
pynsot Documentation

Release 1.1.3

Jathan McCollum

February 22, 2017

1	Installation	3
2	Quick Start	5
2.1	Create a Site	5
2.2	CLI Example	5
2.3	API Example	6
3	Documentation	7
3.1	Configuration	7
3.2	Command-Line	8
3.3	Authentication	27
3.4	Python API	28
4	API Reference	35
4.1	API	35
5	Miscellaneous Pages	43
5.1	Changelog	43
	Python Module Index	45

PyNSoT is the official client library and command-line utility for the [Network Source of Truth \(NSoT\)](#) network source-of-truth and IPAM database. For more information on the core project, please follow the link above.

Table of Contents:

- *Installation*
- *Quick Start*
 - *Create a Site*
 - *CLI Example*
 - *API Example*
- *Documentation*
- *API Reference*
- *Miscellaneous Pages*

Installation

Assuming you've got Python 2.7 and `pip`, all you have to do is:

```
$ pip install pynsot
```

We realize that this might not work for you. More detailed install instructions are Coming Soon™.

Quick Start

How do you use it? Here are some basic examples to get you started.

Important: These examples assume you've already installed and configured `pynsot`. For a detailed walkthrough, please visit [Configuration](#) and then head over to the [Command-Line](#) docs.

Create a Site

Sites are namespaces for all other objects. Before you can do anything you'll need a Site:

```
$ nsot sites add --name 'My Site'
```

These examples also assume the use of a `default_site` so that you don't have to provide the `-s/--sit-id` argument on every query. If this is your only site, just add `default_site = 1` to your `pynsotrc` file.

If you're throughly lost already, check out the [Example Configuration](#).

CLI Example

Here's an example of a few basic CLI lookups after adding several networks:

```
# Add a handful of networks
$ nsot networks add -c 192.168.0.0/16 -a owner=jathan
$ nsot networks add -c 192.168.0.0/24
$ nsot networks add -c 192.168.0.0/25
$ nsot networks add -c 192.168.0.1/32
$ nsot networks add -c 172.16.0.0/12
$ nsot networks add -c 10.0.0.0/24
$ nsot networks add -c 10.1.0.0/24

# And start looking them up!
$ nsot networks list
```

ID	Network	Prefix	Is IP?	IP Ver.	Parent ID	Attributes
1	192.168.0.0	16	False	4	None	owner=jathan
2	10.0.0.0	16	False	4	None	owner=jathan
3	172.16.0.0	12	False	4	None	

```

| 4  10.0.0.0  24  False  4  2
| 5  10.1.0.0  24  False  4  2
+-----+

$ nsot networks list --include-ips
+-----+
| ID  Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 1   192.168.0.0  16     False   4        None       owner=jathan |
| 2   10.0.0.0     16     False   4        None       owner=jathan |
| 3   172.16.0.0   12     False   4        None       |
| 4   10.0.0.0     24     False   4        2         |
| 5   10.1.0.0     24     False   4        2         |
| 6   192.168.0.1  32     True    4        1         |
+-----+

$ nsot networks list --include-ips --no-include-networks
+-----+
| ID  Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 6   192.168.0.1  32     True    4        1         |
+-----+

$ nsot networks list --cidr 192.168.0.0/16 subnets
+-----+
| ID  Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 6   192.168.0.0  24     False   4        1         |
| 7   192.168.0.0  25     False   4        6         |
+-----+

$ nsot networks list -c 192.168.0.0/24 supernets
+-----+
| ID  Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 1   192.168.0.0  16     False   4        None       owner=jathan |
|                                     cluster=   |
|                                     foo=baz   |
+-----+

```

API Example

And for the Python API? Run some Python!

If you want a more detailed walkthrough, check out the [Python API guide](#).

```

from pynsot.client import get_api_client

# get_api_client() is a magical function that returns the proper client
# according to your ``pynsotrc`` configuration
c = get_api_client()
nets = c.sites(1).networks.get()
subnets = c.sites(1).networks('192.168.0.0/16').subnets.get()
supernets = c.sites(1).networks('192.168.0.0/24').supernets.get()

```

Configuration

Configuration Basics

Configuration for pynsot consists of a single INI with two possible locations:

1. `/etc/pynsotrc`
2. `~/.pynsotrc`

The files are discovered and loaded in order, with the settings found in each location being merged together. The home directory takes precedence.

Configuration elements must be under the `pynsot` section.

If you don't create this file, running `nsot` will prompt you to create one interactively.

Like so:

```
$ nsot sites list
/home/jathan/.pynsotrc not found; would you like to create it? [Y/n]: y
Please enter URL: http://localhost:8990/api
Please enter SECRET_KEY: qONJrNpTX0_9v7H_LN1JlA0u4gdTs4rRMQklmQF9WF4=
Please enter EMAIL: jathan@localhost
```

Example Configuration

```
[pynsot]
auth_header = X-NSoT-Email
auth_method = auth_header
default_site = 1
default_domain = company.com
url = https://nsot.company.com/api
```

Configuration Reference

Key	Value	Default	Required
url	API URL. (e.g. <code>http://localhost:8990/api</code>)		Yes
email	User email	<code>\$USER@{default_domain}</code>	No
api_version	API version to use. (e.g. <code>1.0</code>)	None	No
auth_method	<code>auth_token</code> or <code>auth_header</code>		Yes
secret_key	Secret Key from your user profile		No
default_site	Default <code>site_id</code> if not provided w/ <code>-s</code>		No
auth_header	HTTP header used for proxy authentication	<code>X-NSoT-Email</code>	No
default_domain	Domain for email address	<code>localhost</code>	No

Command-Line

Welcome to the NSoT command-line interface (CLI). This is where the party starts!

Before you proceed, please make sure that you've created a `.pynsotrc` file as detailed in the [Configuration guide](#).

Data Model

If you aren't already familiarized with the data model for NSoT, it might be helpful to refer to the [NSoT Data Model guide](#).

Commands

Each object type is assigned a positional command argument:

```
$ nsot --help
Usage: nsot [OPTIONS] COMMAND [ARGS]...

    Network Source of Truth (NSoT) command-line utility.

    For detailed documentation, please visit https://nsot.readthedocs.io

Options:
  -v, --verbose  Toggle verbosity.
  --version      Show the version and exit.
  -h, --help     Show this message and exit.

Commands:
  attributes  Attribute objects.
  changes     Change events.
  devices     Device objects.
  interfaces  Interface objects.
  networks    Network objects.
  sites       Site objects.
  values      Value objects.
```

Actions

Every object type has four eligible actions representing creation, retrieval, update, and deletion (CRUD):

- add
- list
- remove
- update

Note: There are two exceptions: *Changes* and *Values*, which are immutable and can only be viewed.

For example, for nsot devices:

```
$ nsot devices -h
Usage: nsot devices [OPTIONS] COMMAND [ARGS]...

Device objects.

A device represents various hardware components on your network such as
routers, switches, console servers, PDUs, servers, etc.

Devices also support arbitrary attributes similar to Networks.

Options:
  -h, --help  Show this message and exit.

Commands:
  add      Add a new Device.
  list     List existing Devices for a Site.
  remove   Remove a Device.
  update   Update a Device.
```

Getting Help

Every object type and action for each has help text that can be accessed using the `-h/--help` option. It's quite good. Use it!

Verbosity

The CLI utility tries to be as concise as possible when telling you what it's doing. Sometimes it may be useful to increase verbosity using the `-v/--verbose` flag.

For example, if you encounter an error and want to know more:

```
$ nsot devices add --hostname ''
[FAILURE] hostname: This field may not be blank.

$ nsot --verbose devices add --hostname ''
[FAILURE] hostname: This field may not be blank.
400 BAD REQUEST trying to add device with args: bulk_add=None, attributes={}, hostname=
```

Required Options

When adding objects, certain fields will be required. The required options will be designated as such with a `[required]` tag in the help text (for example from `nsot sites add --help`):

```
-n, --name NAME          The name of the Site. [required]
```

If a required option is not provided, nsot will complain:

```
Error: Missing option "-n" / "--name".
```

Site ID

For all object types other than Sites, the `-s/--site-id` option is required to specify which Site you would like the object to be under. See [Configuration Reference](#) for setting a default site.

Resource Types

NSoT refers internally to any object that can have attributes as *Resource Types* or just *Resources* for short. As of this writing this includes Device, Network, and Interfaces objects.

You will also see command-line arguments referencing *Resource Name* to indicate the name of a Resource Type.

There are a number of features, settings, command-line flags and command-line arguments that are common to all Resource Types as they relate to managing or displaying attribute values.

This will be important to note later on in this documentation.

Natural Keys

A “natural key” is a field or set of fields which can uniquely identify an object. Natural keys are intended to be used as a human-readable identifier to improve user experience and simplify interaction with NSoT.

For the purpose of display all objects have a natural key for one or more fields as follows:

- Sites: {name}
- Attributes: {resource_name:name}
- Devices: {hostname}
- Networks: {cidr}
- Interfaces: {device_id:name}

Updating or Removing Objects

When updating or removing objects, you may specify their unique ID or (if applicable) their natural key.

For objects that do not support update by natural key, unique IDs can be obtained using the `list` action.

Currently the only *Resource Types* to currently support update or removal by natural key are:

- Devices (hostname)
- Networks (CIDR)

For example, this illustrates updating a Network object by natural key (cidr) or by ID:

```

$ nsot networks list
+-----+
| ID      Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 1       10.0.0.1      32      False   4        None       desc=test  |
+-----+

$ nsot networks update --cidr 10.0.0.1/32 -a desc="Changing this"
[SUCCESS] updated network

$ nsot networks update -i 1 -a desc="Changing this"
[SUCCESS] updated network

```

Updating Attributes

When modifying attributes on *Resource Types*, you have three actions to choose from:

- Add (`--add-attributes`). This is the default behavior that will add attributes if they don't exist, or update them if they do.
- Delete (`--delete-attributes`). This will cause attributes to be deleted. If combined with `--multi` the attribute will be deleted if either no value is provided, or if the attribute no longer contains a valid value.
- Replace (`--replace-attributes`). This will cause attributes to be replaced. If combined with `--multi` and multiple attributes of the same name are provided, only the last value provided will be used.

Please note that this does not apply when updating Attribute resources themselves. Attribute values attached to *Resource Types* are considered to be “instances” of Attributes.

Viewing Objects

The `list` action for each object type supports `-i/--id`, `-l/--limit` and `-o/--offset` options.

- The `-i/--id` option will retrieve a single object by the provided unique ID and will override any other list options.
- The `-l/--limit` option will limit the set of results to N resources.
- The `-o/--offset` option will skip the first N resources.

Set Queries

All *Resource Types* support a `-q/--query` option that is a representation of set operations for matching attribute/value pairs.

The operations are evaluated from left-to-right, where the first character indicates the set operation:

- `+` indicates a set *union*
- `-` indicates a set *difference*
- no marker indicates a set *intersection*

For example:

- `-q "vendor=juniper"` would return the set intersection of objects with `vendor=juniper`.
- `-q "vendor=juniper -metro=iad"` would return the set difference of all objects with `vendor=juniper` (that is all `vendor=juniper` where `metro` is not `iad`).

- `-q "vendor=juniper +vendor=cisco` would return the set union of all objects with `vendor=juniper` or `vendor=cisco` (that is all objects matching either).

The ordering of these operations is important. If you are not familiar with set operations, please check out [Basic set theory concepts and notation](#) (Wikipedia).

Note: The default display format for set queries is the same as `-N/--natural-key` (see below) for non-set-query lookups.

Important: When performing a set query for more than one operation, you must enclose it in quotations so that the space characters are properly passed to the argument parser.

For example:

```
$ nsot devices list --query vendor=juniper
iad-r1
lax-r2

$ nsot devices list --query vendor=juniper -metro=iad # Needs quotes!
Error: no such option: -m

$ nsot devices list --query 'vendor=juniper -metro=iad' # There we go!
lax-r2

$ nsot devices list --query 'vendor=juniper +vendor=cisco'
chi-r1
chi-r2
iad-r1
iad-r2
lax-r2
```

Because set queries return newline-delimited results, they can be nice for quickly feeding lists of objects to other utilities. For example, `snmpwalk`:

```
# For all top of rack switches, poll SNMP IF-MIB::ifDescr and store in files
nsot devices list -q role=tor | xargs -I '{}' sh -c 'snmpwalk -v2c -c public "$1" .1.3.6.1.2.1.2.2.1
```

Output Modifiers

The following modifying flags are available when viewing objects.

All Objects

The following flags apply to all objects.

- `-N/--natural-key` - Display list results by their uniquely identifying *natural key*.

```
$ nsot sites list --natural-key
Demo Site
```

Resource Types

The following output modifiers apply to *Resource Types* only.

- `-g/--grep` - Display list results in a grep-friendly format. This modifies the output in a way where the natural key is displayed first, and then each attribute/value pair (if any) is displayed one per line.

Note: Objects without any attributes will not be displayed.

```
$ nsot devices list --attributes vendor=juniper --grep
lax-r2 hw_type=router
lax-r2 metro=lax
lax-r2 owner=jathan
lax-r2 vendor=juniper
iad-r1 hw_type=router
iad-r1 metro=iad
iad-r1 owner=jathan
iad-r1 vendor=juniper

$ nsot devices list --attributes vendor=juniper --grep | grep metro
lax-r2 metro=lax
iad-r1 metro=iad
```

- `-d/--delimited` - When performing a set query using `-q/--query`, this will display set query results separated by commas instead of newlines.

```
$ nsot devices list --query vendor=juniper ---delimited
iad-r1,lax-r2
```

Bulk Addition of Objects

Attributes, Devices, and Networks may be created in bulk by using the `-b/--bulk-add` option and specifying a file path to a colon-delimited file.

The format of this file must adhere to the following format:

- The first line of the file must be the field names.
- All required fields must be present, however, the order of any of the fields does not matter.
- Repeat: The fields may be in any order so long as the required fields are present! Missing fields will fallback to their defaults!
- Attribute pairs must be comma-separated, and in format `k=v` and the attributes must exist!
- For any fields that require Boolean values, the following applies:
 - You may specify `True` or `False` and they will be evaluated
 - If the value for a field is not set it will evaluate to `False`
 - Any other value for a field will evaluate to `True`

Attributes

Sample file for `nsot devices add --bulk-add /tmp/attributes:`

```
name:resource_name:required:description:multi:display
owner:Network:True:Network owner:True:True
metro:Device:False:Device metro:False:True
```

Devices

Sample file for nsot devices add --bulk-add /tmp/devices:

```
hostname:attributes
device5:foo=bar,owner=team-networking
device6:foo=bar,owner=team-networking
```

Networks

Sample file for nsot networks add --bulk-add /tmp/networks:

```
cidr:attributes
10.20.30.0/24:foo=bar,owner=team-networking
10.20.31.0/24:foo=bar,owner=team-networking
```

Interfaces

Bulk addition of Interfaces via CLI is not supported at this time.

Working with Objects

This section walks through the basics of how to interact with each object and action from the command-line.

Sites

Sites are the top-level object from which all other objects descend. In other words, Sites contain Attributes, Devices, Networks, Interfaces, etc. These examples illustrate having many Sites, but in practice you'll probably only have one or two sites.

Adding a Site:

```
$ nsot sites add --name Spam --description 'Spam Site'
[SUCCESS] added site with args: name=Space, description=Spam Site!
```

Listing all Sites:

```
$ nsot sites list
+-----+
| ID   Name   Description |
+-----+
| 1    Foo    Foo Site   |
| 2    Bar    Bar Site   |
| 3    Baz    Baz Site   |
| 4    Spam   Sheep Site |
| 5    Sheep   Sheep Site |
+-----+
```

Listing a single Site:

```
$ nsot sites list --name Foo
+-----+
| ID   Name   Description |
+-----+
```

```
| 1   Foo   Foo Site |
+-----+
```

Listing a few Sites:

```
$ nsot sites list --limit 2
+-----+
| ID   Name   Description |
+-----+
| 1    Foo    Foo Site   |
| 2    Bar    Bar Site   |
+-----+
```

Updating a Site:

```
$ nsot sites update --id 2 --name Snickers
[SUCCESS] updated site with args: description=None, name=Snickers!

$ nsot sites list --name Snickers
+-----+
| ID   Name       Description |
+-----+
| 2    Snickers   Bar Site   |
+-----+
```

Removing a Site:

```
$ nsot sites remove --id 1
[SUCCESS] removed site with args: id=1!
```

Attributes

Attributes are flexible key/value pairs or tags you may use to assign arbitrary data to objects.

Note: Before you may assign Attributes to other resources, you must create the Attribute first!

Adding an Attribute:

```
$ nsot attributes add --site-id 1 -n owner --r Device -d "Owner of a device." --required
[SUCCESS] Added attribute!
```

Listing all Attributes:

```
$ nsot attributes list --site-id 1
+-----+
| ID   Name   Resource   Required?   Display?   Multi?   Description           |
+-----+
| 3    owner   Device     True        False      False    Owner of a device.   |
| 4    foo     Network    False       False      False    Foo for devices       |
| 2    owner   Network    False       False      False    Owner of a network.   |
+-----+
```

You may also list Attributes by name:

```
$ nsot attributes list --site-id 1 --name owner
+-----+
| ID   Name   Resource   Required?   Display?   Multi?   Description           |
```

```

+-----+
| 3   owner   Device   False   True    False   Owner of a device. |
| 2   owner   Network  False   False   False   Owner of a network. |
+-----+

```

When listing a single Attribute by ID, you get more detail:

```

$ nsot attributes list --site-id 1 --id 3
+-----+
| Name      Resource  Required?  Display?  Multi?   Constraints          Description
+-----+
| owner     Device    False     False     False    pattern=             Device owner.
|                                     valid_values=
|                                     allow_empty=False
+-----+

```

Updating an Attribute:

```

$ nsot attributes update --site-id 1 --id 3 --no-required
[SUCCESS] Updated attribute!

$ nsot attributes list --site-id 1 --id 3
+-----+
| ID   Name      Resource  Required?  Display?  Multi?  Description          |
+-----+
| 3    owner     Device    False     False     False   Owner of a device.  |
+-----+

```

Attributes may also be uniquely identified by name and resource_name in lieu of using id:

```

$ nsot attributes update --site-id 1 --name owner --resource-name device --multi
[SUCCESS] Updated attribute!

$ nsot attributes list --site-id 1 --name owner --resource-name device
+-----+
| ID   Name      Resource  Required?  Display?  Multi?  Description          |
+-----+
| 3    owner     Device    False     False     True    Owner of a device.  |
+-----+

```

Removing an Attribute:

```

$ nsot attributes remove --site-id 1 --id 6
[SUCCESS] Removed attribute with args: id=6!

```

Networks

A Network resource can represent an IP Network or an IP Address. Working with networks is usually done with CIDR notation. Networks can have any number of arbitrary Attributes.

Adding a Network:

```

$ nsot networks add --site-id 1 --cidr 192.168.0.0/16 --attributes owner=jathan
[SUCCESS] Added network!

```

Listing Networks:

```

$ nsot networks list --site-id 1
+-----+

```

ID	Network	Prefix	Is IP?	IP Ver.	Parent ID	Attributes
1	192.168.0.0	16	False	4	None	owner=jathan
2	10.0.0.0	16	False	4	None	owner=jathan
3	172.16.0.0	12	False	4	None	
4	10.0.0.0	24	False	4	2	
5	10.1.0.0	24	False	4	2	

You may also optionally include IP addresses with `--include-ips`:

```
$ nsot networks list --site-id 1 --include-ips
```

ID	Network	Prefix	Is IP?	IP Ver.	Parent ID	Attributes
1	192.168.0.0	16	False	4	None	owner=jathan
2	10.0.0.0	16	False	4	None	owner=jathan
3	172.16.0.0	12	False	4	None	
4	10.0.0.0	24	False	4	2	
5	10.1.0.0	24	False	4	2	
6	192.168.0.1	32	True	4	1	

Or, you may show only IP addresses by using `--include-ips` with `--no-include-networks`:

```
$ nsot networks list --site-id 1 --include-ips --no-include-networks
```

ID	Network	Prefix	Is IP?	IP Ver.	Parent ID	Attributes
6	192.168.0.1	32	True	4	1	

Performing a set query on Networks by attribute/value:

```
$ nsot networks list --site-id 1 --query owner=jathan
10.0.0.0/16
192.168.0.0/16
```

You may also display the results comma-delimited:

```
$ nsot networks list --site-id 1 --query owner=jathan --delimited
10.0.0.0/16,192.168.0.0/16
```

Updating a Network (`-a/--attributes` can be provide once for each Attribute):

```
$ nsot networks update --site-id 1 --id 1 -a owner=jathan -a foo=bar
[SUCCESS] Updated network!

$ nsot networks list --site-id 1 --id 1
```

ID	Network	Prefix	Is IP?	IP Ver.	Parent ID	Attributes
1	192.168.0.0	16	False	4	None	owner=nobody foo=bar

To delete attributes, reference each attribute by name and include the `--delete-attributes` flag:

```
$ nsot networks update --site-id 1 --id 1 -a owner --delete-attributes
```

```
$ nsot networks list --site-id 1 --id 1
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes  |
+-----+
| 1    192.168.0.0   16     False   4        None       owner=nobody |
+-----+
```

Removing a Network:

```
$ nsot networks remove --site-id 1 --id 2
[SUCCESS] Removed network!
```

You may also remove a Network by its CIDR:

```
$ nsot networks remove --site-id 1 --cidr 10.20.30.0/24
[SUCCESS] Removed network!
```

Ancestors

Recursively get all parents of a network:

```
$ nsot networks list -c 10.20.30.1/32 ancestors
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes  |
+-----+
| 1    10.0.0.0        8       False   4        None       allocated  |
| 20   10.20.0.0       16      False   4        1         allocated  |
| 15   10.20.30.0       24      False   4        1         allocated  |
+-----+
```

Assignments

Get interface assignments for a network:

```
$ nsot networks list -c 10.20.30.1/32 assignments
+-----+
| ID   Hostname  Interface |
+-----+
| 2    foo-bar1   eth0      |
+-----+
```

Children

Get immediate children of a network:

```
$ nsot networks list -c 10.20.30.0/24 children
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes  |
+-----+
| 16   10.20.30.1     32     True    4        15        assigned  |
| 17   10.20.30.3     32     True    4        15        allocated  |
| 18   10.20.30.16    28     False   4        15        allocated  |
| 19   10.20.30.104   32     True    4        15        allocated  |
+-----+
```

Closest Parent

Get the closest matching parent of a network, even if the network isn't found in the database:

```
$ nsot networks list -c 10.101.103.100/30
No network found matching args: include_ips=True, root_only=False, network_address=None, state=None,

$ nsot networks list -c 10.101.103.100/30 closest_parent
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 1    10.0.0.0       8       False   4        None      allocated  |
+-----+
```

Descendants

Recursively get all children of a network:

```
$ nsot networks list -c 10.20.0.0/16 descendants
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 15   10.20.30.0      24      False   4        20        allocated  |
| 16   10.20.30.1      32      True    4        15        assigned   |
| 17   10.20.30.3      32      True    4        15        allocated  |
| 18   10.20.30.16     28      False   4        15        allocated  |
| 19   10.20.30.104    32      True    4        15        allocated  |
+-----+
```

Next Address

Get next available addresses for a network:

```
$ nsot networks list -c 10.20.30.0/24 next_address -n 3
10.20.30.2/32
10.20.30.4/32
10.20.30.5/32
```

Next Network

Get next available networks for a network:

```
$ nsot networks list -c 10.20.30.0/24 next_network -p 28 -n 3
10.20.30.0/28
10.20.30.32/28
10.20.30.48/28
```

Parent

Get parent network of a network:

```
$ nsot networks list -c 10.20.30.0/24 parent
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 20   10.20.0.0     16     False   4        1          allocated  |
+-----+
```

Reserved

Get all reserved networks:

```
$ nsot networks list reserved
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 10   10.10.12.0     24     False   4        5          reserved  type=loopback |
|      |              |      |      |      |      |      |      |      |      |
| 12   10.10.10.15   32     True    4        6          reserved  type=internal |
+-----+
```

Root

Get parent of all ancestors of a network:

```
$ nsot networks list -c 10.20.30.3/32 root
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 1    10.0.0.0       8       False   4        None       allocated  |
+-----+
```

Siblings

Get networks with same parent as a network:

```
$ nsot networks list -c 10.20.30.3/32 siblings
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 16   10.20.30.1     32     True    4        15         assigned  |
| 18   10.20.30.16   28     False   4        15         allocated  |
| 19   10.20.30.104  32     True    4        15         allocated  |
+-----+
```

You may also include the network itself:

```
$ nsot networks list -c 10.20.30.3/32 siblings --include-self
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State      Attributes |
+-----+
| 16   10.20.30.1     32     True    4        15         assigned  |
| 17   10.20.30.3     32     True    4        15         allocated  |
| 18   10.20.30.16   28     False   4        15         allocated  |
| 19   10.20.30.104  32     True    4        15         allocated  |
+-----+
```


Subnets

Given Network 192.168.0.0/16, you may view the Networks it contains (aka subnets):

```
$ nsot networks list --site-id 1 --id 1 subnets
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 6    192.168.0.0  24      False   4         1          |
| 7    192.168.0.0  25      False   4         6          |
+-----+
```

Supernets

Given a Network 192.168.0.0/24, you may view the Networks containing it (aka supernets):

```
$ nsot networks list --site-id 1 --id 6
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 6    192.168.0.0  24      False   4         1          |
+-----+
```

You may view the networks that contain that Network (aka supernets):

```
$ nsot networks list --site-id 1 --id 5 supernets
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  Attributes |
+-----+
| 1    192.168.0.0  16      False   4         None       owner=jathan |
|                                           cluster=    |
|                                           foo=baz    |
+-----+
```

Devices

A Device represents various hardware components on your network such as routers, switches, console servers, PDUs, servers, etc.

Devices also support arbitrary attributes similar to Networks.

Adding a Device:

```
$ nsot devices add --site-id 1 --hostname foo-bar1 --attributes owner=neteng
[SUCCESS] Added device!
```

Listing Devices:

```
$ nsot devices list --site-id 1
+-----+
| ID   Hostname    Attributes |
+-----+
| 1    foo-bar1     owner=jathan |
| 2    foo-bar2     owner=neteng |
| 3    bar-baz1     owner=jathan |
| 4    bar-baz2     owner=neteng |
+-----+
```

Performing a set query on Device by attribute/value:

```
$ nsot devices list --site-id 1 --query owner=neteng
bar-baz2
foo-bar2
```

You may also display the results comma-delimited:

```
$ nsot devices list --site-id 1 --query owner=neteng --delimited
bar-baz2,foo-bar2
```

Updating a Device:

```
$ nsot devices update --id 1 --hostname potato
[SUCCESS] Updated device with args: attributes={}, hostname=potato!

$ ./nsot devices list --site-id 1 --id 1
+-----+
| ID   Hostname  Attributes |
+-----+
| 1    potato   |
+-----+
```

To delete attributes, reference each attribute by name and include the `--delete-attributes` flag:

```
$ nsot devices update --site-id 1 --id 2 -a owner --delete-attributes

$ nsot devices list --site-id 1 --id 2
+-----+
| ID   Hostname  Attributes |
+-----+
| 2    foo-bar2  |
+-----+
```

Removing a Device:

```
$ nsot devices remove --site-id 1 --id 1
[SUCCESS] Removed device!
```

You may also remove a Device by its hostname:

```
$ nsot devices remove --site-id 1 --hostname delete-me
[SUCCESS] Removed device!
```

Interfaces

Note: If you don't have any interfaces yet, that's ok. Skip to the next section and refer back here when you do.

Device objects also allow you to display their interfaces using the `interfaces` sub-command:

```
$ nsot devices list --hostname foo-bar1 interfaces
+-----+
| ID   Device  Name   MAC   Addresses  Attributes |
+-----+
| 1    1       eth0   None  |
| 1    1       eth0   None  |
+-----+
```

Interfaces

An Interface represents a network interface or port on a Device. Interfaces may only be created by “attaching” them to a Device object, just like in real life.

Interfaces, like all other *Resource Types*, support arbitrary attributes.

For these examples, we’re going to assume we’ve got a Device object with hostname `foo-bar1` with id of 1.

Adding an Interface:

```
$ nsot interfaces add --device 1 --name eth0
[SUCCESS] Added interface!
```

Let’s add another Interface:

```
$ nsot interfaces add --device 1 --name eth1
[SUCCESS] Added interface!
```

Listing all Interfaces:

```
$ nsot interfaces list
+-----+
| ID  Device  Name  MAC  Addresses  Attributes |
+-----+
| 1    1      eth0  None  |
| 1    1      eth0  None  |
+-----+
```

Listing a single Interface shows more detail:

```
$ nsot interfaces list --name eth0
+-----+
| ID  Device  Name  MAC  Addresses  Speed  Type  Parent  Attributes |
+-----+
| 1    1      eth0  None  |
| 1000  6      None  |
+-----+
```

But what if you’ve got more than one interface named `eth0`? You can filter interfaces by `-D/--device`, which when listing can either be ID or hostname of the device:

```
$ nsot interfaces list --device foo-bar1 -n eth0
+-----+
| ID  Device  Name  MAC  Addresses  Speed  Type  Parent  Attributes |
+-----+
| 1    1      eth0  None  |
| 1000  6      None  |
+-----+
```

Interfaces also support attributes:

```
$ nsot attributes add --resource-name interface --name vlan
[SUCCESS] Added attribute!

$ nsot interfaces update --id 1 -a vlan=100
[SUCCESS] Updated interface!

$ nsot interfaces update --id 2 -a vlan=100
[SUCCESS] Updated interface!

$ nsot interfaces list --id 1
+-----+
```

ID	Device	Name	MAC	Addresses	Speed	Type	Parent	Attributes
1	1	eth0	None		1000	6	None	vlan=100

Performing a set query on Interfaces by attribute/value (it displays by device_id:name for now. To be improved later.):

```
$ nsot interfaces list --query vlan=100
1:eth0
1:eth1
```

You may also display the results comma-delimited:

```
$ nsot interfaces list --query vlan=100 --delimited
1:eth0,1:eth1
```

You may also specify the type (ethernet, etc... more on this later), speed (in Mbps), and mac_address:

```
$ nsot interfaces update --id 2 --speed 10000 --type 161 --mac-address
[SUCCESS] Updated interface!

$ nsot interfaces list --id 2
+-----+-----+-----+-----+-----+-----+-----+-----+
| ID  | Device | Name  | MAC          | Addresses | Speed  | Type  | Parent | Attributes |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 2   | 1      | eth1  | 6C:40:08:A5:96:86 |           | 10000 | 161   | None   | vlan=100   |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

You may also assign IP addresses to Interfaces. These are represented by an assignment relationship to a Network object that contains a host address (/32 for IPv4 or /128 for IPv6). When assigning an address to an Interface, if a record does not already exist, one is created with state=assigned. If one does exist, its state is updated:

```
$ nsot interfaces update --id 1 --addresses 10.10.10.1/32
[SUCCESS] Updated interface!

$ nsot interfaces list --id 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| ID  | Device | Name  | MAC          | Addresses | Speed  | Type  | Parent | Attributes |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | 1      | eth0  | None         | 10.10.10.1/32 | 1000  | 6     | None   | vlan=100   |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Just like in real life, it is an error to assign an IP address to already assigned to another interface on the same Device:

```
$ nsot interfaces update --id 2 --addresses 10.10.10.1/32
[FAILURE] address: Address already assigned to this Device.
```

Removing an Interface:

```
$ nsot interfaces remove --id 2
[SUCCESS] Removed interface!
```

Addressseses

Given an Interface, you may display the associated Network addresses:

```
$ nsot interfaces list --id 1 addresses
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State   Attributes |
+-----+
| 7    10.10.10.1    32     True    4        6         assigned  type=loopback |
|                                           dns_hostname=lax-r1 |
+-----+
```

Networks

Given an Interface, you may display the containing networks for any addresses assigned to the interface:

```
$ nsot interfaces list --id 1 networks
+-----+
| ID   Network      Prefix  Is IP?  IP Ver.  Parent ID  State   Attributes |
+-----+
| 6    10.10.10.0    24     False   4        5         allocated  type=loopback |
|                                           metro=lax |
+-----+
```

Assignments

Given an Interface, you may display the underlying assignment objects that represent the relationship between Interface <=> Network:

```
$ nsot interfaces list --id 1 assignments
+-----+
| ID   Device      Device ID  Address          Interface  Interface ID |
+-----+
| 1    foo-bar1     1         10.10.10.1/32   eth0      1 |
+-----+
```

Values

Values represent attribute values and cannot be directly manipulated. They can be viewed, however, and this command allows you to do that.

All unique values for a matching Attribute will be displayed.

Displaying values by Attribute name:

```
$ nsot values list --name metro
chi
iad
lax
```

You might have an Attribute with the same name (e.g. metro) across multiple *Resource Types*. If you do, you'll want to also filter by resource name:

```
$ nsot values list --name metro --resource-name network
lax
```

Changes

All Create/Update/Delete events are logged as a Change. A Change includes information such as the change time, user, and the full resource after modification. Changes are immutable and can only be removed by deleting the entire Site.

Listing Changes:

```
$ nsot changes list --site-id 1 --limit 5
+-----+
| ID   Change At           User           Event   Resource  Obj |
+-----+
| 73   2015-03-04 11:12:30  jathan@localhost Delete   Device    1  |
| 72   2015-03-04 11:10:46  jathan@localhost Update   Device    1  |
| 71   2015-03-04 11:06:03  jathan@localhost Create   Device    7  |
| 70   2015-03-04 10:56:54  jathan@localhost Update   Network   6  |
| 69   2015-03-04 10:53:30  jathan@localhost Create   Network   6  |
+-----+
```

When listing a single Change event by ID, you get more detail:

```
$ nsot changes list --site-id 1 --id 73
+-----+
| Change At           User           Event   Resource  ID  Data |
+-----+
| 2015-03-04 11:12:30  jathan@localhost Delete   Device    1  attributes: |
|                                                                hostname:potato |
|                                                                site_id:1      |
|                                                                id:1          |
+-----+
```

Debugging

Is `-v/--verbose` just not cutting it? Are you really confused on what's wrong? Do you want **ALL THE DETAIL**? Then this is for you.

You may toggle debug output by setting the `PYNSOT_DEBUG` environment variable to any true value.

Using the example above:

```
$ export PYNSOT_DEBUG=1

$ nsot devices add --hostname ''
DEBUG:pynsot.commands.callbacks:TRANSFORM_ATTRIBUTES [IN]: ()
DEBUG:pynsot.commands.callbacks:TRANSFORM_ATTRIBUTES [OUT]: {}
DEBUG:pynsot.client:Reading dotfile.
DEBUG:pynsot.client:Validating auth_method: auth_header
DEBUG:pynsot.client:Skipping 'debug' in config for auth_method 'auth_header'
DEBUG:pynsot.client:Using api_version = 1.0
DEBUG:pynsot.commands.callbacks:GOT DEFAULT_SITE: 1
DEBUG:pynsot.commands.callbacks:GOT PROVIDED SITE_ID: None
DEBUG:pynsot.app:adding {'bulk_add': None, 'attributes': {}, 'hostname': u'', 'site_id': '1'}
DEBUG:pynsot.app:rebase: Got site_id: 1
DEBUG:pynsot.app:rebase: Site_id found; rebasing API URL!
INFO:requests.packages.urllib3.connectionpool:Starting new HTTP connection (1): localhost
DEBUG:requests.packages.urllib3.connectionpool:"POST /api/sites/1/devices/ HTTP/1.1" 400 None
DEBUG:pynsot.app:API ERROR: {'error': {'message': {'hostname': [u'This field may not be blank.']}},
DEBUG:pynsot.app:FORMATTING MESSAGE: {'hostname': [u'This field may not be blank.']}
DEBUG:pynsot.app:ERROR MESSAGE = {'hostname': [u'This field may not be blank.']}
```

```
DEBUG:pynsot.app:PRETTY DICT INCOMING DATA = {'hostname': [u'This field may not be blank.']}
DEBUG:pynsot.app:PRETTY DICT INCOMING DATA = {'bulk_add': None, u'attributes': {}, u'hostname': u'}
[FAILURE] hostname: This field may not be blank.
```

Tip: Combine debug output with `-v/--verbosity` for MAXIMUM OUTPUT.

Authentication

NSoT supports two methods of authentication and these are implemented in the client:

1. `auth_token`
2. `auth_header`

The client default is `auth_token`, but `auth_header` is more flexible for “zero touch” use.

If sticking with the defaults, you’ll need to retrieve your key from `/profile` in the web interface.

Refer to *Configuration Reference* for setting these in your `pynsotrc`.

Python Client

Assuming your configuration is correct, the CLI interface doesn’t need anything special to make authentication work. The following only applies to retrieving a client instance in Python.

```
from pynsot.client import AuthTokenClient, EmailHeaderClient, get_api_client

# This is the preferred method, returning the appropriate client according
# to your dotfile if no arguments are supplied.
#
# Alternatively you can override options by passing url, auth_method, and
# other kwargs. See `help(get_api_client)` for more details
c = get_api_client()

# OR using the client objects directly

email = 'jathan@localhost'
secret_key = 'qONJrNpTX0_9v7H_LN1JlA0u4gdTs4rRMQklmQF9WF4='
url = 'http://localhost:8990/api'
c = AuthTokenClient(url, email=email, secret_key=secret_key)

# Email Header Client
domain = 'localhost'
auth_header = 'X-NSoT-Email'
c = EmailHeaderClient(
    url,
    email=email,
    default_domain=domain,
    auth_header=auth_header,
)
```

Python API

The Python API offers flexibility when you need it and works well for extending applications to be NSoT-aware.

In the examples below, we're going to assume the following common pre-work done in your code:

```
from pynsot.client import get_api_client

c = get_api_client()
```

Note: The tables under each section reflect the `networks` resource

Unless otherwise stated, everything applies to other resources as well.

What is Slumber?

You might notice when following along that the API clients are Slumber instances. Slumber is a wrapper around the *requests* library to make REST much more pleasant to use in Python. Each attribute of the client object represents part of the HTTP path.

For the purpose of pynsot you should be in good hands for the rest of this document, but for more information on Slumber, see the [official Slumber documentation](#).

It might help to also refer to the [REST docs](#) for NSoT.

Fetching Resources

HTTP Method	GET
HTTP Path	GET /api/sites/1/networks/ GET /api/sites/1/networks/1/ GET /api/sites/1/networks/10.0.0.0/24/ GET /api/sites/1/networks/?limit=1
Python Client	<code>c.sites(1).networks.get()</code> <code>c.sites(1).networks(1).get()</code> <code>c.sites(1).networks('10.0.0.0/24').get()</code> <code>c.sites(1).networks.get(limit=1)</code>

The details and comparisons are shown above to demystify what the client does. This is the most basic form of fetching one or many resources.

Any of these calls should return either a single dictionary or a list of dictionaries.

In the fourth example, note the HTTP query parameter. `limit` is used to restrict the number of results returned (pagination) and `offset` can be provided to begin after `n`. In the returned payload are the suggested `next` and `previous` page URLs.

Query parameters are used for filtering on resource properties except for by attribute values. (Those are covered in another section)

Note: “all” in the following context means all, regardless of site. Sites are normally used to separate conflicting sets of data so unless you intend to span sites, try to use `c.sites(ID)` for your normal clienting

```

all_nets = c.networks.get()
all_nets_for_site1 = c.sites(1).networks.get()
all_hosts = c.networks.get(prefix_length=32)

all_resources = {}

for resource_name in ['networks', 'devices', 'interfaces']:
    client = getattr(c, resource_name)
    all_resources[resource_name] = client.get()

```

Creating Resources

HTTP Method	POST
HTTP Path	POST /api/sites/1/networks/
Python Client	c.sites(1).networks.post({...})

To create a resource, you POST the payload to the server. Each resource has different required fields and defaults for those that aren't. The easiest way to reference this information is via the CLI help, browsing /api/, or try-except to catch the HTTP 400 and inspect e.response.json().

NSoT also has BULK operations. The only difference is that the payload is an array of resources.

```

net = {'attributes': {}, 'network_address': '10.0.1.0', 'prefix_length': 24}
c.sites(1).networks.post(net)
# {'attributes': {},
#  u'id': 6,
#  u'ip_version': u'4',
#  u'is_ip': False,
#  u'network_address': u'10.0.1.0',
#  u'parent_id': 1,
#  u'prefix_length': 24,
#  u'site_id': 1,
#  u'state': u'allocated'}

try:
    net = {'network_address': '8.8.8.0', 'prefix_length': 24}
    c.sites(1).networks.post(net)
except Exception as e:
    print(e.response.json())
    # {'error': {'code': 400,
    #  u'message': {u'attributes': [u'This field is required.']}},
    #  u'status': u'error'}

```

Updating Resources (Replace)

HTTP Method	PUT
HTTP Path	PUT /api/sites/1/networks/1/ PUT /api/sites/1/networks/10.0.0.0/24/
Python Client	c.sites(1).networks(1).put({...}) c.sites(1).networks('10.0.0.0/24').put({...})

In NSoT, a PUT/Replace action means to update properties of a resource while resetting to default the unspecified properties. This is typically to replace attributes but applies to any non set-in-stone property such as parent_id, id, the resource identity keys (hostname, network, etc), and others.

A successful call will return the new payload representing the upstream resource.

Like Creating, PUT also supports BULK operations.

```
# Fetch example resource
net = c.sites(1).networks('10.0.1.0/24').get()
# {u'attributes': {u'desc': u'test'},
#  u'id': 3,
#  u'ip_version': u'4',
#  u'is_ip': False,
#  u'network_address': u'10.0.1.0',
#  u'parent_id': 1,
#  u'prefix_length': 24,
#  u'site_id': 1,
#  u'state': u'allocated'}

net['attributes'] = {}
c.sites(1).networks('10.0.1.0/24').put(net)
# {u'attributes': {},
#  u'id': 3,
#  u'ip_version': u'4',
#  u'is_ip': False,
#  u'network_address': u'10.0.1.0',
#  u'parent_id': 1,
#  u'prefix_length': 24,
#  u'site_id': 1,
#  u'state': u'allocated'}
```

Updating Resources (Partial)

HTTP Method	PATCH
HTTP Path	PATCH /api/sites/1/networks/1/ PATCH /api/sites/1/networks/10.0.0.0/24/
Python Client	c.sites(1).networks(1).patch({...}) c.sites(1).networks('10.0.0.0/24').patch({...})

As opposed to PUT which can replace existing data, PATCH is “safer” in that regard. If you don’t provide some keys in your update, they will be untouched.

As with PUT and POST, a successful one should return the new payload.

Like Creating, PATCH also supports BULK operations.

```
net = c.sites(1).networks('10.0.1.0/24').get()
# {u'attributes': {u'dc': u'sfo'},
#  u'id': 3,
#  u'ip_version': u'4',
#  u'is_ip': False,
#  u'network_address': u'10.0.1.0',
#  u'parent_id': 1,
#  u'prefix_length': 24,
#  u'site_id': 1,
#  u'state': u'allocated'}

net.pop('attributes')
c.sites(1).networks('10.0.1.0/24').patch(net)
# {u'attributes': {u'dc': u'sfo'},
#  u'id': 3,
#  u'ip_version': u'4',
#  u'is_ip': False,
```

```
# u'network_address': u'10.0.1.0',
# u'parent_id': 1,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'}
```

Deleting Resources

HTTP Method	DELETE
HTTP Path	DELETE /api/sites/1/networks/1/ DELETE /api/sites/1/networks/10.0.0.0/24/
Python Client	c.sites(1).networks(1).delete() c.sites(1).networks('10.0.0.0/24').delete()

This one should be straight forward and there is no payload to deal with. Will return bool.

```
c.sites(1).networks('10.0.1.0/24').delete()
# True
```

Querying by Attribute Values

HTTP Method	GET
HTTP Path	GET /api/sites/1/networks/query/?query='set query here'
Python Client	c.sites(1).networks.query.get(query='set query here')

Set queries are the way to filter based on attributes and their values. The syntax is typical set query syntax and is lightly discussed in *Set Queries*.

The query itself is passed as a query param to the /query/ endpoint and can contain regular expressions by suffixing the attribute name, as shown below:

```
# Everything matching exactly desc == test
c.sites(1).networks.query.get(query='desc=test')
# [{u'attributes': {u'dc': u'sfo', u'desc': u'test'},
#   u'id': 2,
#   u'ip_version': u'4',
#   u'is_ip': False,
#   u'network_address': u'10.0.0.0',
#   u'parent_id': 1,
#   u'prefix_length': 24,
#   u'site_id': 1,
#   u'state': u'allocated'}}]

# Everything with desc matching regex test.*
c.sites(1).networks.query.get(query='desc_regex=test.*')
# [{u'attributes': {u'desc': u'testing'},
#   u'id': 1,
#   u'ip_version': u'4',
#   u'is_ip': False,
#   u'network_address': u'10.0.0.0',
#   u'parent_id': None,
#   u'prefix_length': 8,
#   u'site_id': 1,
#   u'state': u'allocated'},
#   {u'attributes': {u'dc': u'sfo', u'desc': u'test'},
```

```
# u'id': 2,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'10.0.0.0',
# u'parent_id': 1,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'},
# {u'attributes': {u'dc': u'chi', u'desc': u'tester'},
# u'id': 4,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'10.0.2.0',
# u'parent_id': 1,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'}}

# Everything NOT dc == chi
c.sites(1).networks.query.get(query='-dc=chi')
# [{u'attributes': {},
# u'id': 7,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'8.8.8.0',
# u'parent_id': None,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'},
# {u'attributes': {u'desc': u'testing'},
# u'id': 1,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'10.0.0.0',
# u'parent_id': None,
# u'prefix_length': 8,
# u'site_id': 1,
# u'state': u'allocated'},
# {u'attributes': {u'dc': u'sfo', u'desc': u'test'},
# u'id': 2,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'10.0.0.0',
# u'parent_id': 1,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'},
# {u'attributes': {u'dc': u'sfo'},
# u'id': 6,
# u'ip_version': u'4',
# u'is_ip': False,
# u'network_address': u'10.0.1.0',
# u'parent_id': 1,
# u'prefix_length': 24,
# u'site_id': 1,
# u'state': u'allocated'},
# {u'attributes': {},
# u'id': 5,
```

```
# u'ip_version': u'4',
# u'is_ip': True,
# u'network_address': u'10.0.2.1',
# u'parent_id': 4,
# u'prefix_length': 32,
# u'site_id': 1,
# u'state': u'allocated'}}

# Everything dc == chi
c.sites(1).networks.query.get(query='dc=chi')
# [{u'attributes': {u'dc': u'chi', u'desc': u'tester'},
#   u'id': 4,
#   u'ip_version': u'4',
#   u'is_ip': False,
#   u'network_address': u'10.0.2.0',
#   u'parent_id': 1,
#   u'prefix_length': 24,
#   u'site_id': 1,
#   u'state': u'allocated'}}]
```

API Abstraction Models

These models were created to abstract most of the API away from the user if they didn't want or need it. An instance can be created by providing minimal info such as CIDR and desired attributes or it can take raw payload from the API and turn it into a model instance.

You can read the docstring in the `pynsot.models` module or follow the links below for usage examples. We think it's a pretty solid way to do most basic interaction.

- `pynsot.models.Network`
- `pynsot.models.Device`
- `pynsot.models.Interface`

API Reference

If you are looking for information on a specific function, class, or method, go forward.

API

This part of the documentation is for API reference

Clients

Simple Python API client for NSoT REST API.

The easiest way to get a client is to call `get_api_client()` with no arguments. This will read the user's `~/.pynsotrc` file and pass the values to the client constructor:

```
>>> from pynsot.client import get_api_client
>>> api = get_api_client()
>>> api
AuthTokenClient(url=http://localhost:8990/api)>
```

```
pynsot.client.get_api_client(auth_method=None, url=None, extra_args=None,
                             use_dotfile=True)
```

Safely create an API client so that users don't see tracebacks.

Any arguments that aren't explicitly passed will be replaced by the contents of the user's dotfile.

Parameters

- **auth_method** – Auth method used by the client
- **url** – API URL
- **extra_args** – Dict of extra keyword args to be passed to the API client class
- **use_dotfile** – Whether to read the dotfile or not.

```
class pynsot.client.BaseClient(base_url=None, **kwargs)
    Magic REST API client for NSoT.
```

error (*exc*)

Take errors and make them human-readable.

Parameters exc – Exception instance

get_auth (**kwargs)

Subclasses should reference kwargs from `self._kwargs`.

Parameters **client** – Client instance. Defaults to `self`.

get_resource (*resource_name*)

Return a single resource object.

Parameters **resource_name** – Name of resource

class `pynsot.client.EmailHeaderClient` (*base_url=None, **kwargs*)
Default client using email auth header method.

authentication_class

alias of `EmailHeaderAuthentication`

class `pynsot.client.AuthTokenClient` (*base_url=None, **kwargs*)
Client that uses `auth_token` method.

authentication_class

alias of `AuthTokenAuthentication`

class `pynsot.client.BaseClientAuth` (*client*)

append_api_version (*request*)

Append API version into Accept header.

class `pynsot.client.EmailHeaderAuthentication` (*client*)
Special authentication that sets the email auth header.

classmethod **get_user** ()

Get the local username, or if root, the sudo username.

class `pynsot.client.AuthTokenAuthentication` (*client*)
Special authentication that utilizes `auth_tokens`.

Adds header for “Authorization: ApiToken {email}:{auth_token}”

get_token (*base_url, email, secret_key*)

Currently ghetto: Hit the API to get an `auth_token`.

Parameters

- **base_url** – API URL
- **email** – User’s email
- **secret_key** – User’s `secret_key`

API Models

API Model Classes

These resource models are derived from `collections.MutableMapping` and, thus, act like dicts and can instantiate with raw resource output from API as well as simplifying by providing `site_id` and (usually) the natural key (`cidr`, `hostname`, etc).

Example

```

>>> from pynsot.models import Network, Device, Interface
>>> from pynsot.client import get_api_client
>>> client = get_api_client()
>>> net = Network(raw=client.networks.get()[-1])
>>>
>>> # Or also...
>>> # >>> net = Network(client=client, site_id=1, cidr='8.8.8.0/24')
>>>
>>> net.exists()
>>> True
>>>
>>> net.existing_resource()
{'attributes': {},
 u'id': 81,
 u'ip_version': u'4',
 u'is_ip': False,
 u'network_address': u'254.0.0.0',
 u'parent_id': 1,
 u'prefix_length': 24,
 u'site_id': 1,
 u'state': u'allocated'}
>>>
>>> net.purge()
True
>>>
>>> net.exists()
False
>>>
>>> net.ensure()
True
>>>
>>> net.exists()
True
>>>
>>> net['site_id']
2
>>>
>>> net['site_id'] = 4
>>>
>>> net.exists()
False
>>>
>>> net['site_id'] = 2
>>>
>>> net.exists()
True

```

class pynsot.models.**Resource** (*site_id=None, client=None, raw=None, attributes=None, **kwargs*)
Base API Abstraction Models Class

Instances of an API abstraction model represent a single NSoT resource and provide methods for managing the state of it upstream. They can be instantiated by the raw returned object from NSoT API or more simply by a few descriptive kwargs.

Resource is a subclass of :class:collections.MutableMapping which makes it act as a dictionary would. The mapping represents the payload that would be accepted by NSoT and can be manipulated as desired like a normal dict.

Subclassing:**Subclasses must adhere to a simple contract:**

- Overload the abstract methods and abstract properties this class uses
- If custom arguments are needed, overload `self.postinit`. Just make sure to call `self.init_payload()` at the end. All kwargs that aren't handled by `self.__init__` are passed here

Parameters

- **site_id** (*int*) – Site ID this resource belongs to. Required unless `raw` is supplied.
- **attributes** (*dict*) – Attributes to add to resource. If supplying `raw`, add these after the instantiation like:

```
>>> obj = Device(raw=RAW_API_DICT)
>>> obj['attributes'] = {}
```

- **client** (`pynsot.client.BaseClient`) – Pynsot client for API interactions. Will be lazily loaded if not provided, but might be cheaper to supply it up front.
- **raw** (*dict*) – Raw NSoT resource object. What would be returned from a GET, POST, PUT, or PATCH operation for a single resource. Gets mapped directly to payload

clear_cache()

Clears state of certain properties

This is ideally done during a write operation against the API, such as `.purge()` or `.ensure()`. Helps prevent representing out-of-date information

ensure()

Ensure object in current state exists in NSoT

By site, make sure resource exists. True if it is or was able to get to the desired state, False if not and logged in `last_error`.

Having this operation be done individually for each resource has some pros and cons. Generally as long as the amount of items is small enough, it's not a huge difference but it is an extra HTTP request.

Sending in bulk halts at the first error and fails the following so it requires more handling.

Cache is cleared first thing and before return.

Return type bool

ensure_client(kwargs)**

Ensure that object has a client object

Client may be passed during `__init__` as a kwarg, but call this before doing any client work to ensure

Parameters **kwargs** (*dict*) – These will be passed to `get_api_client`.

existing_resource()

Returns upstream resource

If nothing exists, empty dict is returned. The result of this is cached in a property (`_existing_resource`) for cheaper re-checks. This can be cleared via `.clear_cache()`

Return type dict

exists()

Does the current resource exist?

Return type bool

identifier

Human-friendly string to represent the resource

Used in log messages and magic methods for comparison. Examples here would be CIDR, hostname, etc

Return type str

init_payload()

Initializes the payload dictionary using resource specific data

log_error(*error*)

Log and append errors to object

This does a check to see if response object available from HTTP request. If not, that's OK too and it'll just append the raw error.

payload

Represents exact payload sent to NSoT server

Returns `_payload`

Return type dict

postinit(*kwargs*)**

Overloadable method for post `__init__`

Use this for things that need to happen post-init, including subclass-specific argument handling.

This method is called at the very end of `__init__` unless `raw` is given.

Params *kwargs* All unhandled `kwargs` from `__init__` are passed here

purge()

Ensure resource doesn't exist upstream

By site, make sure resource is deleted. True if it is or was able to get to the desired state, False if not and logged in `last_error`.

Cache is cleared first thing and before return.

Return type bool

resource

Pynsot client for resource type

Return type `pynsot.client.BaseClient`

resource_name

Name of resource

Must be plural

Return type str

class `pynsot.models.Network` (*site_id=None, client=None, raw=None, attributes=None, **kwargs*)

Network API Abstraction Model

Subclass of Resource.

```
>>> n = Network(cidr='8.8.8.0/24', site_id=1)
>>> n.exists()
False
>>> n.ensure()
True
>>> n.exists()
```

```

True
>>> n.existing_resource()
{'attributes': {},
 u'id': 31,
 u'ip_version': u'4',
 u'is_ip': True,
 u'network_address': u'8.8.8.0',
 u'parent_id': 1,
 u'prefix_length': 24,
 u'site_id': 1,
 u'state': u'assigned'}
>>>
>>> n.purge()
True
>>> n.exists()
False
>>> n.closest_parent()
<Network: 8.8.0.0/16>

```

Parameters `cidr` – CIDR for network

closest_parent()

Returns resource object of the closest parent network

Empty dictionary if no parent network

Returns Parent resource

Return type `pynsot.models.Network` or dict

init_payload()

This will init the payload property

class `pynsot.models.Device` (*site_id=None, client=None, raw=None, attributes=None, **kwargs*)

Device Resource

Subclass of Resource.

```

>>> dev = Device(hostname='router1-nyc', site_id=1)
>>> dev.exists()
False
>>>
>>> dev.ensure()
True
>>>
>>> dev.existing_resource()
{'attributes': {}, u'hostname': u'router1-nyc', u'id': 1, u'site_id': 1}
>>> dev.purge()
True
>>>
>>> dev.exists()
False

```

Parameters `hostname` – Device hostname

class `pynsot.models.Interface` (*site_id=None, client=None, raw=None, attributes=None, **kwargs*)

Interface Resource

Subclass of Resource

Parameters

- **addresses** (*list*) – Addresses on interface
- **description** – Interface description
- **device** (`pynsot.models.Device`) – Required, device interface is on. TODO: broken as currently implemented but will soon reflect the following type
- **type** (*int*) – Interface type as described by SNMP IF-TYPE's
- **mac_address** – MAC of interface
- **name** – Required, name of interface
- **parent_id** (*int*) – ID of parent interface
- **speed** (*int*) – Speed of interface

`attempt_device()`

Attempt to set `device` attribute to its ID if hostname was given

If an ID was provided during init, this happens in init. If a hostname was provided, we need to take care of a couple things.

1. Attempt to fetch ID for existing device by the hostname. If this works, use this ID
2. Should #1 fail, set device id to 0 to signify this device doesn't exist yet, hoping this method will be called again when it's time to create. The thought here is if a user is instantiating lots of resources at the same time, there's a chance the device this relates to is going to be created before this one actually gets called upon.

0 is used instead of a more descriptive message because should the device still not exist when executing this, at least a device doesn't exist error would be returned instead of expecting int not str.

Utilities

`pynsot.util.get_result(response)`

Get the desired result from an API response.

Parameters `response` – Requests API response object

Dotfile

`class pynsot.dotfile.Dotfile(filepath=u'/home/docs/.pynsotrc', **kwargs)`

Create, read, and write a dotfile.

Miscellaneous Pages

TBD

Changelog

Version History

1.1.3 (2017-02-21)

- Fix #119 - Add ability to use set queries on *list* subcommands (#133)

1.1.2 (2017-02-06)

- Add support for strict allocations (added in NSoT v1.1.2)
- Change requirements.txt to use Compatible Release version specifiers

1.1.1 (2017-01-31)

- Corrected the spelling for the `descendants` sub-command on Networks. The old misspelled form of `descendents` will display a warning to users.

1.1.0 (2017-01-30)

- Adds support for Circuit objects (added in NSoT v1.1)

1.0.2 (2017-01-23)

- Bump NSoT requirement to v1.0.13, fix tests that were broken
- Fix #125 - Support natural key when working with interface. Interfaces can now be referred to using *device_name:interface_name* in addition to the unique ID given by the database.

1.0.1 (2016-12-12)

- Network objects are now properly sorted by network hierarchy instead of by alphanumeric order.
- Streamlined the way that objects are displayed by natural key to simplify future feature development.

1.0 (2016-04-27)

- OFFICIAL VERSION 1.0!
- Fully compatible with NSoT REST API version 1

Logo by Vecteezy is licensed under [CC BY-SA 3.0](#)

p

pynsot, 35
pynsot.client, 35
pynsot.models, 36

A

append_api_version() (pynsot.client.BaseClientAuth method), 36
attempt_device() (pynsot.models.Interface method), 41
authentication_class (pynsot.client.AuthTokenClient attribute), 36
authentication_class (pynsot.client.EmailHeaderClient attribute), 36
AuthTokenAuthentication (class in pynsot.client), 36
AuthTokenClient (class in pynsot.client), 36

B

BaseClient (class in pynsot.client), 35
BaseClientAuth (class in pynsot.client), 36

C

clear_cache() (pynsot.models.Resource method), 38
closest_parent() (pynsot.models.Network method), 40

D

Device (class in pynsot.models), 40
Dotfile (class in pynsot.dotfile), 41

E

EmailHeaderAuthentication (class in pynsot.client), 36
EmailHeaderClient (class in pynsot.client), 36
ensure() (pynsot.models.Resource method), 38
ensure_client() (pynsot.models.Resource method), 38
error() (pynsot.client.BaseClient method), 35
existing_resource() (pynsot.models.Resource method), 38
exists() (pynsot.models.Resource method), 38

G

get_api_client() (in module pynsot.client), 35
get_auth() (pynsot.client.BaseClient method), 35
get_resource() (pynsot.client.BaseClient method), 36
get_result() (in module pynsot.util), 41
get_token() (pynsot.client.AuthTokenAuthentication method), 36

get_user() (pynsot.client.EmailHeaderAuthentication class method), 36

I

identifier (pynsot.models.Resource attribute), 39
init_payload() (pynsot.models.Network method), 40
init_payload() (pynsot.models.Resource method), 39
Interface (class in pynsot.models), 40

L

log_error() (pynsot.models.Resource method), 39

N

Network (class in pynsot.models), 39

P

payload (pynsot.models.Resource attribute), 39
postinit() (pynsot.models.Resource method), 39
purge() (pynsot.models.Resource method), 39
pynsot (module), 35
pynsot.client (module), 35
pynsot.models (module), 36

R

Resource (class in pynsot.models), 37
resource (pynsot.models.Resource attribute), 39
resource_name (pynsot.models.Resource attribute), 39