
Author	@aricart
Status	Implemented
Tags	client, spec

Release History

Revision	Date	Description
1	2022-11-23	Initial release
2	2023-09-12	Configurable queue group
3	2023-10-07	Add version regex info
4	2023-11-10	Explicit naming
5	2024-08-08	Optional queue groups, immutable metadata
6	2025-02-17	Clarify adding endpoints after service creation

Context and Problem Statement

Simplify the development of NATS micro-services.

The design goal of the API is to reduce development to having a complexity similar to that of writing a NATS subscription, but then by using simple configuration, allow the specified metadata to allow for standardization of discovery and observability.

Design

Service configuration relies on the following:

- `name` - really the *kind* of the service. Shared by all the services that have the same name. This `name` can only have `A-Z, a-z, 0-9, dash, underscore`.
- `version` - a SemVer string - impl should validate that this is SemVer. One of the official semver regex should be used.
- `description` - a human-readable description about the service (optional)
- `metadata` - (optional) an object of strings holding free form metadata about the deployed instance implemented consistently with Metadata for Stream and Consumer ADR-33. Must be immutable once set.
- `statsHandler` - an optional function that returns unknown data that can be serialized as JSON. The handler will be provided the endpoint for which it is building a `EndpointStats`
- `queueGroup` - overrides a default queue group.

All services are created using a function called `addService()` where the above options are passed. The function returns an object/struct that represents the service. At a minimum the service is expected to offer functions/methods that allow:

- a `stop(error?)` function that allows user code to stop the service. Optionally this function should allow for an optional error. Stop should always drain its service subscriptions.
- `reset()` to reset any tracked metrics
- `info()` returns the `ServiceInfo`
- `stats()` to return the stats of the service
- A callback handler or promise where the framework can notify when the service has stopped. Note that this is independent of the NATS connection, and it should be possible to run multiple services under a single connection.
- `addEndpoint(name, handler, options?)` to configure new endpoints on a service. See Adding groups and endpoints
- `addGroup(name)` to set up endpoint group. `addGroup()`. See Adding groups and endpoints

On startup a service is assigned an unique `id`. This `id` is used to distinguish different instances of the service and allow for a specific instance of the service to be addressed.

Discovery and Status

Using the specified `name` and automatically generated `id` the service will automatically create a subscription to handle discovery and monitoring requests.

The subject for discovery and requests is prefixed by `$SRV`. Note that this prefix needs to be overridable much in the way as we do for `$JS`, in order to enable targetting tools to work across accounts. The prefix will honor whatever case was specified.

The initial *verbs* supported by the service include:

- `PING`
- `STATS`
- `INFO`

Using the above verbs, it becomes possible to build a service subject hierarchy like:

`$SRV.PING|STATS|INFO` - pings and retrieves status for all services `$SRV.PING|STATS|INFO.<name>` - pings or retrieves status for all services having the specified name `$SRV.PING|STATS|INFO.<name>.<id>` - pings or retrieves status of a particular service instance

Note that `name` matches whatever was specified and `id` matches whatever was generated by the service.

Services should respond to:

- All service requests
- All service requests that match their `name`
- All services requests that match their `name` and `id`

Standard Field

All discovery and status responses contain the following fields:

```
/**
 * An identifier of the message type for example io.nats.micro.v1.stats
 */
type: string,
/**
* The kind of the service reporting the status
*/
name: string,
/**
* The unique ID of the service reporting the status
*/
id: string,
/**
* The version of the service
* Should be validated using official semver regexp:
* https://semver.org/#is-there-a-suggested-regular-expression-regex-to-check-a-
semver-string
*/
version: string
/**
 *  The supplied service metadata
 */
metadata: Record<string,string>;
```

INFO

Returns a JSON having the following structure:

```
{
    type: string,
    name: string,
    id: string,
    version: string,
    metadata: Record<string,string>;
    /**
     * Description for the service
     */
    description: string,
    /**
     * An array of info for all service endpoints
     */
    endpoints: EndpointInfo[]
}
```

```
// EndpointInfo
{
    /**
     * The name of the endpoint
     */
    name: string,
    /**
     * The subject on which the endpoint is listening.
     */
    subject: string,
    /**
     * Queue group to which this endpoint is assigned to
     */
    queue_group: string,
    /**
     * Metadata of a specific endpoint
     */
    metadata: Record<string,string>,
}
```

All the fields above map 1-1 to the metadata provided when the service was created.

The type for this is `io.nats.micro.v1.info_response`.

PING

Returns the following JSON (the standard response fields)

```
{
    type: string,
    name: string,
    id: string,
    version: string,
    metadata: Record<string,string>;
}
```

The intention of `PING` is for clients to calculate RTT to a service and discover services.

The type for this is `io.nats.micro.v1.ping_response`.

STATS

```
{
    type: string,
    name: string,
    id: string,
    version: string,
    metadata: Record<string,string>,
    /**
    * Individual endpoint stats
    */
    endpoints: EndpointStats[],
    /**
```

```
      * ISO Date string when the service started in UTC timezone
      */
     started: string
}

/**
 * EndpointStats
 */
{
    /**
    * The name of the endpoint
    */
    name: string;
    /**
    * The subject on which the endpoint is listening.
    */
    subject: string;
    /**
    * Queue group to which this endpoint is assigned to
    */
    queue_group: string;
    /**
    * The number of requests received by the endpoint
    */
    num_requests: number;
    /**
    * Number of errors that the endpoint has raised
    */
    num_errors: number;
    /**
    * If set, the last error triggered by the endpoint
    */
    last_error?: Error;
    /**
    * A field that can be customized with any data as returned by stats handler
see {@link ServiceConfig}
    */
    data?: unknown;
    /**
    * Total processing_time for the service
    */
    processing_time: Nanos;
    /**
    * Average processing_time is the total processing_time divided by the
num_requests
    */
    average_processing_time: Nanos;
}
```

The type for this is `io.nats.micro.v1.stats_response`.

Adding groups and endpoints

A service can be extended by adding additional groups and endpoints. Endpoints can be added after the service has been created and started. For now, there is no option to remove an endpoint or a group.

Groups

A group serves as a common prefix to all endpoints registered in it. A group can be created using `addGroup(name)` method on a Service. Group name should be a valid NATS subject or an empty string, but cannot contain `>` wildcard (as group name serves as subject prefix). Group can have a default `queueGroup` for endpoints that overrides service `queueGroup`.

Group should expose following methods:

- `addEndpoint(name, handler, options?)` - registers new endpoint for the service. By default, the endpoint should be registered on subject created by concatenating group name and endpoint subject: `{this.group_name}.{name}`. Alternatively, user may pass `subject` as an option, in which case service will be registered on `{this.group_name}.{subject}`
- `addGroup(name)` - creates and returns a new group. The prefix for this group is created as follows: `{this.group_name}.{name}`.

Endpoints

Each service endpoint consists of the following fields:

- `name` - an alphanumeric human-readable string used to describe the endpoint. Multiple endpoints can have the same names.
- `handler` - request handler - see Request Handling
- `metadata` - an optional `Record<string,string>` providing additional information about the endpoint. Must be immutable once set.
- `subject` - an optional NATS subject on which the endpoint will be registered. A subject is created by concatenating the subject provided by the user with group prefix (if applicable). If subject is not provided, use `name` instead.
- `queueGroup` - optional override for a service and group `queueGroup`.

Endpoints can be created either on the service directly (`Service.addEndpoint()`) or on a group (`Group.addEndpoint`).

Clients should provide an idiomatic way to set no `queueGroup` when unset the subscription for the endpoint will be a normal subscribe instead of a queue subscribe.

Error Handling

Services may communicate request errors back to the client as they see fit, but to help standardization they also must include the headers: `Nats-Service-Error` and `Nats-Service-Error-Code`.

`Nats-Service-Error-Code` should be a value that is always safe to parse as a number. `Nats-Service-Error` should be a string describing the error that could be shown to the user.

This means that clients making request from the service *must* check if the response is an error by looking for these headers. This allows client code to be fairly standard in terms of handling regardless of additional error handling conventions.

Service API libraries *must* provide an error formatting function that users can use to produce the properly formatted response headers.

Service Msg

Clients may optionally implement a Service Msg, which adds additional respond functionality such as:

```
respondError(code: number, description: string, data?: Uint8Array, opts?:
PublishOptions): boolean
```

This enables a service to easily on-board the service error without requiring users to create their own shims. The above adds two required arguments: the error code, and description, the rest should match the client's implementation of `respond()`.

Request Handling

All service request handlers operate under the default queue group `q`. This means that in order to scale up or down all the user needs to do is add or stop services. Its possible to send request to multiple services, for example to minimize response time by using the quickest responder. To achieve that, it requires running some service instances with different `queueGroup`.

For each configured endpoint, a queue subscription should be created. Unless the option to create a normal enqueued subscription is activated.

> Note: Handler subject does not contain the `$SRV` prefix. This prefix is reserved for internal handlers.

The handlers specified by the client to process requests should operate as any standard subscription handler. This means that no assumption is made on whether returning from the callback signals that the request is completed. The framework will dispatch requests as fast as the handler returns.

Naming

For consistency of documentation and understanding by users, clients that implement the service API and tooling that interacts with services should use the term "service" or "services".