

Configuring Subject Mapping



- [Configuring subject mapping](#)
 - [Static authentication](#)
 - [JWT authentication](#)
 - [Simple Mapping](#)
 - [Subject Token Reordering](#)
 - [Weighted Mappings for A/B Testing or Canary Releases](#)
 - [Traffic Shaping in Testing](#)
 - [Artificial Loss](#)
-

Supported since NATS Server version 2.2

Subject mapping is a very powerful feature of the NATS server, useful for canary deployments, A/B testing, chaos testing, and migrating to a new subject namespace.

Configuring subject mapping

Subject mappings are defined and applied at the account level. If you are using static account security you will need to edit the server configuration file, however if you are using JWT Security (Operator Mode), then you need to use nsc or customer tools to edit and push changes to your account.

NOTE: You can also use subject mapping as part of defining imports and exports between accounts

Static authentication

In any of the static authentication modes the mappings are defined in the server configuration file, any changes to mappings in the configuration file will take effect as soon as a reload signal is sent to the server process (e.g. use `nats-server --signal reload`).

The `mappings` stanza can occur at the top level to apply to the global account or be scoped within a specific account.

```

mappings = {

  # Simple direct mapping.  Messages published to foo are mapped to bar.
  foo: bar

  # remapping tokens can be done with $<N> representing token position.
  # In this example bar.a.b would be mapped to baz.b.a.
  bar.*.*: baz.$2.$1

  # You can scope mappings to a particular cluster
  foo.cluster.scoped : [
    { destination: bar.cluster.scoped, weight:100%, cluster: us-west-1 }
  ]

  # Use weighted mapping for canary testing or A/B testing.  Change dynamically
  # at any time with a server reload.
  myservice.request: [
    { destination: myservice.request.v1, weight: 90% },
    { destination: myservice.request.v2, weight: 10% }
  ]

  # A testing example of wildcard mapping balanced across two subjects.
  # 20% of the traffic is mapped to a service in QA coded to fail.
  myservice.test.*: [
    { destination: myservice.test.$1, weight: 80% },
    { destination: myservice.test.fail.$1, weight: 20% }
  ]

  # A chaos testing trick that introduces 50% artificial message loss of
  # messages published to foo.loss
  foo.loss.>: [ { destination: foo.loss.>, weight: 50% } ]
}

```

JWT authentication

When using the JWT authentication mode, the mappings are defined in the account's JWT. Account JWTs can be created or modified either through the [_JWT API_](#) or using the `nsc` CLI too. For more detailed information see `nsc add mapping --help`, `nsc delete mapping --help`. Subject mapping changes take effect as soon as the modified account JWT is pushed to the nats servers (i.e. `nsc`

`push`).

Examples of using `nsc` to manage mappings:

- Add a new mapping: `nsc add mapping --from "a" --to "b"`
- Modify an entry, say to set a weight after the fact: `nsc add mapping --from "a" --to "b" --weight 50`
- Add two entries from one subject, set weights and execute multiple times: `nsc add mapping --from "a" --to "c" --weight 50`
- Delete a mapping: `nsc delete mapping --from "a"`

Simple Mapping

The example of `foo:bar` is straightforward. All messages the server receives on subject `foo` are remapped and can be received by clients subscribed to `bar`.

Subject Token Reordering

Wildcard tokens may be referenced via `$(position)`. For example, the first wildcard token is \$1, the second is \$2, etc. Referencing these tokens can allow for reordering.

With this mapping:

```
bar.*.*: baz.$2.$1
```

Messages that were originally published to `bar.a.b` are remapped in the server to `baz.b.a`. Messages arriving at the server on `bar.one.two` would be mapped to `baz.two.one`, and so forth.

Weighted Mappings for A/B Testing or Canary Releases

Traffic can be split by percentage from one subject to multiple subjects. Here's an example for canary deployments, starting with version 1 of your service.

Applications would make requests of a service at `myservice.requests`. The responders doing the work of the server would subscribe to `myservice.requests.v1`. Your configuration would look like this:

```
myservice.requests: [  
  { destination: myservice.requests.v1, weight: 100% }  
]
```

All requests to `myservice.requests` will go to version 1 of your service.

When version 2 comes along, you'll want to test it with a canary deployment. Version 2 would subscribe to `myservice.requests.v2`. Launch instances of your service (don't forget about queue subscribers and load balancing).

Update the configuration file to redirect some portion of the requests made to `myservice.requests` to version 2 of your service. In this case we'll use 2%.

```
myservice.requests: [  
  { destination: myservice.requests.v1, weight: 98% },  
  { destination: myservice.requests.v2, weight: 2% }  
]
```

You can [reload](#) the server at this point to make the changes with zero downtime. After reloading, 2% of your requests will be serviced by the new version.

Once you've determined Version 2 stable switch 100% of the traffic over and reload the server with a new configuration.

```
myservice.requests: [  
  { destination: myservice.requests.v2, weight: 100% }  
]
```

Now shutdown the version 1 instances of your service.

Traffic Shaping in Testing

Traffic shaping is useful in testing. You might have a service that runs in QA that simulates failure scenarios which could receive 20% of the traffic to test the service requestor.

```
myservice.requests.*: [  
  { destination: myservice.requests.$1, weight: 80% },  
  { destination: myservice.requests.fail.$1, weight: 20% }  
]
```

Artificial Loss

Alternatively, introduce loss into your system for chaos testing by mapping a percentage of traffic to the same subject. In this drastic example, 50% of the traffic published to `foo.loss.a` would be artificially dropped by the server.

```
foo.loss.>: [ { destination: foo.loss.>, weight: 50% } ]
```

You can both split and introduce loss for testing. Here, 90% of requests would go to your service, 8% would go to a service simulating failure conditions, and the unaccounted for 2% would simulate message loss.

```
myservice.requests: [  
  { destination: myservice.requests.v3, weight: 90% },  
  { destination: myservice.requests.v3.fail, weight: 8% }  
  # the remaining 2% is "lost"  
]
```

📄 #3

📄 cocobook-sync 📄 2026-10-28 11:14:24 UTC

📄 cocobook-sync 📄 2026-10-29 01:42:39 UTC