

In Depth JWT Guide



- [Concepts](#)
 - [What are Accounts?](#)
 - [Key Takeaways](#)
 - [What are NKEYs?](#)
 - [Key Takeaways](#)
- [JSON Web Tokens \(JWT\)](#)
 - [Motivation for JWT](#)
 - [Key Takeaways](#)
 - [Decentralized Authentication/Authorization using JWT](#)
 - [Key Takeaways](#)
 - [NATS JWT Hierarchy](#)
 - [Decentralized Chain of Trust](#)
 - [Obtain an Account JWT](#)
 - [JWT and Chain of Trust Verification](#)
 - [Obtain a User JWT - Client Connect](#)
 - [Key Takeaways](#)
 - [Deployment Models Enabled by Chain of Trust](#)
 - [Key Takeaways](#)
- [Accounts Re-visited](#)
 - [Key Takeaways](#)
- [Tooling And Key Management](#)
 - [nsc](#)
 - [Environment](#)
 - [Backup](#)
 - [NKEYS store directory](#)
 - [JWT store directory](#)
 - [Names in JWT](#)
 - [Setup an Operator](#)

- [Create/Edit Operator - Operator Environment - All Deployment modes](#)
- [Import Operator - Non Operator/Administrator Environment - Decentralized/Self Service Deployment Mod](#)
- [Import Operator - Self Service Deployment Modes](#)
- [Setup an Account](#)
 - [Create/Edit Account - All Environments - All Deployment modes](#)
 - [Export Account - Non Operator/Administrator Environment - Decentralized Deployment Modes](#)
 - [Export Account - Non Operator/Administrator Environment - Self Service Deployment Modes](#)
 - [Publicize an Account with Push - Operator Environment/Environment with push permissions - All Deploy](#)
 - [nats-resolver setup and push example - Operator Environment/Environment with push permissions - All](#)
- [Setup User](#)
 - [Create/Edit Account - All Environments - All Deployment modes](#)
- [Automated sign up services - JWT and NKEY libraries](#)
- [Simple user creation](#)
 - [Create user NKEY](#)
 - [Create user JWT](#)
 - [Distributed User Creation](#)
 - [User creation using NATS](#)
 - [Straight forward Setup](#)
 - [Account based Setup](#)
 - [Stamping JWT in languages other than Go](#)
- [System Account](#)
 - [Event Subjects](#)
- [Service Subjects](#)
 - [Subjects always available](#)
 - [Subjects available when using NATS-based resolver](#)
 - [Old Subjects](#)
- [Leaf Node Connections - Outgoing](#)
 - [Non Operator Mode](#)
 - [Operator Mode](#)

- [Connecting Accounts](#)
 - [Exports](#)
 - [Imports](#)
 - [Import Subjects](#)
 - [Import Remapping](#)
 - [Visualizing Export/Import Relationships](#)
 - [Managing Keys](#)
 - [Protect Identity NKEYs](#)
 - [Reissue Identity NKEYs](#)
 - [Operator](#)
 - [Account](#)
 - [Revocations](#)
 - [User](#)
 - [Activations](#)
 - [Accounts](#)
 - [Signing keys](#)
-

This document provides a step by step deep dive into JWT usage within NATS. Starting with related concepts, it will introduce JWTs and how they can be used in NATS. This will NOT list every JWT/nsc option, but will focus on the important options and concepts.

- [Concepts](#)
 - [What are Accounts?](#)
 - [Key Takeaways](#)
 - [What are NKEYs?](#)
 - [Key Takeaways](#)
- [JSON Web Tokens \(JWT\)](#)
 - [Motivation for JWT](#)
 - [Key Takeaways](#)
 - [Decentralized Authentication/Authorization using JWT](#)
 - [Key Takeaways](#)
 - [NATS JWT Hierarchy](#)
 - [Decentralized Chain of Trust](#)
 - [Obtain an Account JWT](#)
 - [JWT and Chain of Trust Verification](#)

- [Obtain a User JWT - Client Connect](#)
 - [Key Takeaways](#)
- [Deployment Models Enabled by Chain of Trust](#)
- [Key Takeaways](#)
- [Accounts Re-visited](#)
- [Key Takeaways](#)
- [Tooling And Key Management](#)
- [nsc](#)
 - [Environment](#)
 - [Backup](#)
 - [NKEYS store directory](#)
 - [JWT store directory](#)
 - [Names in JWT](#)
 - [Setup an Operator](#)
 - [Create/Edit Operator - Operator Environment - All Deployment modes](#)
 - [Import Operator - Non Operator/Administrator Environment - Decentralized/Self Service Deployment Modes](#)
 - [Import Operator - Self Service Deployment Modes](#)
 - [Setup an Account](#)
 - [Create/Edit Account - All Environments - All Deployment modes](#)
 - [Export Account - Non Operator/Administrator Environment - Decentralized Deployment Modes](#)
 - [Export Account - Non Operator/Administrator Environment - Self Service Deployment Modes](#)
 - [Publicize an Account with Push - Operator Environment/Environment with push permissions - All Deployment Modes](#)
 - [nats-resolver setup and push example - Operator Environment/Environment with push permissions - All Deployment Modes](#)
 - [Setup User](#)
 - [Create/Edit Account - All Environments - All Deployment modes](#)
 - [Automated sign up services - JWT and NKEY libraries](#)
 - [Simple user creation](#)
 - [Create user NKEY](#)
 - [Create user JWT](#)

- [Distributed User Creation](#)
- [User creation using NATS](#)
 - [Straight forward Setup](#)
 - [Account-based Setup](#)
- [Stamping JWT in languages other than Go](#)
- [System Account](#)
 - [Event Subjects](#)
 - [Service Subjects](#)
 - [Subjects always available](#)
 - [Subjects available when using NATS-based resolver](#)
 - [Old Subjects](#)
 - [Leaf Node Connections - Outgoing](#)
 - [Non-Operator Mode](#)
 - [Operator Mode](#)
- [Connecting Accounts](#)
 - [Exports](#)
 - [Imports](#)
 - [Import Subjects](#)
 - [Import Remapping](#)
 - [Visualizing Export/Import Relationships](#)
- [Managing Keys](#)
 - [Protect Identity NKEYs](#)
 - [Reissue Identity NKEYs](#)
 - [Operator](#)
 - [Account](#)
 - [Revocations](#)
 - [User](#)
 - [Activations](#)
 - [Accounts](#)
 - [Signing keys](#)

To exercise listed examples please have the following installed:

- nats-server: <https://github.com/nats-io/nats-server>
- nats (cli): <https://github.com/nats-io/natscli>

- nk (cli & library): <https://github.com/nats-io/nkeys>
- nsc (cli): <https://github.com/nats-io/nsc>
- jwt (library): <https://github.com/nats-io/jwt>

🔧 To install `nats-server`, `nats`, `nk`, `nsc`:

```
export GO111MODULE=on
go install github.com/nats-io/nats-server/v2@latest
go install github.com/nats-io/natscli/nats@latest
go install github.com/nats-io/nkeys/nk@latest
go install github.com/nats-io/nsc/v2@latest
```

To practice the examples below:

Save the configuration below in a file named say `server.conf`, and start nats server via:

```
nats-server -c server.conf
```

So that the server started. Later when we change the configuration we can do a reload like this:

```
nats-server --signal reload
```

Concepts

What are Accounts?

Accounts are the NATS isolation context.

```
accounts: {
  A: {
    users: [{user: a, password: a}]
  },
  B: {
    users: [{user: b, password: b}]
  }
}
```

```
    },  
  }  
}
```

Messages published in one account won't be received in another.

Listen for any message on account **a**:

```
nats -s nats://a:a@localhost:4222 sub ">"
```

Publish a message from account **b**:

```
nats -s nats://b:b@localhost:4222 pub "foo" "user b"
```

Note that you do not see this message received by your subscriber.

Now publish a messages from account **a**:

```
nats -s nats://a:a@localhost:4222 pub "foo" "user a"
```

This time the message is received by the subscriber:

```
17:57:06 [#1] Received on "foo"  
user a
```

The above example shows no message flow between user **a** associated with account **A** and user **b** in account **B**. Messages are delivered only within the same account. That is, unless you explicitly define it.

Below is a similar example, this time with messages crossing explicit account boundaries.

```
accounts: {  
  A: {  
    users: [{user: a, password: a}]  
    imports: [{stream: {account: B, subject: "foo"}}]  
  },  
  B: {  
    users: [{user: b, password: b}]  
    exports: [{stream: "foo"}]  
  },  
}
```

Modify `server.conf` and run `nats-server --signal reload`

Subscribe to everything as user 'a'

```
nats -s nats://a:a@localhost:4222 sub ">"
```

Publish on 'foo' as user 'b':

```
nats -s nats://b:b@localhost:4222 pub "foo" "user b"
```

This time the message is received by the subscriber:

```
18:28:25 [#1] Received on "foo"  
user b
```

Accounts are a lot more powerful than what has been demonstrated here. Take a look at the complete documentation of [accounts](#) and the [users](#) associated with them. All of this is in a plain NATS config file. (Copy the above config and try it using this command: `nats-server -c <filename>`) In order to make any changes, every participating nats-server config file in the same security domain has to change. This configuration is typically controlled by one organization or the administrator.

Key Takeaways

- Accounts are isolated from each other.
- One can selectively combine accounts,
- Need to modify a config file to add/remove/modify accounts and users,
- The config file can be applied to take effect via `nats-server --signal reload`.

What are NKEYs?

NKEYs are decorated, Base32 encoded, CRC16 check-summed, [Ed25519](#) keys.

Ed25519 is:

- a public key signature system. (can sign and verify signatures)
- resistant to side channel attacks (no conditional jumps in algorithm)

NATS server can be configured with public NKEYs as user (identities). When a client connects the nats-server sends a challenge for the client to sign in order to prove it is in possession of the corresponding private key. The nats-server then verifies the signed challenge. Unlike with a password based scheme, the secret never left the client.

To assist with knowing what type of key one is looking at, in config or logs, the keys are decorated as follows:

- Public Keys, have a one byte prefix: `0`, `A`, `U` for various types, `0` for operator, `A` for account, and `U` meaning user.
- Private Keys, have a two byte prefix `S0`, `SA`, `SU`. `S` stands for seed. The remainders(`0`, `A` and `U`) are the same meaning as in public keys.

NKEYs are generated as follows:

```
nk -gen user -pubout > a.nk
```

To view the key:

```
cat a.nk
```

```
SUAAEZYNLTEA2MDTG7L5X7Q0DZXYHP0I2LT2KH5I4GD6YVP24SE766EGPA
UC435ZYS52HF72E2VMQF4G06CUJ0CHDUUPEBU7XDXW5AQLIC6JZ46P05
```

Create another key:

```
nk -gen user -pubout > b.nk
```

View the key:

```
cat b.nk
```

```
SUANS4XLL5NWBTM57GSVHLN4TMFW55WGGWNI5YXXSI0YFJQYFVNHJK5GFY
UARZVI6JAV7YMJTPRANXAN00W4K3ZCD45NYP6S7C7XKCBHPVN2TFZ7ZC
```

Replacing the user/password with NKEY in account config example:

```
accounts: {
  A: {
    users: [{nkey:UC435ZYS52HF72E2VMQF4G06CUJ0CHDUUPEBU7XDXW5AQLIC6JZ46P05}]
    imports: [{stream: {account: B, subject: "foo"}}]
  },
  B: {
    users: [{nkey:UARZVI6JAV7YMJTPRANXAN00W4K3ZCD45NYP6S7C7XKCBHPVN2TFZ7ZC}]
    exports: [{stream: "foo"}]
  },
}
```

Simple example:

Subscribe with `nats -s nats://localhost:4222 sub --nkey=a.nk ">"`

Publish a message using `nats -s nats://localhost:4222 pub --nkey=b.nk foo nkey` the subscriber should receive it.

When the nats-server was started with `-v` tracing, you can see the signature in the `CONNECT` message (formatting added manually):

```
[95184] 2020/10/26 12:15:44.350577 [TRC] [::1]:55551 - cid:2 - <<- [CONNECT {
  "echo": true,
  "headers": true,
  "lang": "go",
  "name": "NATS CLI",
  "nkey": "UC435ZYS52HF72E2VMQF4G06CUJ0CHDUUPEBU7XDXW5AQLIC6JZ46P05",
  "no_responders": true,
  "pedantic": false,
  "protocol": 1,
  "sig":
"lopzgs98JBQYyRdw1zT_BoBpSFRDCfTvT4le5MYSKrt0IqGWZ20XhPW1J_zo2_sBod8XaWgQc9oWohWBN0NdDg",
  "tls_required": false,
  "verbose": false,
  "version": "1.11.0"
}]
```

On connect, clients are instantly sent the nonce to sign as part of the `INFO` message (formatting added manually). Since `telnet` will not authenticate, the server closes the connection after hitting the [authorization](#) timeout.

```
> telnet localhost 4222
Trying ::1...
Connected to localhost.
Escape character is '^]'.
INFO {
  "auth_required": true,
  "client_id": 3,
  "client_ip": "::1",
  "go": "go1.14.1",
  "headers": true,
  "host": "0.0.0.0",
```

```
"max_payload": 1048576,  
"nonce": "-QPTE1Jsk8kI3rE",  
"port": 4222,  
"proto": 1,  
"server_id": "NBSHIXACRHU0DC4FY2Z30YXSZSRUBRH6VWIKQNGVPK0TA7H4YTXWJRT0",  
"server_name": "NBSHIXACRHU0DC4FY2Z30YXSZSRUBRH6VWIKQNGVPK0TA7H4YTXWJRT0",  
"version": "2.2.0-beta.26"  
}  
-ERR 'Authentication Timeout'  
Connection closed by foreign host.
```

Key Takeaways

- NKEYS are a secure way to authenticate clients,
- Private keys are never accessed or stored by the NATS server,
- The public key still needs to be configured in NATS server.

JSON Web Tokens (JWT)

Motivation for JWT

In a large organization the centralized configuration approach can lead to less flexibility and more resistance to change when controlled by one entity. Alternatively, instead of operating one infrastructure, it can be deployed more often (say per team) thus making import/export relationships harder as they have to bridge separate systems. In order to make accounts truly powerful, they should ideally be configured separately from the infrastructure, only constrained by limits. This is similar for user. An account contains the user but this relationship could be a reference as well, such that alterations to user do not alter the account. Users of the same account should be able to connect from anywhere in the same infrastructure and be able to exchange messages as long as they are in the same authentication domain.

Key Takeaways

- JWT splits a nats-server configuration into separate artifacts manageable by different entities.
- Management of Accounts, Configuration, and Users are separated.
- Accounts do NOT correspond to infrastructure, they correspond to teams or applications.
- Connect to any cluster in the same infrastructure and be able to communicate with all other users in your account.

- Infrastructure and its topology have nothing to do with Accounts and where an Account's User connects from.

Decentralized Authentication/Authorization using JWT

Account and User creation managed as separate artifacts in a decentralized fashion using NKEYs. Relying upon a hierarchical chain of trust between three distinct NKEYs and associated roles:

1. Operator: corresponds to operator of a set of NATS servers in the same authentication domain (entire topology, crossing gateways and leaf nodes),
2. Account: corresponds to the set of a single account's configuration,
3. User: corresponds to one user's configuration.

Each NKEY is referenced, together with additional configuration, in a JWT document. Each JWT has a subject field and its value is the public portion of an NKEY and serves as identity. Names exist in JWT but as of now are only used by tooling, `nats-server` does not read this value. The referenced NKEY's role determines the JWT content:

1. Operator JWTs contain server [configuration](#) applicable throughout all operated NATS servers,
2. Account JWTs contain Account specific [configuration](#) such as exports, imports, limits, and default user permissions,
3. User JWTs contain user specific [configuration](#) such as permissions and limits.

In addition, JWTs can contain settings related to their decentralized nature, such as expiration/revocation/signing. At no point do JWTs contain the private portion of an NKEY, only signatures that can be verified with public NKEY. JWT content can be viewed as public, although its content may reveal which subjects/limits/permissions exist.

Key Takeaways

- JWTs are hierarchically organized in operator, account and user,
- They carry corresponding configuration and the configuration has been dedicated to the decentralized nature of NATS JWT usage.

NATS JWT Hierarchy

Decentralized Chain of Trust

A `nats-server` is configured to trust an operator. Meaning, the Operator JWT is part of its server configuration and requires a restart or `nats-server -s signal reload` once the configuration changed. It is also configured with a way to obtain account JWT in one of three ways (explained below).

Clients provide a User JWT when connecting. An Account JWT is not used by clients talking to a `nats-server`. The clients also possess the private NKEY corresponding to the JWT identity, so that they can prove their identity as described [above](#).

The issuer field of the User JWT identifies the Account, and the `nats-server` then independently obtains the current Account JWT from its configured source. The server can then verify that signature on the User JWT was issued by an NKEY of the claimed Account, and in turn that the Account has an issuer of the Operator and that an NKEY of the Operator signed the Account JWT. The entire three-level hierarchy is verified.

Obtain an Account JWT

To obtain an Account JWT, the `nats-server` is configured with one of three [resolver](#) types. Which one to pick depends upon your needs:

- [mem-resolver](#): Very few or very static accounts
 - You are comfortable changing the server config if the operator or any accounts changed,
 - You can generate a user programmatically using NKEYs and a JWT library (more about that later),
 - Users do not need to be known by `nats-server`.
- [url-resolver](#): Very large volume of accounts
 - Same as `mem-resolver`, except you do not need to modify the server configurations when accounts are added or changed,
 - Changes to the operator still require a server reloading (only a few operations require that),
 - Will download Accounts from a web server
 - Allows for easy publication of account JWTs programmatically generated using NKEYs and the JWT library.
 - The `nats-account-server` is such a webserver. When set up correctly, it will inform `nats-server` of Account JWT changes.
 - Depending on configuration, requires read and/or write access to persistent storage.
- `nats-resolver`: Same as `url-resolver`, just uses NATS instead of http
 - No separate binary to run/config/monitor,
 - Easier clustering when compared to `nats-account-server`. Will eventually converge on the union of all account JWTs known to every participating `nats-server`,
 - Requires persistent storage in the form of a directory for `nats-server` to *exclusively* write to (it can be on a shared Network File System, but the directories themselves can not be shared between servers),
 - Optionally, directly supports Account JWT removal,

- Between `nats-resolver` and `url-resolver`, the `nats-resolver` is the clear recommendation.

JWT `nats-resolver` is recommended to use in production environment. With JWT `nats-resolver`, you can manage huge accounts and users without server reloading. But before adopting JWT `nats-resolver`, make sure you understand correctly how it works. You can make use of static account settings(probably with NKEYs) and `memory-resolver` as the necessary steps forwarding JWT `nats-resolver` fully understanding.

JWT and Chain of Trust Verification

Each JWT document has a subject(`sub`) it represents. This is the public identity NKEY represented by the JWT document. JWT documents contain an issued at (`iat`) time of signing. This time is in seconds since Unix epoch. It is also used to determine which of two JWTs for the same subject is more recent. Furthermore JWT documents have an issuer, this may be an (identity) NKEY or a dedicated signing NKEY of an item one level above it in the trust hierarchy. A key is a signing key if it is listed as such in the JWT (above). Signing NKEYs adhere to same NKEY roles and are additional keys that unlike identity NKEY may change over time. In the hierarchy, signing keys can only be used to sign JWT for the role right below them. User JWTs have no signing keys for this reason. To modify one role's set of signing keys, the identity NKEY needs to be used.

Each JWT is signed as below:

```
jwt.sig = sign(hash(jwt.header + jwt.body), private-key(jwt.issuer))
(jwt.issuer is part of jwt.body)
```

If a JWT is valid, the JWT above it is validated as well. If all of them are valid, the chain of trust between them is tested top down as follows:

Type	Trust Rule	Obtained
Operator	<code>jwt.issuer == jwt.subject (self signed)</code>	configured to trust
Account	<code>jwt.issuer == trusted issuing operator (signing/identity) key</code>	configured to obtain
User	<code>jwt.issuer == trusted issuing account (signing/identity) key && jwt.issuedAt > issuing account revocations[jwt.subject]</code>	provided on connect

This is a conceptual view. While all these checks happen, the results of earlier evaluations might be cached: if the Operator/Account is trusted already and the JWT did not change since, then there is no reason to re-evaluate.

Below are examples of decoded JWT (`iss == issuer`, `sub == subject`, `iat == issuedAt`):

```
nsc describe operator --json
```

```
{
  "iat": 1603473819,
  "iss": "0BU505FJ324UDPRBIVRGF7CNE0HGLPS7EYPBTVQZKSBHIIZIB6HD66JF",
  "jti": "57BWRLW67I6JTVYMQAZQF54G2G37DJB5WG5IFIPVYI4PEYNX57ZQ",
  "name": "DEMO",
  "nats": {
    "account_server_url": "nats://localhost:4222",
    "system_account": "AAAXAUVSGK7TCRHFIRAS4SYXVJ76EWD MNXZM6ARFGXP7BASNDGLKU7A5"
  },
  "sub": "0BU505FJ324UDPRBIVRGF7CNE0HGLPS7EYPBTVQZKSBHIIZIB6HD66JF",
  "type": "operator"
}
```

```
nsc describe account -n demo-test --json:
```

```
{
  "iat": 1603474600,
  "iss": "0BU505FJ324UDPRBIVRGF7CNE0HGLPS7EYPBTVQZKSBHIIZIB6HD66JF",
  "jti": "CZDE4PM7MGFNYHRZSE6INTP6QDU4DSLACVHPQFA7XEYNJT6R6LLQ",
  "name": "demo-test",
  "nats": {
    "limits": {
      "conn": -1,
      "data": -1,
      "exports": -1,
      "imports": -1,
      "leaf": -1,
      "payload": -1,
      "subs": -1,
      "wildcards": true
    }
  },
  "sub": "ADKGAJU55CHY0IF5H432K2Z2ME3NPSJ5S3VY5Q42Q30TY0CYRRG7W0WV",
  "type": "account"
}
```

```
nsc describe user -a demo-test -n alpha --json:
```

```
{
  "iat": 1603475001,
```

```
"iss": "ADKGAJU55CHY0IF5H432K2Z2ME3NPSJ5S3VY5Q42Q30TY0CYRRG7W0WV",
"jti": "G00PXCFDWVMEU3U6I6MT344Z56MGBYIS42GDXMUXDFA3NYDR2RUQ",
"name": "alpha",
"nats": {
  "pub": {},
  "sub": {}
},
"sub": "UC56LV5NNMP5FURQZ7HZTGWCRRTWSMHZNNELQMHDH3DCYNGX57B2TN6",
"type": "user"
}
>
```

Obtain a User JWT - Client Connect

When a client connects, the steps below have to succeed. The following nats-server configuration is used (for ease of understanding, we are using url-resolver):

```
operator: ./trustedOperator.jwt
resolver: URL(http://localhost:9090/jwt/v1/accouts/)
```

1. Client connects and the `nats-server` responds with `INFO` ([identical to NKEYs](#)) and a containing nonce.

```
> telnet localhost 4222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
INFO {
  "auth_required": true,
  "client_id": 5,
  "client_ip": "127.0.0.1",
  "go": "go1.14.1",
  "headers": true,
  "host": "localhost",
  "max_payload": 1048576,
  "nonce": "aN9-ZtS7taDoAZk",
  "port": 4222,
  "proto": 1,
  "server_id": "NCIK6FX5MRIEPM EK22YL2ECLIWVJBH2SWFD5EQWSI5XRDQPKZXWKX3VP",
  "server_name": "NCIK6FX5MRIEPM EK22YL2ECLIWVJBH2SWFD5EQWSI5XRDQPKZXWKX3VP",
```

```
"tls_required": true,  
"version": "2.2.0-beta.26"  
}  
Connection closed by foreign host.
```

For ease of use, the NATS CLI uses a creds file that is the concatenation of JWT and private user identity/NKEY.

```
> cat user.creds  
-----BEGIN NATS USER JWT-----  
  
eyJ0eXAiOiJqd3Q3Q0lCJHbGciOiJlZDI1NTE5In0.eyJqdGkiOiJXNkFYSlSS1RHVTNFUk1QM0dSRDdNV0FQT  
zQ2VzQ2Vzc3R1JNMk5SWFFlQ0VRQ0tCRjJRIiwiaWF0IjoxNjAzNDczNzg4LCJpc3MiOiJBQUFYQVZVWU0dLN1  
RDUkhGSVJBUzRTWVhWSjc2RVdETU5YWk02QVJGR1hQN0JBU05ER0xLVTdBNSIsIm5hbWUiOiJzeXMiLCJzdWI  
iOiJvRE5ZMktLUFRjQVBQ0tCRjJRIiwiaWF0IjoxNjAzNDczNzg4LCJpc3MiOiJBQUFYQVZVWU0dLN1RDUkhGSVJBUzRTWVhWSjc2RVdETU5YWk02QVJGR1hQN0JBU05ER0xLVTdBNSIsIm5hbWUiOiJzeXMiLCJzdWI  
cGUiOiJlZDI1NTE5In0.eyJqdGkiOiJXNkFYSlSS1RHVTNFUk1QM0dSRDdNV0FQTzQ2VzQ2Vzc3R1JNMk5SWFFlQ0VRQ0tCRjJRIiwiaWF0IjoxNjAzNDczNzg4LCJpc3MiOiJBQUFYQVZVWU0dLN1RDUkhGSVJBUzRTWVhWSjc2RVdETU5YWk02QVJGR1hQN0JBU05ER0xLVTdBNSIsIm5hbWUiOiJzeXMiLCJzdWI  
RZbXZlZG90dFhNPQI5LsPrR00Q0Slc2tic6e3sn8YM5x0Rt34FryDA  
-----END NATS USER JWT-----  
  
***** IMPORTANT *****  
  
NKEY Seed printed below can be used to sign and prove identity.  
NKEYs are sensitive and should be treated as secrets.  
  
-----BEGIN USER NKEY SEED-----  
SUAZU5G7U0UR7VXQ7DBD5RQTBW5402C0GSXAVIYWVZE4GCZ5C70CZ5JLY  
-----END USER NKEY SEED-----  
  
*****
```

```
> nats -s localhost:4222 "--creds=user.creds" pub "foo" "hello world"
```

2. The Client responds with a `CONNECT` message (formatting added manually), containing a JWT and signed nonce. (output copied from `nats-server` started with `-v`)

```
[98019] 2020/10/26 16:07:53.861612 [TRC] 127.0.0.1:56830 - cid:4 - <<- [CONNECT {  
  "echo": true,  
  "headers": true,  
  "jwt":  
  "eyJ0eXAiOiJqd3Q3Q0lCJHbGciOiJlZDI1NTE5In0.eyJqdGkiOiJXNkFYSlSS1RHVTNFUk1QM0dSRDdNV0FQT  
  TzQ2VzQ2Vzc3R1JNMk5SWFFlQ0VRQ0tCRjJRIiwiaWF0IjoxNjAzNDczNzg4LCJpc3MiOiJBQUFYQVZVWU0dLN1  
  1RDUkhGSVJBUzRTWVhWSjc2RVdETU5YWk02QVJGR1hQN0JBU05ER0xLVTdBNSIsIm5hbWUiOiJzeXMiLCJzdWI  
  RZbXZlZG90dFhNPQI5LsPrR00Q0Slc2tic6e3sn8YM5x0Rt34FryDA"
```

```
Ii0iJvRE5ZMktLUFRJQVBQTK90T0xBVE5SWLBHTVBMTkZXsFFQS1VYSjZBMllUQTQ3Tk41Vk5GSU80NSIsInR
5cGUiOiJlc2VyIiwibmF0cyI6eyJwdWIiOnt9LCJzdWIiOnt9fX0. ae30vcapjQgbXhI2QbgIs32AWr3iBb2U
FRZbXzIg0duFHNPQI5LsprR00QoS1c2tic6e3sn8YM5x0Rt34FryDA",
    "lang": "go",
    "name": "NATS CLI",
    "no_responders": true,
    "pedantic": false,
    "protocol": 1,
    "sig": "VirwM--xq5i2RI9VEQiFYv_6JBs-
IR4o0bypglR7qVxYtXDUtIKIr1qXW_M54iHFB6Afu698J_in5CfBRjuVBg",
    "tls_required": true,
    "verbose": false,
    "version": "1.11.0"
  }
}
```

3. Server verifies if a JWT returned is a user JWT and if it is consistent: `sign.jwt.sig, jwt.issuer) == hash(jwt.header+jwt.body)` (issuer is part of body),
4. Server verifies if nonce matches JWT.subject, thus proving client's possession of private user NKEY,
5. Server either knows referenced account or downloads it from `http://localhost:9090/jwt/v1/accouts/AAAXAUVSGK7TCRHFIRAS4SYXVJ76EWD MNXZM6ARFGXP7BASNDGLKU7A5`,
6. Server verifies downloaded JWT is an account JWT and if it is consistent: `sign.jwt.sig, jwt.issuer) == hash(jwt.header+jwt.body)` (issuer is part of body),
7. Server verifies if an account JWT issuer is in configured list of trusted operator keys (derived from operator JWT in configuration),
8. Server verifies that a user JWT subject is not in the account's revoked list, or if `jwt.issuedAt` field has a higher value,
9. Server verifies that a user JWT issuer is either identical to the account JWT subject or part of the account JWT signing keys,
10. If all of the above holds true, the above invocation will succeed, only if the user JWT does not contain permissions or limits restricting the operation otherwise

```
> nats -s localhost:4222 "--creds=user.creds" pub "foo" "hello world"
> 16:56:02 Published 11 bytes to "foo"
```

11. Output if `user.creds` were to contain a JWT where the maximum message payload is limited to 5 bytes

```
> nats -s localhost:4222 "--creds=user.creds" pub "foo" "hello world"
nats: error: nats: Maximum Payload Violation, try --help
>
```

Key Takeaways

- JWTs are secure,
- JWTs carry configuration appropriate to their role as Operator/Accounts/User,
- JWTs provide a basis for operating one single NATS infrastructure which serves separate, yet optionally connected, entities,
- Account resolvers are a way to obtain unknown Account JWTs,
- On connect clients *provide* only the **User** JWT and *use* the NKEY for the JWT to authenticate,
- JWTs can be issued programmatically.

Deployment Models Enabled by Chain of Trust

Depending on which entity has access to private Operator/Account identity or signing NKEYs, different deployment models are enabled. When picking one, it is important to pick the simplest deployment model that enables what you need it to do. Everything beyond just results in unnecessary configuration and steps.

1. Centralized config: one (set of) user(s) has access to all private operator and account NKEYs,
Administrators operating the shared infrastructure call all the shots
2. Decentralized config (with multiple `nsc` environments, explained later):
 1. Administrator/Operator(s) have access to private operator NKEYs to sign accounts.
By signing or not signing an account JWT, Administrators can enforce constraints (such as limits).
 2. Other sets of users (teams) have access to their respective private account identity/signing NKEYs and can issue/sign a user JWT.
This can also be used by a single entity to not mix up `nsc` environments as well.
3. Self-service, decentralized config (shared dev cluster):
Is similar to 2, but sets of users in 2.1 have access to an operator private signing NKEY.
This allows teams to add/modify their own accounts.
Since administrators give up control over limits, there should be at least one organizational mechanism to prevent unchecked usage.
Administrators operating the infrastructure can add/revoke access by controlling the set of operator signing keys.
4. Mix of the above - as needed: separate sets of users (with multiple `nsc` environments).
For some user/teams the Administrator operates everything.

Signing keys can not only be used by individuals in one or more `nsc` environments, but also by programs facilitating [JWT](#) and [NKEY](#) libraries. This allows the implementation of sign-up services.

- Account signing key enabled on the fly:

- user generation (explained later),
- export activation generation (explained later).
- Operator signing key enables on the fly account generation.

Key Takeaways

- JWTs and the associated chain of trust allows for centralized, decentralized, or self-service account configuration,
- It is important to pick the deployment model that fits your needs, NOT the most complicated one,
- Distributing Operator/Account JWT NKEYs between Administrators and teams enables these deployment models,
- Sign-up services for Accounts/Users can be implemented by programs in possession of the parent type's signing keys,

Accounts Re-visited

A deeper understanding of accounts will help you to best setup NATS JWT based security.

- What entity do accounts correspond to:
Our official suggestion is to scope accounts by application/service offered.
This is very fine grained and will require some configuration.
This is why some users gravitate to accounts per team. One account for all Applications of a team.
It is possible to start out with less granular accounts and as applications grow in importance or scale become more fine grained.
- Compared to file based config, Imports and Exports change slightly.
To control who gets to import an export, activation tokens are introduced.
These are JWTs that an importer can embed.
They comply to similar verification rules as user JWT, thus enabling a `nats-server` to check if the exporting account gave explicit consent.
Due to the use of a token, the exporting account's JWT does not have to be modified for each importing account.
- Updates of JWTs are applied as `nats-server` discover them
 - How this is done depends on the resolver
 - `mem-resolver` require `nats-server --signal reload` to re-read all configured account JWTs,
 - `url-resolver` and `nats-resolver` listen on a dedicated update subject of the system account and applied if the file is valid,
 - `nats-resolver` will also also update the corresponding JWT file and compensate in case the update message was not received due to temporary disconnect.
 - User JWTs only depend on the issuing Account NKEY, they do NOT depend on a particular version of an Account JWT,

- Depending on the change, the internal Account representation will be updated and existing connections re-evaluated.
- The System Account is the account under which `nats-server` offers (administrative) services and monitoring events.

Key Takeaways

- Accounts can be arbitrarily scoped, from Application to Team,
- Account Exports can be restricted by requiring use of activation tokens,
- Receiving a more recent Account JWT causes the nats-server to apply changes and re-evaluate existing connections.

Tooling And Key Management

This section will introduce `nsc` cli to generate and manage operator/accounts/user. Even if you intend to primarily generate your Accounts/User programmatically, in all likelihood, you won't do so for an operator or all accounts. Key Management and how to do so using `nsc` will also be part of this section.

nsc

Environment

`nsc` is a tool that uses the [JWT](#) and [NKEY](#) libraries to create NKEYs (if asked to) and all types of JWT. It then stores these artifacts in separate directories.

It keeps track of the last operator/account used. Because of this, commands do not need to reference operator/accounts but can be instructed to do so (recommended for scripts). It supports an interactive mode when `-i` is provided. When used, referencing accounts/keys is easier.

`nsc env` will show where NKEYS/JWT are stored and what current defaults are. For testing you may want to switch between nsc environments: Changing the (JWT) store directory: `nsc env --store <different folder>` Changing the (NKEY) store directory by having an environment variable set: `export NKEYS_PATH=<different folder>`

Subsequent sections will refer to different environments in context of different [deployment modes](#). As such you can skip over all mentions for modes not of interest to you. The mixed deployment mode is not mentioned and left as an exercise to the reader.

Backup

NKEYS store directory

Possessing NKEYS gives access to the system. Backups should therefore best be offline and access to them should be severely restricted. In cases where regenerating all/parts of the operator/accounts is not an option, signing NKEYs must be used and identity NKEYs **should be archived and then removed** from the original store directory, so that in the event of a data breach you can recover without a flag-day change-over of identities. Thus, depending on your scenario, relevant identity NKEYS need to only exist in very secure offline backup(s).

JWT store directory

The store directory contains JWTs for operators, accounts, and users. It does not contain private keys. Therefore it is ok to back these up or even store them in a VCS such as git. But be aware that depending on content, JWT may reveal which permissions/subjects/public-nkeys exist. Knowing the content of a JWT does not grant access; only private keys will. However, organizations may not wish to make those public outright and thus have to make sure that these external systems are secured appropriately.

When restoring an older version, be aware that:

- All changes made since will be lost, specifically revocations may be undone,
- Time has moved on and thus JWTs that were once valid at the time of the backup or commit may be expired now. Thus you may have to edit them to match your expectations again,
- NKEYS are stored in a separate directory, so not to restore a JWT for which the NKEY has been deleted since:
 - Either keep all keys around; or
 - Restore the NKEY directory in tandem.

Names in JWT

JWTs allow you to specify names. But names do NOT represent an identity, they are only used to ease referencing of identities in our tooling. At no point are these names used to reference each other, instead, the public identity NKEY is used for that. The `nats-server` does not read them at all. Because names do not relate to identity, they may collide. Therefore, when using `nsc`, these names need to be kept unique.

Setup an Operator

Create/Edit Operator - Operator Environment - All Deployment modes

Create operator with system account and system account user:

```
nsc add operator -n <operator-name> --sys
```

The command `nsc edit operator [flags]` can subsequently be used to modify the operator. For example if you are setting the account server url (used by `url-resolver` and `nats-resolver`), `nsc` does not require them being specified on subsequent commands. `nsc edit operator --account-jwt-server-url "nats://localhost:4222"`

⚠ Note that if you update an operator JWT that is installed on a server you will need to manually update the operator JWT and reload the server. While `nsc` is able to update accounts, it never updates the operator.

We always recommend using signing keys for an operator. Generate one for an operator (`-o`) and store it in the key directory (`--store`). The output will display the public portion of the signing key, use that to assign it to the operator (`--sk 0...`). `nsc generate nkey -o --store` followed by `nsc edit operator --sk 0B7420V630E2U55Z7UZHUB2DUVGQHRA5QVR4RZU6NXN0KBKJGKF6WRTZ`. To pick the operator signing key for account generation, provide the `-i` option when doing so.

The system account is the account under which `nats-server` offers system services as will be explained below in the [system-account](#) section. To access these services a user with credentials for the system account is needed. Unless this user is restricted with appropriate permissions, this user is essentially the admin user. They are created like any other user.

For cases where signing keys are generated and immediately added `--sk generate` will create an NKEY on the fly and assign it as signing NKEY.

Import Operator - Non Operator/Administrator Environment - Decentralized/Self Service Deployment Modes

In order to import an Operator JWT, say the one just created, into a separate nsc environment maintained by a different entity/team, the following has to happen:

1. Obtain the operator JWT using: `nsc describe operator --raw` and store the output in a file named `operator.jwt`. The option `--raw` causes the raw JWT to be emitted,
2. Exchange that file or it's content in any way you like, email works fine (as there are no credentials in the JWT),

3. Import the operator JWT into the second environment with: `nsc add operator -u operator.jwt`.

If the operator should be changed and an update is required, simply repeat these steps but provide the `--force` option in the last step. This will overwrite the stored operator JWT.

Import Operator - Self Service Deployment Modes

In addition to the [previous step](#), self service deployments require an operator signing key and a system account user. Ideally you would want an operator signing key per entity to distribute a signing key too. Simply repeat the command shown [earlier](#) but:

1. Perform `nsc generate nkey -o --store` in this environment instead,
2. Exchange the public key with the Administrator/Operator via a way that assures you sent the public key and not someone else's,
3. Perform `nsc edit operator --sk` in the operator environment,
4. Refresh the operator JWT in this environment by performing the [import steps using](#) `--force`

To import the system account user needed for administrative purposes as well as monitoring, perform these steps:

1. Perform `nsc describe account -n SYS --raw` and store the output in a file named `SYS.jwt`. The option `-n` specifies the (system) account named `SYS`.
2. Exchange the file,
3. Import the account `nsc import account --file SYS.jwt`,
4. Perform `nsc generate nkey -u --store` in this environment,
5. Exchange the public key printed by the command with the Administrator/Operator via a way that assures you sent the public key and not someone else's,
6. Create a system account user named (`-n`) any way you like (here named `sys-non-op`) providing (`-k`) the exchanged public key `nsc add user -a SYS -n sys-non-op -k UDJKPL7H6QY4KP4LISNHENU6Z434G6RLDEXL2C64YZXDABNCE0AZ4YY2` in the operator environment. (`-a` references the Account `SYS`.),
7. If desired edit the user,
8. Export the user `nsc describe user -a SYS -n sys-non-op --raw` from the operator environment and store it in a file named `sys.jwt`. (`-n` references the user `sys-non-op`),
9. Exchange the file,
10. Import the user in this environment using `nsc import user --file sys.jwt`

As a result of these operations, your operator environment should have these keys and signing keys:

```
nsc list keys --all
```

```

+-----+
---+
|
Keys                               |
+-----+-----+-----+
+-----+
| Entity          | Key                               | Signing Key |
Stored |
+-----+-----+-----+
+-----+
| DEMO            | OD5FHU4LXGDSGDH07UNRMLW6I36QX5VPJXRQHFHMRUIKSHOPEDSHVPBB |
*      |
| DEMO            | OBYAIG4T4PVR6GVYDERN74RRW7VBKRWB7I7ULLMM6BRHUID4AAQL7SGA | *
|      |
| ACC             | ADRB4JJYFDLWKIMX4DH6MX2DMKA3TENJWGMNVM5ILYLZTT6BN7QIF5ZX |
|      |
| SYS             | AAYVLZJC2ULKSH5HNSKMIKFMCEHCNU5V0V5KG56IRL7ENHLBUGZ27CZT |
*      |
| sys             | UBVZYLLCAFMHBXBUDKKKFKH62T4AW7Q5MAAE3R3KKAIKRCZNYITZPDQZ3 |
*      |
| sys-non-op     | UDJKPL7H6QY4KP4LISNHENU6Z434G6RLDEXL2C64YZXDABNCE0AZ4YY2 |
|      |
+-----+-----+-----+
+-----+

```

And your account should have the following ones:

```
nsc list keys --all
```

```

+-----+
---+
|
Keys                               |
+-----+-----+-----+
+-----+
| Entity          | Key                               | Signing Key |
Stored |
+-----+-----+-----+
+-----+
| DEMO            | OD5FHU4LXGDSGDH07UNRMLW6I36QX5VPJXRQHFHMRUIKSHOPEDSHVPBB |

```

```

|          |
| DEMO      | OBYAIG4T4PVR6GVYDERN74RRW7VBKRWB7I7ULLMM6BRHUID4AAQL7SGA | *
*          |
| SYS       | AAYVLZJC2ULKSH5HNSKMIKFMCEHCNU5V0V5KG56IRL7ENHLBUGZ27CZT |
|          |
| sys-non-op | UDJKPL7H6QY4KP4LISNHENU6Z434G6RLDEXL2C64YZXDABNCE0AZ4YY2 |
*          |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+

```

Between the two outputs, compare the `Stored` column.

Alternatively if the administrator is willing to exchange private keys and the exchange can be done securely, a few of these steps fall away. The signing key and system account user can be generated in the administrator/operator environment, omitting `--store` to avoid unnecessary key copies. Then the public/private signing NKEYS are exchanged together with the system account user as creds file. A creds file can be generated with `nsc generate creds -a SYS -n sys-non-op` and imported into this environment with `nsc import user --file sys.jwt`. If the signing key is generated before the operator is imported into this environment, operator update falls away.

Setup an Account

Create/Edit Account - All Environments - All Deployment modes

Create an account as follows:

```
nsc add account -n <account name> -i
```

In case you have multiple operator signing keys `-i` will prompt you to select one. `nsc edit account [flags]` can subsequently be used to modify the account. (Edit is also applicable to the system account)

Similar to the operator signing keys are recommended. Generate signing key for an account (`-a`) and store it in the key directory maintained by nsc (`--store`) The output will display the public portion of the signing key, use that to assign it to the account (`--sk A...`) `nsc generate nkey -a --store` `nsc edit account --sk ACW2QC262CIQUX4ACG00S5XLKSZ2BY2QFBAA0F3V0P7AWAVI37E20QZX` To pick the signing key for user generation, provide the `-i` option when doing so.

Export Account - Non Operator/Administrator Environment - Decentralized Deployment Modes

In this mode, the created account is self-signed. To have it signed by the operator perform these steps:

1. In this environment export the created account as a JWT like this `nsc describe account -n <account name> --raw`.
Store the output in a file named `import.jwt`.
2. Exchange the file with the Administrator/Operator via a way that assures it is your JWT and not someone elses.
3. In the operator environment import the account with `nsc import account --file import.jwt`.
This step also re-signs the JWT so that it is no longer self-signed.
4. The Administrator/operator can now modify the account with `nsc edit account [flags]`

If the account should be changed and an update is required, simply repeat these steps but provide the `--force` option during the last step. This will overwrite the stored account JWT.

Export Account - Non Operator/Administrator Environment - Self Service Deployment Modes

This environment is set up with a signing key, thus the account is already [created properly signed](#). The only step that is needed is to push the Account into the NATS network. However, this depends on your ability to do so. If you have no permissions, you have to perform the same steps as for the [decentralized deployment mode](#). The main difference is that upon import, the account won't be re-signed.

Publicize an Account with Push - Operator Environment/Environment with push permissions - All Deployment Modes

How accounts can be publicized wholly depends on the resolver you are using:

- [mem-resolver](#): The operator has to have all accounts imported and generate a new config,
- [url-resolver](#): `nsc push` will send an HTTP POST request to the hosting webserver or `nats-account-server`,
- [nats-resolver](#): Every environment with a system account user that has permissions to send properly signed account JWT as requests to:
 - `$SYS.REQ.CLAIMS.UPDATE` can upload and update all accounts. Currently, `nsc push` uses this subject.
 - `$SYS.REQ.ACCOUNT.*.CLAIMS.UPDATE` can upload and update specific accounts.

`nsc generate config <resolver-type>` is an utility that generates the relevant NATS config. Where `<resolver-type>` can be `--mem-resolver` or `--nats-resolver` for the corresponding resolver. Typically the generated output is stored in a file that is then

[included](#) by the NATS config. Every server within the same authentication domain needs to be configured with this configuration.

nats-resolver setup and push example - Operator Environment/Environment with push permissions - All Deployment Modes

This is a quick demo of the nats-based resolver from operator creation to publishing a message. Please be aware that the ability to push only relates to permissions to do so and does not require an account keys. Thus, how accounts to be pushed into the environment (outright creation/import) does not matter. For simplicity, this example uses the operator environment.

Operator Setup:

```
nsc add operator -n DEMO --sys
```

```
[ OK ] generated and stored operator key
"ODHUV0UVUA3XIBV25XSQS2NM2UN4IKJYLAMCGLWRFAV7F7KUWADCM4K6"
[ OK ] added operator "DEMO"
[ OK ] created system_account: name:SYS
id:AA6W5MRDIFIQWE6UE6D4YWQT5L4YZG7ZRHSKYCPF2VIEMUHRZH3VQZ27
[ OK ] created system account user: name:sys
id:UABM73CE5F3ZYFNC3ZD0DAF7GIB62W2WXV5D0LMYLGEW4MEHYBC46PN4
[ OK ] system account user creds file stored in
`~/test/demo/env1/keys/creds/DEMO/SYS/sys.creds`
```

```
nsc edit operator --account-jwt-server-url nats://localhost:4222
```

```
[ OK ] set account jwt server url to "nats://localhost:4222"
[ OK ] edited operator "DEMO"
```

Inspect the setup:

```
nsc list keys --all
```

```
+-----+
|                                     Keys                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Entity | Key                                     | Signing Key | Stored |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

DEMO	ODHUV0UVUA3XIBV25XSQS2NM2UN4IKJYLAMCGLWRF7F7KUWADCM4K6	*	
SYS	AA6W5MRDIFIQWE6UE6D4YWQT5L4YZG7ZRHSKYCPF2VIEMUHRZH3VQZ27	*	
sys	UABM73CE5F3ZYFNC3ZD0DAF7GIB62W2WXV5D0LMYLGEW4MEHYBC46PN4	*	
+-----+-----+-----+-----+			

nsc describe operator:

+-----+	
Operator Details	
+-----+	
Name	DEMO
Operator ID	ODHUV0UVUA3XIBV25XSQS2NM2UN4IKJYLAMCGLWRF7F7KUWADCM4K6
Issuer ID	ODHUV0UVUA3XIBV25XSQS2NM2UN4IKJYLAMCGLWRF7F7KUWADCM4K6
Issued	2020-11-04 19:25:25 UTC
Expires	
Account JWT Server	nats://localhost:4222
System Account	AA6W5MRDIFIQWE6UE6D4YWQT5L4YZG7ZRHSKYCPF2VIEMUHRZH3VQZ27
+-----+	

nsc describe account:

+-----+	
Account Details	
+-----+	
Name	SYS
Account ID	AA6W5MRDIFIQWE6UE6D4YWQT5L4YZG7ZRHSKYCPF2VIEMUHRZH3VQZ27
Issuer ID	ODHUV0UVUA3XIBV25XSQS2NM2UN4IKJYLAMCGLWRF7F7KUWADCM4K6
Issued	2020-11-04 19:24:41 UTC
Expires	
+-----+	
Max Connections	Unlimited
Max Leaf Node Connections	Unlimited
Max Data	Unlimited
Max Exports	Unlimited
Max Imports	Unlimited
Max Msg Payload	Unlimited
Max Subscriptions	Unlimited
Exports Allows Wildcards	True
+-----+	
Imports	None

Exports	None	
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+		

Generate the config and start the server in the background. Also, inspect the generated config. It consists of the mandatory operator, explicitly lists the system account and corresponding JWT:

```
nsc generate config --nats-resolver > nats-res.cfg
nats-server -c nats-res.cfg --addr localhost --port 4222 &
```

```
[2] 30129
[30129] 2020/11/04 14:30:14.062132 [INF] Starting nats-server version 2.2.0-beta.26
[30129] 2020/11/04 14:30:14.062215 [INF] Git commit [not set]
[30129] 2020/11/04 14:30:14.062219 [INF] Using configuration file: nats-res.cfg
[30129] 2020/11/04 14:30:14.062220 [INF] Trusted Operators
[30129] 2020/11/04 14:30:14.062224 [INF]   System   : ""
[30129] 2020/11/04 14:30:14.062226 [INF]   Operator: "DEMO"
[30129] 2020/11/04 14:30:14.062241 [INF]   Issued   : 2020-11-04 14:25:25 -0500 EST
[30129] 2020/11/04 14:30:14.062244 [INF]   Expires  : 1969-12-31 19:00:00 -0500 EST
[30129] 2020/11/04 14:30:14.062652 [INF] Managing all jwt in exclusive directory
/demo/env1/jwt
[30129] 2020/11/04 14:30:14.065888 [INF] Listening for client connections on localhost:4222
[30129] 2020/11/04 14:30:14.065896 [INF] Server id is
NBQ6AG5YIRC6PRCUPCAUSVCSCQWAAWW2XQXIM6UPW5AFPGZBUKZJTRRS
[30129] 2020/11/04 14:30:14.065898 [INF] Server name is
NBQ6AG5YIRC6PRCUPCAUSVCSCQWAAWW2XQXIM6UPW5AFPGZBUKZJTRRS
[30129] 2020/11/04 14:30:14.065900 [INF] Server is ready
>
```

Add an account and a user for testing:

```
nsc add account -n TEST
```

```
[ OK ] generated and stored account key
"ADXDDDR2QJNNOSZZX44C2HYBPRUIPJSQ5J3YG2X0U00E0POBNMMFLAIU"
[ OK ] added account "TEST"
```

```
nsc add user -a TEST -n foo
```

```
[ OK ] generated and stored user key
"UA62PGBNKKQQWDTILKP5U4LYUYF3B6NQHPNHL56IZIPPQH6A7XSRWE2"
```

```
[ OK ] generated user creds file `/DEMO/TEST/foo.creds`  
[ OK ] added user "foo" to account "TEST"
```

Without having pushed the account the user can't be used yet.

```
nats -s nats://localhost:4222 pub --creds=/DEMO/TEST/foo.creds "hello" "world"
```

Doesn't work

```
nats: error: read tcp 127.0.0.1:60061->127.0.0.1:4222: i/o timeout, try --help  
[9174] 2020/11/05 16:49:34.331078 [WRN] Account  
[ADI4H2XRYMT5ENVBBS3UKYC2FBLGB3NF4VV5L57HUZI04AMYR0B4LMYF] fetch took 2.000142625s  
[9174] 2020/11/05 16:49:34.331123 [WRN] Account fetch failed: fetching jwt timed out  
[9174] 2020/11/05 16:49:34.331182 [ERR] 127.0.0.1:60061 - cid:5 - "v1.11.0:go:NATS CLI Version  
development" - authentication error  
[9174] 2020/11/05 16:49:34.331258 [WRN] 127.0.0.1:60061 - cid:5 - "v1.11.0:go:NATS CLI Version  
development" - Readloop processing time: 2.000592801s
```

Push the account, or push all accounts:

```
nsc push -a TEST
```

```
[ OK ] push to nats-server "nats://localhost:4222" using system account "SYS" user "sys":  
  [ OK ] push TEST to nats-server with nats account resolver:  
    [ OK ] pushed "TEST" to nats-server  
NBQ6AG5YIRC6PRCUPCAUSVCSCQWAAWW2XQXIM6UPW5AFPGZBUKZJTRRS: jwt updated  
  [ OK ] pushed to a total of 1 nats-server
```

```
nsc push --all
```

```
[ OK ] push to nats-server "nats://localhost:4222" using system account "SYS" user "sys":  
  [ OK ] push SYS to nats-server with nats account resolver:  
    [ OK ] pushed "SYS" to nats-server  
NBENVYIBPNQGYVP32Y3P6WLGB0ISORNAZYHA6SCW6LTBE42ORTIQMWHX: jwt updated  
  [ OK ] pushed to a total of 1 nats-server  
  [ OK ] push TEST to nats-server with nats account resolver:  
    [ OK ] pushed "TEST" to nats-server  
NBENVYIBPNQGYVP32Y3P6WLGB0ISORNAZYHA6SCW6LTBE42ORTIQMWHX: jwt updated  
  [ OK ] pushed to a total of 1 nats-server
```

For the NATS resolver, each `nats-server` that responds will be listed. In case you get fewer responses than you have servers or a server reports an error, it is best practice to resolve this issue and retry. The NATS resolver will gossip missing JWTs in an eventually consistent way. Servers without a copy will perform a lookup from servers that do. If during an initial push only one server responds there is a window where this server goes down or worse, loses its disk. During that time the pushed account is not available to the network at large. Because of this, it is important to make sure that initially, more servers respond than what you are comfortable with losing in such a way at once.

Once the account is pushed, its user can be used:

```
nats -s nats://localhost:4222 pub --creds=/DEMO/TEST/foo.creds "hello" "world"
```

Setup User

Create/Edit Account - All Environments - All Deployment modes

Create a user as follows: `nsc add user --account <account name> --name <user name> -i` `nsc edit user [flags]` can subsequently be used to modify the user. In case you have multiple account signing keys, for either command, `-i` will prompt you to select one.

In case you generate a user on behalf of another entity that has no nsc environment, you may want to consider not exchanging the NKEY. 1. To do this, have the other entity generate a user NKEY pair like this: `nsc generate nkey -u` (`--store` is omitted so as to not have an unnecessary copy of the key) 2. Exchange the public key printed by the command via a way that assures what is used is not someone else's. 3. Create the user by providing (`-k`) the exchanged public key `nsc add user --account SYS -n sys-non-op -k UDJKPL7H6QY4KP4LISNHENU6Z434G6RLDEXL2C64YZXDABNCE0AZ4YY2` in your environment. ([system account user example](#)) 4. If desired edit the user 5. Export the user `nsc describe user --account SYS -n sys-non-op --raw` from your environment and store the output in a JWT file. 6. Exchange the JWT file 7. Use the JWT file and the NKEY pair in your application.

Automated sign up services - JWT and NKEY libraries

`nsc` essentially uses the [NKEY](#) and [JWT](#) libraries to generate operator/accounts/users. You can use these libraries to generate the necessary artifacts as well. Because there is only one, generating the operator this way makes little sense. Accounts only if you need them dynamically, say for everyone of your customer. Dynamically provision user and integrate that process with your existing infrastructure, say LDAP, is the most common use case for these libraries.

The next sub sections demonstrate dynamic user generation. The mechanisms shown are applicable to dynamic account creation as well. For dynamic user/account creation, signing keys are highly recommended.

By generating users or accounts dynamically, it becomes YOUR RESPONSIBILITY to properly authenticate incoming requests for these users or accounts

For sign up service issued JWTs, ALWAYS set the SHORTEST POSSIBLE EXPIRATION

Simple user creation

This example illustrates the linear flow of the algorithm and how to use the generated artifacts. In a real world application you would want this algorithm to be distributed over multiple processes. For simplicity of the examples, keys may be hard coded and error handling is omitted.

```
func GetAccountSigningKey() nkeys.KeyPair {
    // Content of the account signing key seed can come from a file or an environment variable
    as well
    accSeed := []byte("SAAJGCAHPHHM6AVJJWQ2YAS3I4NETXMWVQSTCQMJ7VVTGAJF5UCN3IX7J4")
    accountSigningKey, err := nkeys.ParseDecoratedNKey(accSeed)
    if err != nil {
        panic(err)
    }
    return accountSigningKey
}

func RequestUser() {
    // Setup! Obtain the account signing key!
    accountPublicKey := GetAccountPublicKey()
    accountSigningKey := GetAccountSigningKey()
    userPublicKey, userSeed, userKeyPair := generateUserKey()
    userJWT := generateUserJWT(userPublicKey, accountPublicKey, accountSigningKey)
    // userJWT and userKeyPair can be used in conjunction with this nats.Option
    var jwtAuthOption nats.Option
    jwtAuthOption = nats.UserJWT(func() (string, error) {
        return userJWT, nil
    },
    func(bytes []byte) ([]byte, error) {
        return userKeyPair.Sign(bytes)
    },
    )
}
```

```

// Alternatively you can create a creds file and use it as nats.Option
credsContent, err := jwt.FormatUserConfig(userJWT, userSeed);
if err != nil {
    panic(err)
}
ioutil.WriteFile("my.creds", credsContent, 0644)
jwtAuthOption = nats.UserCredentials("my.creds")
// use in a connection as desired
nc, err := nats.Connect("nats://localhost:4222", jwtAuthOption)
// ...
}

```

Create user NKEY

```

func generateUserKey() (userPublicKey string, userSeed []byte, userKeyPair nkeys.KeyPair) {
    kp, err := nkeys.CreateUser()
    if err != nil {
        return "", nil, nil
    }
    if userSeed, err = kp.Seed(); err != nil {
        return "", nil, nil
    } else if userPublicKey, err = kp.PublicKey(); err != nil {
        return "", nil, nil
    }
    return
}

```

Create user JWT

```

func generateUserJWT(userPublicKey, accountPublicKey string, accountSigningKey nkeys.KeyPair)
(userJWT string) {
    uc := jwt.NewUserClaims(userPublicKey)
    uc.Pub.Allow.Add("subject.foo") // only allow publishing to subject.foo
    uc.Expires = time.Now().Add(time.Hour).Unix() // expire in an hour
    uc.IssuerAccount = accountPublicKey
    vr := jwt.ValidationResults{}
    uc.Validate(&vr)
    if vr.IsBlocking(true) {
        panic("Generated user claim is invalid")
    }
}

```

```

var err error
userJWT, err = uc.Encode(accountSigningKey)
if err != nil {
    return ""
}
return
}

```

Inspect the [user claim](#) for all available properties/limits/permissions to set. When using an [account claim](#) instead, you can dynamically generate accounts. Additional steps are to push the new account as outlined [here](#). Depending on your needs, you may want to consider exchanging the accounts identity NKEY in a similar way that the users key is exchanged in the [next section](#).

Distributed User Creation

As mentioned earlier this example needs to be distributed. This example makes uses of Go channels to encode the same algorithm, uses closures to encapsulate functionalities and Go routines to show which processes exist. Sending and receiving from channels basically illustrates the information flow. To realize this, you can pick `HTTP`, NATS itself etc... (For simplicity, properly closing channels, error handling, waiting for Go routines to finish is omitted.)

The above example did not need authentication mechanisms, `RequestUser` possessed the signing key. How you decide to trust an incoming request is completely up to you. Here are a few examples:

- everyone
- username/password
- 3rd party authentication token

In this example, this logic is encapsulated as placeholder closures `ObtainAuthorizationToken` and `IsTokenAuthorized` that do nothing.

```

func ObtainAuthorizationToken() interface{} {
    // whatever you want, 3rd party token/username&password
    return ""
}

func IsTokenAuthorized(token interface{}) bool {
    // whatever logic to determine if the input authorizes the requester to obtain a user jwt
    return token.(string) == ""
}

```

```

// request struct to exchange data
type userRequest struct {
    UserJWTResponseChan chan string
    UserPublicKey        string
    AuthInfo             interface{}
}

func startUserProvisioningService(isAuthorizedCb func(token interface{}) bool) chan
userRequest {
    userRequestChan := make(chan userRequest) // channel to send requests for jwt to
    go func() {
        accountSigningKey := GetAccountSigningKey() // Setup, obtain account signing key
        for {
            req := <-userRequestChan // receive request
            if !isAuthorizedCb(req.AuthInfo) {
                fmt.Printf("Request is not authorized to receive a JWT, timeout on purpose")
            } else if userJWT := generateUserJWT(req.UserPublicKey, accountSigningKey);
userJWT != "" {
                req.UserJWTResponseChan <- userJWT // respond with jwt
            }
        }
    }()
    return userRequestChan
}

func startUserProcess(userRequestChan chan userRequest, obtainAuthorizationCb func()
interface{}) {
    requestUser := func(userRequestChan chan userRequest, authInfo interface{}) (jwtAuthOption
nats.Option) {
        userPublicKey, _, userKeyPair := generateUserKey()
        respChan := make(chan string)
        // request jwt
        userRequestChan <- userRequest{
            respChan,
            userPublicKey,
            authInfo,
        }
        userJWT := <-respChan // wait for response
        // userJWT and userKeyPair can be used in conjunction with this nats.Option
        jwtAuthOption = nats.UserJWT(func() (string, error) {

```

```

        return userJWT, nil
    },
    func(bytes []byte) ([]byte, error) {
        return userKeyPair.Sign(bytes)
    },
)
// Alternatively you can create a creds file and use it as nats.Option
return
}
go func() {
    jwtAuthOption := requestUser(userRequestChan, obtainAuthorizationCb())
    nc, err := nats.Connect("nats://localhost:4222", jwtAuthOption)
    if err != nil {
        return
    }
    defer nc.Close()
    time.Sleep(time.Second) // simulate work one would want to do
}()
}

func RequestUserDistributed() {
    reqChan := startUserProvisioningService(IsTokenAuthorized)
    defer close(reqChan)
    // start multiple user processes
    for i := 0; i < 4; i++ {
        startUserProcess(reqChan, ObtainAuthorizationToken)
    }
    time.Sleep(5 * time.Second)
}

```

In this example the users NKEY is generated by the requesting process and the public key is sent to the user sign up service. This way the service does not need to know or send the private key. Furthermore, any process receiving the initial request or even response, may have the user JWT but will not be able to proof possession of private NKEY. However, you can have the provisioning service generate the NKEY pair and respond with the NKEY pair and the user JWT. This is less secure but would enable a less complicated protocol where permissible.

User creation using NATS

The [previous example](#) used Go channels to demonstrate data flows. You can use all sorts of protocols to achieve this data flow and pick whatever fits best in your existing infrastructure.

However, you can use NATS for this purpose as well.

Straight forward Setup

You can replace send and receive `<-` with nats publish and subscribe or - for added redundancy on the sign up service - queue subscribe. To do so, you will need connections that enable the sign up service as well as the requestor to exchange messages. The sign up service uses the same connection all of the time and (queue) subscribes to a well known subject. The requestor uses the connection and sends a request to the well known subject. Once the response is received the first connection is closed and the obtained JWT is used to establish a new connection.

Here in lies a chicken and and egg problem. The first connection to request the JWT itself needs credentials. The simplest approach is to set up a different NATS server/cluster that does not require authentication, connect first to cluster 1 and keep requesting the user JWT. Once obtained disconnect from cluster 1 and connect to cluster 2 using the obtained JWT.

Account based Setup

The [earlier setup](#) can be simplified by using accounts instead of separate server/clusters. But a JWT/operator based setup requires JWT authentication. Thus, would be, connections to a different cluster are replaced by connections to the same cluster but different accounts.

- Cluster 1 translates to connections to a `signup` account.
- Cluster 2 translates to connections to accounts to who's signing keys have been used to sign the user JWT. (This happens the first setup as well)

Connections to the `signup` accounts use two kinds of credentials. 1. Sign up service(s) use(s) credentials generated for it/them. 2. All requestors use the same JWT and NKEY, neither of which are used for actual authentication.

- That JWT is probably generated using `nsc` itself.
- Do not use this JWT/NKEY for anything else but contacting the sign up service.
- You want to allow publish only to the well known subject.
- Depending on your deployment you need to back up the account (signing) NKEY so that the account can be re generated without invalidating deployed requestors (which may be hard to replace).

Stamping JWT in languages other than Go

The NKEY library does exist or is incorporated in all languages where NATS supports NKEY. The NATS JWT library on the other hand is written in Go. This may not be your language of choice. Other than encoding JWTs, most of what the that library does is maintain the NATS JWT schema. If you use `nsc` to generate a user as a template for the sign up service and work off of that template you don't need the JWT library. The sample shows how a program that takes an account identity NKEY and account signing NKEY as arguments and outputs a valid creds file.

```
// dotnet add package NATS.NKeys --prerelease
// dotnet add package SimpleBase
using NATS.NKeys;
using System.Security.Cryptography;
using System.Text;

string creds = IssueUserCreds();
Console.WriteLine(creds);

static string IssueUserJwt(string userKeyPub)
{
    // Load account signing key and account identity for
    // the account you wish to issue users for
    const string accSeed = "SAANWFZ3JJNNPERWT3ALE45U7GYT2ZDW6GJUIVPDKUF6GKAX6AISZJMAS4";
    const string accId = "ACV63DGCZG0IT3P5ZA7PQT3KYJ6UDFFHZ7KETHYMDMZ4N44KYAQ2ZZ5F";
    KeyPair accountSigningKey = KeyPair.FromSeed(accSeed);
    string accSigningKeyPub = accountSigningKey.GetPublicKey();

    // Use nsc to create a user any way you like.
    // Export the user as json using:
    // nsc describe user --name <user name> --account <account name> --json
    // Turn the output into a format string and replace values you want replaced.
    // Fields that need to be replaced are:
    // iat (issued at), iss (issuer), sub (subject) and jti (claim hash)
    const string claimFmt = @"{{
    ""iat"": {0},
    ""iss"": ""{1}"" ,
    ""jti"": ""{2}"" ,
    ""name"": ""{3}"" ,
    ""nats"": {{
        ""data"": -1,
        ""issuer_account"": ""{4}"" ,
        ""payload"": -1,
        ""pub"": {{}},
        ""sub"": {{}},
        ""subs"": -1,
        ""type"": ""user"" ,
        ""version"": 2
    }},
    ""sub"": ""{3}""
```

```

}}";

    const string header = @"{
    ""typ"":""JWT"",
    ""alg"":""ed25519-nkey""
}";

    // Issue At time is stored in unix seconds
    long issuedAt = DateTimeOffset.Now.ToUnixTimeSeconds();

    // Generate a claim without jti so we can compute jti off of it
    string claim = string.Format(
        claimFmt,
        issuedAt,
        accSigningKeyPub,
        "", /* blank jti */
        userKeyPub,
        accId);

    // Compute jti, a base32 encoded sha256 hash
    string jti = SimpleBase.Base32.Rfc4648.Encode(
        SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(claim)),
        false);

    // recreate full claim with jti set
    claim = string.Format(
        claimFmt,
        issuedAt,
        accSigningKeyPub,
        jti,
        userKeyPub,
        accId
    );

    // all three components (header/body/signature) are base64url encoded
    string encHeader = ToBase64Url(Encoding.UTF8.GetBytes(header));
    string encBody = ToBase64Url(Encoding.UTF8.GetBytes(claim));

    // compute the signature off of header + body (. included on purpose)
    byte[] sig = Encoding.UTF8.GetBytes($"{encHeader}.{encBody}");
    var signature = new byte[64];

```

```

accountSigningKey.Sign(sig, signature);
string encSig = ToBase64Url(signature);

// append signature to header and body and return it
return $"{encHeader}.{encBody}.{encSig}";
}

static string IssueUserCreds()
{
    // Generate a user NKEY for the new user.
    // The private portion of the NKEY is not needed when issuing the jwt.
    // Therefore generating the key can also be done separately from the JWT.
    // Say by the requester.
    KeyPair userSeed = KeyPair.CreatePair(PrefixByte.User);
    string userKeyPub = userSeed.GetPublicKey();
    string jwt = IssueUserJwt(userKeyPub);

    // return jwt and corresponding user seed as creds
    return $"@-----BEGIN NATS USER JWT-----
{jwt}
-----END NATS USER JWT-----

***** IMPORTANT *****

NKEY Seed printed below can be used to sign and prove identity.
NKEYs are sensitive and should be treated as secrets.

-----BEGIN USER NKEY SEED-----
{userSeed.GetSeed()}
-----END USER NKEY SEED-----

*****";
}

static string ToBase64Url(byte[] input)
{
    var stringBuilder = new StringBuilder(Convert.ToBase64String(input).TrimEnd('='));
    stringBuilder.Replace('+', '-');
    stringBuilder.Replace('/', '_');
    return stringBuilder.ToString();
}

```

If .NET is your language of choice, you can also use the [NATS.Jwt](#) package.

```
// dotnet add package NATS.Jwt --prerelease
using NATS.Jwt;
using NATS.Jwt.Models;
using NATS.NKeys;

const string accSeed = "SAANWFZ3JINNPERWT3ALE45U7GYT2ZDW6GJUIVPDKUF6GKAX6AISZJMAS4";
const string accId = "ACV63DGCZG0IT3P5ZA7PQT3KYJ6UDFFHZ7KETHYMDMZ4N44KYAQ2ZZ5F";

var jwt = new NatsJwt();

// Load account signing key
KeyPair accountSigningKey = KeyPair.FromSeed(accSeed);

// Create a user keypair
KeyPair ukp = KeyPair.CreatePair(PrefixByte.User);
string upk = ukp.GetPublicKey();
NatsUserClaims uc = jwt.NewUserClaims(upk);

// Set to the public ID of the account
uc.User.IssuerAccount = accId;

// Sign the user claims with the account signing key
string userJwt = jwt.EncodeUserClaims(uc, accountSigningKey);

// The seed is a version of the keypair that is stored as text
// and it is considered sensitive information.
string userSeed = ukp.GetSeed();

// Generate a creds formatted file that can be used by a NATS client
string creds = jwt.FormatUserConfig(userJwt, userSeed);
Console.WriteLine(creds);
```

System Account

The system account is the account under which nats-server offer services. To use it either the operator JWT has to specify it, which happens during `nsc init` or when providing `--sys` to `nsc add operator`. Alternatively you can encode it in the server configuration by providing `system_account`

with the public NKEY of the account you want to be the system account:

```
system_account: AAAXAUVSGK7TCRHFIRAS4SYXVJ76EWD MNXZM6ARFGXP7BASNDGLKU7A5
```

It is NOT recommended to use this account to facilitate communication between your own applications. Its sole purpose is to facilitate communication with and between `nats-server`.

Event Subjects

Events are published as they happen. But you MUST NOT rely on a particular ordering or due to the possibility of loss, events matching. Say, `CONNECT` for a client always matching a `DISCONNECT` for the same client. Your subscriber may simply be disconnected when either event happens. Some messages carry aggregate data and are periodically emitted. There, missing a message for one reason or another is compensated by the next one.

Subjects to subscribe on	Description	Repeats
<code>\$SYS.SERVER.<server-id>.SHUTDOWN</code>	Sent when a server shuts down	
<code>\$SYS.SERVER.<server-id>.CLIENT.AUTH.ERR</code>	Sent when client fails to authenticate	
<code>\$SYS.SERVER.<server-id>.STATSZ</code>	Basic server stats	Periodically
<code>\$SYS.ACCOUNT.<account-id>.LEAFNODE.CONNECT</code>	Sent when Leafnode connected	
<code>\$SYS.ACCOUNT.<account-id>.CONNECT</code>	Sent when client connected	
<code>\$SYS.ACCOUNT.<account-id>.DISCONNECT</code>	Sent when Client disconnected	
<code>\$SYS.ACCOUNT.<account-id>.SERVER.CONNS</code>	Sent when an accounts connections change	

The subject `$SYS.SERVER.ACCOUNT.<account-id>.CONNS` is still used but it is recommended to subscribe to it's new name `$SYS.ACCOUNT.<account-id>.SERVER.CONNS`.

Service Subjects

Subjects always available

Subjects to publish requests to	Description	Message Output
<code>\$SYS.REQ.SERVER.PING.STATZ</code>	Exposes the <code>STATZ</code> HTTP monitoring endpoint, each server will respond with one message	Same as HTTP endpoint
<code>\$SYS.REQ.SERVER.PING.VARZ</code>	- same as above for - <code>VARZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.SUBZ</code>	- same as above for - <code>SUBZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.CONNZ</code>	- same as above for - <code>CONNZ</code>	- same as above -

Subjects to publish requests to	Description	Message Output
<code>\$SYS.REQ.SERVER.PING.ROUTEZ</code>	- same as above for - <code>ROUTEZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.GATEWAYZ</code>	- same as above for - <code>GATEWAYZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.LEAFZ</code>	- same as above for - <code>LEAFZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.ACCOUNTZ</code>	- same as above for - <code>ACCOUNTZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.PING.JSZ</code>	- same as above for - <code>JSZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.STATZ</code>	Exposes the <code>STATZ</code> HTTP monitoring endpoint, only requested server responds	Same as HTTP endpoint
<code>\$SYS.REQ.SERVER.<server-id>.VARZ</code>	- same as above for - <code>VARZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.SUBZ</code>	- same as above for - <code>SUBZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.CONNZ</code>	- same as above for - <code>CONNZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.ROUTEZ</code>	- same as above for - <code>ROUTEZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.GATEWAYZ</code>	- same as above for - <code>GATEWAYZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.LEAFZ</code>	- same as above for - <code>LEAFZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.ACCOUNTZ</code>	- same as above for - <code>ACCOUNTZ</code>	- same as above -
<code>\$SYS.REQ.SERVER.<server-id>.JSZ</code>	- same as above for - <code>JSZ</code>	- same as above -
<code>\$SYS.REQ.ACCOUNT.<account-id>.SUBSZ</code>	Exposes the <code>SUBSZ</code> HTTP monitoring endpoint, filtered by account-id.	Same as HTTP endpoint
<code>\$SYS.REQ.ACCOUNT.<account-id>.CONNZ</code>	- same as above for <code>CONNZ</code> -	- same as above -
<code>\$SYS.REQ.ACCOUNT.<account-id>.LEAFZ</code>	- same as above for <code>LEAFZ</code> -	- same as above -
<code>\$SYS.REQ.ACCOUNT.<account-id>.JSZ</code>	- same as above for <code>JSZ</code> -	- same as above -
<code>\$SYS.REQ.ACCOUNT.<account-id>.CONNS</code>	Exposes the event <code>\$SYS.ACCOUNT.<account-id>.SERVER.CONNS</code> as request	- same as above -
<code>\$SYS.REQ.ACCOUNT.<account-id>.INFO</code>	Exposes account specific information similar to <code>ACCOUNTZ</code>	Similar to <code>ACCOUNTZ</code>

Each of the subjects can be used without any input. However, for each request type (`STATZ`, `VARZ`, `SUBSZ`, `CONNS`, `ROUTEZ`, `GATEWAYZ`, `LEAFZ`, `ACCOUNTZ`, `JSZ`) a json with type specific options can be sent. Furthermore all subjects allow for filtering by providing these values as json:

Option	Effect
<code>server_name</code>	Only server with matching server name will respond.
<code>cluster</code>	Only server with matching cluster name will respond.
<code>host</code>	Only server running on that host will respond.
<code>tags</code>	Filter responders by tags. All tags must match.

Subjects available when using NATS-based resolver

Subject	Description	Input	Output
<code>\$SYS.REQ.ACCOUNT.<account-id>.CLAIMS.UPDATE</code>	Update a particular account JWT (only possible if properly signed)	JWT body	
<code>\$SYS.REQ.ACCOUNT.<account-id>.CLAIMS.LOOKUP</code>	Responds with requested JWT		JWT body
<code>\$SYS.REQ.CLAIMS.PACK</code>	Single responder compares input, sends all JWT if different.	xor of all sha256(stored-jwt). Send empty message to download all JWT.	If different, responds with all stored JWT (one message per JWT). Empty message to signify EOF
<code>\$SYS.REQ.CLAIMS.LIST</code>	Each server responds with list of account ids it stores		list of account ids separated by newline
<code>\$SYS.REQ.CLAIMS.UPDATE</code>	Exposes <code>\$SYS.REQ.ACCOUNT..CLAIMS.UPDATE</code> without the need for <code><account-id></code>	JWT body	
<code>\$SYS.REQ.CLAIMS.DELETE</code>	When the resolver is configured with <code>allow_delete: true</code> , deleting accounts is enabled.	Generic operator signed JWT claim with a field <code>accounts</code> containing a list of account ids.	

Old Subjects

Subject	Alternative Mapping
<code>\$SYS.REQ.SERVER.PING</code>	<code>\$SYS.REQ.SERVER.PING.STATSZ</code>
<code>\$SYS.ACCOUNT.<account-id>.CLAIMS.UPDATE</code>	<code>\$SYS.REQ.ACCOUNT.<account-id>.CLAIMS.LOOKUP</code>

Leaf Node Connections - Outgoing

It is important to understand that leaf nodes do not multiplex between accounts. Every account that you wish to connect across a leaf node connection needs to be explicitly listed. Thus, the system account is not automatically connected, even if both ends of a leaf node connection use the same system account. For leaf nodes connecting into a cluster or super cluster, the system account needs to be explicitly connected as separate `remote` to the same URL(s) used for the other account(s). The system account user used by providing `credentials` can be heavily restricted and for example, only allow publishing on some subjects. This also holds true when you don't use the system account yourself, but indirectly need it for NATS based account resolver or centralized monitoring.

Examples in sub sections below assume that the cluster to connect into is in operator mode.

Non Operator Mode

The outgoing connection is not in Operator mode, thus the system account may differ from the user account. This example shows how to configure a user account and the system account in a leaf node. Credentials files provided have to contain credentials that are valid server/cluster reachable by `url`. In the example, no accounts are explicitly configured, yet some are referenced. These are the default Account `$G` and the default system account `$SYS`

```
leafnodes {
  remotes = [
    {
      url: "nats://localhost:4222"
      credentials: "./your-account.creds"
    },
    {
      url: "nats://localhost:4222"
      account: "$SYS"
      credentials: "./system-account.creds"
    },
  ]
}
```

Operator Mode

Outgoing connection is in operator mode as well. This example assumes usage of the same operator and thus system account. However, using a different operator would look almost identical. Only the credentials would be issued by accounts of the other operator.

```
operator: ./trustedOperator.jwt
system_account: AAAXAUVSGK7TCRHFIRAS4SYXVJ76EWD MNXZM6ARFGXP7BASNDGLKU7A5
leafnodes {
  remotes = [
    {
      url: "nats://localhost:4222"
      account: "ADKGAJU55CHY0IF5H432K2Z2ME3NPSJ5S3VY5Q42Q30TY0CYRRG7W0WV"
      credentials: "./your-account.creds"
    },
    {
      url: "nats://localhost:4222"
      account: "AAAXAUVSGK7TCRHFIRAS4SYXVJ76EWD MNXZM6ARFGXP7BASNDGLKU7A5"
      credentials: "./system-account.creds"
    }
  ]
}
```

```
    },  
  ],  
}
```

Connecting Accounts

As shown in [what are accounts](#), they can be connected via exports and imports. While in configuration files this is straight forward, this becomes a bit more complicated when using JWTs. In part this is due to the addition of new concepts such as public/private/activation tokens that do not make sense in a config based context.

Exports

Add an export with: `nsc add export --name <export name> --subject <export subject>` This will export a public stream that can be imported by any account. To alter the export to be a service add `--service`.

To have more control over which account is allowed to import provide the option `--private`. When doing so only accounts for which you generate tokens can add the matching import. A token can be generated and stored in a file as follows: `nsc generate activation --account <account name> --subject <export subject> --output-file <token file> --target-account <account identity public NKEY>` The resulting file can then be exchanged with the importer.

Imports

To add an import for a public export use `nsc add import --account <account name> --src-account <account identity public NKEY> --remote-subject <subject of export>`. To import a service provide the option `--service`.

To add an import for a private export use `nsc add import --account <account name> --token <token file or url>` *If your nsc environment contains operator and account signing NKEYs, `nsc add import -i` will generate token to embed on the fly*

Import Subjects

Between export/import/activation tokens there are many subjects in use. Their relationship is as follows:

- Import subject is identical to or a subset of the exported subject.
- An activation token's subject is identical to or a subset of the exported subject.
- An activation token's subject is also identical to or a subset of the import subject of the account it is embedded in.

Import Remapping

In order to be independent of subject names chosen by the exporter, importing allows to remap the imported subject. To do so provide the option `--remote-subject <subject name>` to the import command.

This example will change the subject name the importing account uses locally from the exporter picked subject `foo` to `bar`.

```
nsc add import --account test --src-account
ACJ6G45BE7LL0FCVAZSZR3RY4XELXQ32B0QRI7KQMLICXXXJRP4P45Q --remote-subject foo --local-subject
bar
```

```
[ OK ] added stream import "blo"
```

Visualizing Export/Import Relationships

NSC can generate diagrams of inter account relationships using: `nsc generate diagram component --output-file test.uml` The generated file contains a [plantuml](#) component diagram of all accounts connected through their exports/imports. To turn the file into a .png execute: `plantuml -tpng test.uml` If the diagram is cut off, increase available memory and image size limit with these options: `-Xmx2048m -DPLANTUML_LIMIT_SIZE=16384`

Managing Keys

Identity keys are extremely important, so you may want to keep them safe and instead hand out more easily replaceable signing keys to operators. Key importance generally follows the chain of trust with operator keys being more important than account keys. Furthermore, identity keys are more important than signing keys.

There are instances where regenerating a completely new identity key of either type is not a feasible option. For example, you might have an extremely large deployment (IoT) where there is simply too much institutional overhead. In this case, we suggest you securely backup identity keys offline and use exchangeable signing keys instead. Depending on which key was compromised, you may have to exchange signing keys and re-sign all JWTs signed with the compromised key. The compromised key may also have to be revoked.

Whether you simply plan to regenerate new NKEY/JWT or exchange signing NKEYs and re-sign JWTs, in either case, you need to prepare and try this out beforehand and not wait until disaster strikes.

Protect Identity NKEYs

Usage of signing keys for Operator and Account has been shown in the [nsc](#) section. This shows how to take an identity key offline. Identity NKEY of the operator/account is the only one allowed to

modify the corresponding JWT and thus add/remove signing keys. Thus, initial signing keys are best created and assigned prior to removing the private identity NKEY.

Basic strategy: take them offline & delete in `nsc` NKEY directory.

Use `nsc env` to determine your NKEY directory. (Assuming `~/.nkeys` for this example) `nsc list keys --all` lists all keys under your operator and indicates if they are present and if they are signing keys.

Keys for your Operator/Account can be found under `<nkeyesdir>/keys/0/../../<public-nkey>.nk` or `<nkeyesdir>/keys/A/../../<public-nkey>.nk`. The operator identity NKEY ODMFND7EIJ2MBHNP02JHCKOZIAY6NAK7OT4V2ZT2C5O6LEB3DPKYV3QL would reside under `~/.nkeys/keys/0/DM/ODMFND7EIJ2MBHNP02JHCKOZIAY6NAK7OT4V2ZT2C5O6LEB3DPKYV3QL.nk`.

Please note that key storage is sharded by the 2nd and 3rd letter in the key

Once these files are backed up and deleted `nsc list keys --all` will show them as not stored. You can continue as normal, `nsc` will pick up signing keys instead.

Since you typically distribute user keys or creds files to your applications, there is no need for `nsc` to hold on to them in the first place. Credentials files are a concatenated user JWT and the corresponding private key, so don't forget to delete that as well.

Key and creds can be found under `<nkeyesdir>/keys/U/../../<public-nkey>.nk` and `<nkeyesdir>/creds/<operator-name>/<account-name>/<user-name>.creds`

Reissue Identity NKEYs

If you can easily re-deploy all necessary keys and JWTs, simply by re-generating a new account/user (possibly operator) this will be the simplest solution. The steps necessary are identical to the initial setup, which is why it would be preferred. In fact, for user NKEYs and JWT, generating and distributing new ones to affected applications is the best option.

Even if regeneration of an account or operator is not your first choice, it may be your method of last resort. Below sections outline the steps this would entail.

Operator

In order to reissue an operator identity NKEY use `nsc reissue operator`. It will generate a new identity NKEY and use it to sign the operator. `nsc` will also re-sign all accounts signed by the original identity NKEY. Accounts signed by operator signing keys will remain untouched.

The altered operator JWT will have to be deployed to all affected `nats-server` (one server at a time). Once all `nats-server` have been restarted with the new operator, push the altered accounts. Depending on your deployment mode you may have to distribute the operator JWT and altered account JWT to all other `nsc` environments.

This process will be a lot easier when operator signing keys were used throughout and no account will be re-signed because of this. If they were not, you can convert the old identity NKEY into a signing key using `nsc reissue operator --convert-to-signing-key`. On your own time - you can then remove the then signing NKEY using `nsc edit operator --rm-sk 0..` and redeploy the operator JWT to all `nats-server`.

Account

Unlike with the operator, account identity NKEYs can not be changed as easily. User JWT explicitly reference the account identity NKEY such that the `nats-server` can download them via a resolver. This complicates reissuing these kind of NKEYs, which is why we strongly suggest sticking to signing keys.

The basic approach is to:

1. generate a new account with similar settings - including signing NKEYs,
2. re-sign all users that used to be signed by the old identity NKEY,
3. push the account and,
4. deploy the new user JWT to all programs running inside the account.

When signing keys were used, the account identity NKEY would only be needed to self-sign the account JWT exchange with an administrators/operators `nsc` environment.

Revocations

JWTs for user, activations and accounts can be explicitly revoked. Furthermore, signing keys can be removed, thus invalidating all JWTs signed by the removed NKEY.

User

To revoke all JWTs for a user in a account issue `nsc revocations add-user --account <account name> --name <user name>`.

With the argument `--at` you can specify a time different than now. Use `nsc revocations list-users --account <account name>` to inspect the result or `nsc revocations delete-user --account <account name> --name <user name>` to remove the revocation.

```
nsc revocations add-user --account SYS --name sys
```

```
[ OK ] revoked user "UCL5YXXUKCE04HDTTYU0HDMHP4JJ6MGE3SVQBDWFZUGJUMUKE24DEUCU"
```

```
nsc revocations list-users
```

```

+-----+
|                               Revoked Users for test5                               |
+-----+
| Public Key                               | Revoke Credentials Before |
+-----+
| UAX7KQJJNL5NIRTSQSANKE3DNBHLLFUYKRXCD5QRKI75XBEHQ0A4ZZGV | Wed, 10 Feb 2021 12:51:09 EST |
+-----+

```

Please note that the revocation created only applies to JWTs issued before the time listed. Users created or updated after revocation will be valid as they are outside of the revocation time. Also, please be aware that adding a revocation will modify the account and therefore has to be pushed in order to publicize the revocation.

Activations

To revoke all activations of the export, identified by `--account` and `--subject` (`--stream` if the export is a stream), issued for a given Account identity NKEY use:

```

nsc revocations add-activation --account <account name> --subject <export name> \
  --target-account <account identity public NKEY>

```

Use `nsc revocations list-activations --account SYS` to inspect the result or:

```

nsc revocations delete_activation --account <account name> \
  --subject <export name> --target-account <account identity public NKEY>

```

to remove the revocation.

```

nsc revocations add-activation --account SYS --subject foo \
  --target-account AAUDEW26FB4T0JAQN3DYMDLCVXZMNIJWP2EMOAM5HGKLF6RGM02PV7WP

```

```

[ OK ] revoked activation "foo" for account
AAUDEW26FB4T0JAQN3DYMDLCVXZMNIJWP2EMOAM5HGKLF6RGM02PV7WP

```

```

nsc revocations list-activations --account SYS

```

```

+-----+
|                               Revoked Accounts for stream foo                               |
+-----+
| Public Key                               | Revoke Credentials Before |
+-----+

```

Please note the revocation created only applies to JWTs issued before the time listed. Activations created or edited after, will be valid as they are outside of the revocation time. Also be aware that adding a revocation will modify the account and therefore has to be pushed in order to publicize the revocation.

Accounts

Account identity NKEYS can not be revoked like user or activations. Instead lock out all users by setting the connection count to 0 using `nsc edit account --name <account name> --conns 0` and pushing the change using `nsc push --all`.

Alternatively you can also remove the account using `nsc delete account --name` and keep it from found by the account resolver. How to do this depends on your resolver type:

- [mem-resolver](#):

Remove the JWT from the configuration field `resolver_preload` and restart all `nats-server`

- [url-resolver](#):

Manually delete the JWT from the `nats-account-server` store directory.

- `nats-resolver`: Prune removed accounts using: `nsc push --all --prune`.

For this to work, the resolver has to have deletion enabled (`allow_delete: true`) and you need to be in possession of an operator signing key.

Signing keys

Accounts, Activations, and Users can be revoked in bulk by removing the respective signing key.

Remove an operator signing key: `nsc edit operator --rm-sk <signing key>` As a modification of the operator, in order to take effect, all dependent `nsc` installations as well as `nats-server` will need this new version of the operator JWT.

Remove an account signing key: `nsc edit account --name <account name> --rm-sk <signing key>`. In order to take effect, a modification of an account needs to be pushed: `nsc push --all`.

☐☐ #4

☐ cocobook-sync ☐☐ 2026-10-28 11:14:45 UTC

☐ cocobook-sync ☐☐ 2026-10-29 03:45:44 UTC