Network Source of Truth Documentation

Release 1.1.3

Gary M. Josack

May 26, 2017

Contents

Network Source of Truth (NSoT) a source of truth database and repository for tracking inventory and metadata of network entities to ease management and automation of network infrastructure.

NSoT is an API-first application that provides a REST API and a web application front-end for managing IP addresses (IPAM), network devices, and network interfaces.

Contents:

CHAPTER 1

Quick Start

Network Source of Truth is super easy to get running. If you just can't wait to skip ahead, this guide is for you.

Note: This quick start assumes a lot. If it doesn't work for you, please skip this and read the Installation guide.

1. Install NSoT:

```
$ pip install nsot
```

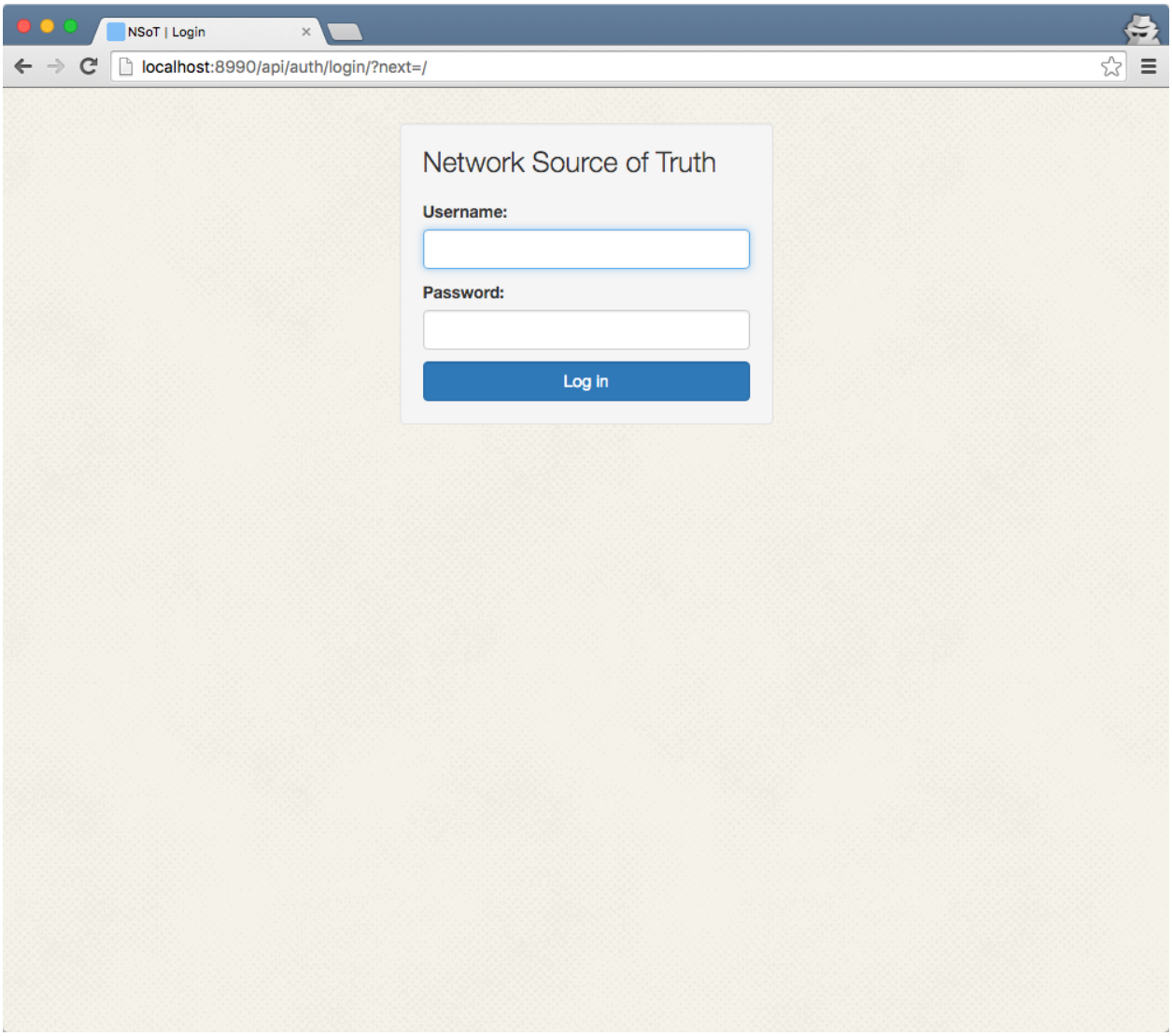
2. Initialize the config (this will create a default config in `~/.nsot/nsot.conf.py`):

```
$ nsot-server init
```

3. Start the server on `8990/tcp` (the default) and create a superuser when prompted:

```
$ nsot-server start
```

4. Now fire up your browser and visit http://localhost:8990!



5. Use the username/password created in step 3 to login.

Now, head over to the Tutorial to start getting acquainted with NSoT!

CHAPTER 2

Installation

Dependencies

Network Source of Truth (NSoT) should run on any Unix-like platform that has:

- Python 2.7
- pip

Python dependencies

If you install using pip (which you should) these will be installed for you automatically. For the brave, check out the contents of `requirements`.

Platform-Specific Installation Instructions

These guides go into detail on how to install NSoT on a given platform.

CentOS

This installation guide assumes that you have installed CentOS 6.4 on your machine, and are wanting to install NSoT. This guide will help you install NSoT and then run it locally from a browser window.

Installation

To ensure your CentOS installation is up to date, please update it. Once complete, open a command prompt and run the following:

```
$ sudo yum install -y openssl-devel python-devel libffi-devel gcc-plugin-devel
$ sudo yum install -y epel-release
$ sudo yum install -y python-pip
```

Next you'll need to upgrade Pip to the latest version with some security addons:

```
$ sudo pip install --upgrade pip
$ sudo pip install requests[security]
```

Now we are ready to Pip install NSoT:

```
$ sudo pip install nsot
```

Now you are ready to follow the Quick Start starting at step 2!

Fedora

This installation guide assumes that you have installed Fedora 22 on your machine, and are wanting to install NSoT. This guide will help you install NSoT and then run it locally from a browser window.

Installation

To ensure your Fedora installation is up to date, please update it. Once complete, open a command prompt and run the following:

```
$ sudo dnf -y install gcc gcc-c++ libffi libffi-devel python-devel openssl-devel
$ sudo dnf -y gcc-plugin-devel make automake kernel kernel-devel psmisc
$ sudo dnf -y install python2-devel
```

Next you'll need to upgrade Pip to the latest version:

```
$ sudo pip install --upgrade pip
```

Now we are ready to install NSoT:

```
$ sudo pip install nsot
```

Now you are ready to follow the Quick Start starting at step 2!

Mac OS X

This tutorial is designed to get a working version os NSoT installed inside a Mac OS X system. We will install virtual environment wrappers to keep things tidy, and isolate our installation.

Installation

Xcode

It is assumed that you have a Mac, running OS X (written for 10.10.5), and xcode already installed. If you don't have Xcode, please install it. You may need to agree to a command line license. We suggest running this via command line prior to install:

```
$ xcodebuild -license
```

Prerequisites

We will put our installation of NSoT inside a Python virtual environment to isolate the installation. To do so we need virtualenv, and virtualenvwrapper. Open a command prompt, and install them with pip:

```
$ pip install virtualenv
$ pip install virtualenvwrapper
```

Ensure installation by running a which, and finding out where they now live:

```
$ which virtualenvwrapper
$ which virtualenv
```

Next we tell bash where these virtual environments are, and where to save the associated data:

```
$ vi ~/.bashrc
```

Add these three lines:

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

Now restart bash to implement the changes:

```
$ source ~/.bashrc
```

Install NSoT

NSoT will be installed via command line, into the folder of your choice. If you don't have a preffered folder, may we suggest this:

```
$ mkdir ~/sandbox && cd ~/sandbox
```

CD into the folder, make a virtual environment, and start it:

```
$ mkvirtualenv nsot
$ pip install
```

Once in the folder of choice, install NSoT:

```
$ pip install nsot
```

Now you are ready to follow the Quick Start starting at step 2!

SuSe

This installation guide assumes that you have installed SuSe 13 on your machine, and are wanting to install NSoT. This guide will help you install NSoT and then run it locally from a browser window.

Installation

To ensure your SuSe installation is up to date, please update it. We'll begin by opening a command prompt. Make sure your certificates are properly installed, or use this certificate:

```
$ wget --no-check-certificate 'https://raw.githubusercontent.com/mitchellh/vagrant/master/keys/vagra
```

Now we'll install the prerequisite software with zypper:

```
$ sudo zypper --non-interactive in python-devel gcc gcc-c++ git libffi48-devel libopenssl-devel pytho
```

Next you'll need to upgrade Pip and security addons:

```
$ sudo pip install --upgrade pip
$ sudo pip install requests[security]
```

Now we are ready to install NSoT:

```
$ sudo pip install nsot
```

SuSe Firewall

To access NSoT from a local browser we'll need to turn off the security for this demo:

```
$  sudo /sbin/service SuSEfirewall2_setup stop
```

For production installations we reccomend adding a rule to your iptables for NSoT on ports `8990/tcp` (or the port of your choosing).

Now you are ready to follow the Quick Start starting at step 2!

Ubuntu

This installation guide assumes that you are running Ubuntu version 12.04, 14.04, or 16.04 on your machine, and are wanting to install NSoT. This guide will help you install NSoT and then run it locally from a browser window.

Installation

To ensure your Ubuntu installation is up to date, please update it. Open a command prompt and run the following:

```
$ sudo apt-get -y update
```

Once your machine is up to date, we need to install development libraries to allow NSoT to build:

```
$ sudo apt-get -y install build-essential python-dev libffi-dev libssl-dev
```

The Python Pip installer and the git repository management tools are needed too. We'll go ahead and get those next:

```
$ sudo apt-get --yes install python-pip git
```

```
$ sudo pip install nsot
```

Now you are ready to follow the Quick Start starting at step 2!

Virtual Machine Install Instructions

These guides go into detail on how to get running NSoT on virtual machines.

Docker

Want to use Docker? More on this later. For now you may look at the `docker` directory at the top of the repository on GitHub, or if you're feeling plucky, check out the contents of `../dockerfile`.

Quick start

```
$ cd docker
$ docker run -p 8990:8990 -d --name=nsot nsot/nsot start --noinput
```

README

Here is the readme until we clean up these docs and include them for real here.

```
# NSoT Docker Image

This Docker image runs NSoT. Perfect for quick developing and even deploying in
production.

## Using this image

`nsot-server --config=/etc/nsot/nsot.conf.py` is the image entrypoint, so the
command passed to docker run becomes CLI parameters. This is equivalent to what
the default is:

```
$ docker run -p 8990:8990 -d --name=nsot nsot/nsot start --noinput
```

Image tags should correspond with NSoT release version numbers. Basic usage is
like:

```bash

$ NSOT_SECRET='X9HqplzM_0E3Ghf3QOPDnO2k5VpVHkfzsZsVer4OeKA='
$ docker run -p 8990:8990 -d --name=nsot -e NSOT_SECRET=$NSOT_SECRET nsot/nsot:1.0.10
```

Getting started

With the docker container running, you need to create a superuser
From the command above, create the super user as follows:

```bash
$ docker exec -it nsot bash
# nsot-server --config=/etc/nsot/nsot.conf.py createsuperuser --email your@email.here
```

This will prompt you for a password, which you can then use to log into http://dockerhost:8990/

If you have an established database and you don't wish to attempt to upgrade it
then you'll need to specify `--no-upgrade`

If you wanted to do interactive debugging, use the docker run flags `-ti` and
pass the relevant options:

```bash
$ docker run -p 8990:8990 -ti --rm nsot/nsot dbshell
    SQLite version 3.8.2 2013-12-06 14:53:30
    Enter ".help" for instructions
    Enter SQL statements terminated with a ";"
    sqlite> exit

OR

$ docker run -p 8990:8990 -ti --rm nsot/nsot shell_plus
    # Shell Plus Model Imports
    from django.contrib.admin.models import LogEntry
    from django.contrib.auth.models import Group, Permission
```
```

```
    from django.contrib.contenttypes.models import ContentType
    from django.contrib.sessions.models import Session
    from nsot.models import Assignment, Attribute, Change, Device, Interface,
    Network, Site, User, Value
    # Shell Plus Django Imports
    from django.utils import timezone
    from django.conf import settings
    from django.core.cache import cache
    from django.db.models import Avg, Count, F, Max, Min, Sum, Q, Prefetch
    from django.core.urlresolvers import reverse
    from django.db import transaction
    Python 2.7.6 (default, Jun 22 2015, 17:58:13)
    Type "copyright", "credits" or "license" for more information.

    IPython 3.1.0 -- An enhanced Interactive Python.
    ?         -> Introduction and overview of IPython's features.
    %quickref -> Quick reference.
    help      -> Python's own help system.
    object?   -> Details about 'object', use 'object??' for extra details.

    In [1]:
```

If you want to add an entire custom config, volume mount it to
`/etc/nsot/nsot.conf.py`

Ports

Only TCP 8990 is exposed

Environment Variables

Pass these with `-e` to control the configuration. `NSOT_SECRET` should be the
bare minimum set, setting an external DB if in production or wanting
persistence should be second.

Note that the `NSOT_SECRET` must be 32 url-safe base64-encoded bytes. You may
generate one by executing this:

```
python -c "import base64, os; print base64.urlsafe_b64encode(os.urandom(32))"
```

Variable	Default Value
`DB_ENGINE`	`django.db.backends.sqlite3`
`DB_NAME`	`nsot.sqlite3`
`DB_USER`	`nsot`
`DB_PASSWORD`	''
`DB_HOST`	''
`DB_PORT`	''
`NSOT_EMAIL`	`X-NSoT-Email`
`NSOT_SECRET`	`nJvyRB8tckUWvquJZ3ax4QnhpmqTgVX2k3CDY13yK9E=`

Contributing

This image is maintained upstream under `docker/Dockerfile.sub` template.
Changes to `docker/Dockerfile` will be overwritten during the next version

```
bump.
```

Vagrant

The *Vagrantfile* in the root of this repo creates a fresh Vagrant box running Ubuntu and NSoT.

Prerequisites

To proceed you must have working installations of Vagrant and Virtualbox on your machine. If you already have these, you may skip this step.

If you do not have a working Vagrant environment configured along with Virtualbox, please follow the Vagrant's "Getting Started" instructions before proceeding.

Instructions

Provision the box

5-10 minutes on a fast connection

To provision the virtual machine open a command prompt, and run the following command from this directory:

```
$ vagrant up
```

This will build a new Vagrant box, and pre-install NSoT for you.

Launch NSoT

Login to the new virtual machine via ssh:

```
$ vagrant ssh
```

Start the server on `8990/tcp` (the default) and create a superuser when prompted:

```
$ nsot-server start
```

Point your browser to http://192.168.33.11:8990 and login!

Now you are ready to follow the Tutorial to start playing around.

Official Client

We maintain the official NSoT client under a separate project called pyNSoT. PyNSoT provides a Python API client and an excellent CLI utility.

If you wish to utilize NSoT from the command-line, or follow along in the Tutorial, you're going to need this!

Installing the client is as easy as running `pip install pynsot`. Setup is a breeze, too. If you run into any issues, please refer to the official pyNSoT documentation.

Demo

If you would like to run the demo, make sure you've got NSoT installed and that you have a fresh clone of the NSoT repository from GitHub.

If you don't already have a clone, clone it and change into the `nsot` directory:

```
$ git clone https://github.com/dropbox/nsot
$ cd nsot
```

Then to switch to the `demo` directory and fire up the demo:

```
$ cd nsot/demo
$ ./run_demo.sh
```

The demo will be available at http://localhost:8990/

CHAPTER 3

Configuration

Configuring NSoT

This section describes how to get started with configuring the NSoT server.

Initializing the Configuration

You may generate an initial configuration by executing `nsot-server init`. By default the file will be created at `~/.nsot/nsot.conf.py`. You may specify a different location for the configuration as the argument to `init`:

```
nsot-server init /etc/nsot.conf.py
```

Specifying your Configuration

If you do not wish to utilize the default location, you must provide the `--config` argument when executing `nsot-server` so that it knows where to find it. For example, to start the server with the configuration in an alternate location:

```
nsot-server --config=/etc/nsot.conf.py start
```

You may also set the `NSOT_CONF` enviroment variable to the location of your configuration file so that you don't have to provide the `--config` argument:

```
$ export NSOT_CONF=/etc/nsot.conf.py
$ nsot-server start
```

Sample Configuration

Below is a sample configuration file that covers the primary settings you may care about, and their default values.

```
"""
This configuration file is just Python code. You may override any global
defaults by specifying them here.

For more information on this file, see
https://docs.djangoproject.com/en/1.8/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/1.8/ref/settings/
"""


from nsot.conf.settings import *
import os.path


# Path where the config is found.
CONF_ROOT = os.path.dirname(__file__)

# A boolean that turns on/off debug mode. Never deploy a site into production
# with DEBUG turned on.
# Default: False
DEBUG = False

#############
# Database #
#############
# https://docs.djangoproject.com/en/dev/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(CONF_ROOT, 'nsot.sqlite3'),
        'USER': 'nsot',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}

###############
# Application #
###############

# The address on which the application will listen.
# Default: localhost
NSOT_HOST = 'localhost'

# The port on which the application will be accessed.
# Default: 8990
NSOT_PORT = 8990

# The number of gunicorn worker processes for handling requests.
# Default: 4
NSOT_NUM_WORKERS = 4
```

```
# Timeout in seconds before gunicorn workers are killed/restarted.
# Default: 30
NSOT_WORKER_TIMEOUT = 30

# If True, serve static files directly from the app.
# Default: True
SERVE_STATIC_FILES = True

#############
# Security #
#############

# A URL-safe base64-encoded 32-byte key. This must be kept secret. Anyone with
# this key is able to create and read messages. This key is used for
# encryption/decryption of sessions and auth tokens. A unique key is randomly
# generated for you when you utilize ``nsot-server init``
# https://cryptography.io/en/latest/fernet/#cryptography.fernet.Fernet.generate_key
SECRET_KEY = u'fMK68NKgazLCjjTXjDtthhoRUS8IV4lwD-9G7iVd2Xs='

# Header to check for Authenticated Email. This is intended for use behind an
# authenticating reverse proxy.
USER_AUTH_HEADER = 'X-NSoT-Email'

# The age, in seconds, until an AuthToken granted by the API will expire.
# Default: 600
AUTH_TOKEN_EXPIRY = 600  # 10 minutes

# A list of strings representing the host/domain names that this Django site can
# serve. This is a security measure to prevent an attacker from poisoning caches
# and triggering password reset emails with links to malicious hosts by
# submitting requests with a fake HTTP Host header, which is possible even under
# many seemingly-safe web server configurations.
# https://docs.djangoproject.com/en/1.8/ref/settings/#allowed-hosts
ALLOWED_HOSTS = ['*']

##############
# Interfaces #
##############

# The default format for displaying MAC addresses. This defaults to
# ":"-separated and expanded (e.g. '00:00:00:00:00:00')
MACADDRESS_DEFAULT_DIALECT = 'macaddress.mac_linux'

# The default speed in Mbps for newly device interfaces if not otherwise
# specified.
INTERFACE_DEFAULT_SPEED = 1000  # In Mbps (e.g. 1Gbps)

# Whether to compress IPv6 for display purposes, for example:
# - Default: 2620:0100:6000:0000:0000:0000:0000:0000/40
# - Compressed: 2620:100:6000::/40
# Default: True
NSOT_COMPRESS_IPV6 = True

# Temp debug logging
if os.getenv('NSOT_DEBUG'):
    DEBUG = True
    LOGGING['loggers']['nsot']['level'] = 'DEBUG'
    LOGGING['loggers']['django.db.backends'] = {
```

```
        'handlers': ['console'],
        'level': 'DEBUG'
    }
```

Advanced Configuration

This section covers additional configuration options available to the NSoT server and advanced configuration topics.

Database

NSoT defaults to utilizing SQLite as a database backend, but supports any database backend supported by Django. The default backends available are SQLite, MySQL, PostgreSQL, and Oracle.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'nsot.sqlite3',
    }
}
```

For more information on configuring the database, please see the official Django database documentation.

Caching

Note: At this time only Interface objects are cached if caching is enabled!

NSoT includes built-in support for caching of API results. The default is to use to the "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

The cache is invalidated on any update or delete of an object. Caching can dramatically perform read operations of databases with a large amount of network Interface objects.

If you need caching, see the official Django caching documentation on how to set it up.

CHAPTER 4

Tutorial

Here's how to use NSoT.

This document assumes that you've already have an instance of NSoT installed, configured, and running on your system. If you don't, please either check out the Quick Start or head over to the full-blown Installation guide and then return here.

First Steps

Important: Because this is a work-in-progress, we're going to use the command-line utility provided by the official NSoT client to get you acquainted with NSoT.

Install the Client

First things first, you'll need to install pyNSoT, the official Python API client and CLI utility:

```
$ pip install pynsot
```

Configure the Client

After you've installed the client, please follow the pyNSoT Configuration guide to establish a `.pynsotrc` file.

Using the Command-Line

Once you've got a working `nsot` CLI setup, please follow the pyNSoT Command-Line guide. This will get you familiarized with the basics of how NSoT works.

Understanding the Data Model

NSoT has a relatively simple data model, but the objects themselves can be quite sophisticated. Familiarize yourself with the Data Model.

Using the REST API

Familiarize yourself with the basics of the REST API.

Administering the Server

Familiarize with the `nsot-server` command that is used to manage your server instance by checking out the Server Administration guide.

CHAPTER 5

Server Administration

Here's how to administer NSoT using the `nsot-server` command.

Important: As NSoT is built using Django there are a number of commands available that we won't cover here. Any commands specific to operating NSoT, however, will be covered here in detail.

Getting help

To see all available commands:

```
$ nsot-server help
```

Additionally, all commands have a `-h/--help` flag for all available options and arguments.

Initialize the configuration

Create a new configuration in `~/.nsot/nsot.conf.py`.

```
$ nsot-server init
Configuration file created at '/Users/jathan/.nsot/nsot.conf.py'
```

Alternately, you may specify a path for the file by providing it as an argument.

```
$ ./nsot-server init myconfig.py
Configuration file created at 'myconfig.py'
```

Create a superuser

You need at least one superuser to administer the system.

```
$ nsot-server createsuperuser --email admin@localhost
Password:
Password (again):
Superuser created successfully.
```

Start the server

This starts the built-in WSGI server using gevent + gunicorn. There are a ton of options. Use `--h/--help` to see them all!

Note: Many of the options fallback to global defaults specificed in your `settings.py` if they are not provided at the command-line. Please see the *Configuration* guide for customizing the defaults.

```
$ nsot-server start
Performing upgrade before service startup...
Performing collectstatic before service startup...

0 static files copied to '/Users/jathan/sandbox/virtualenvs/nsot/lib/python2.7/site-packages/nsot/sta
Running service: 'http', num workers: 4, worker timeout: 30
[2016-04-29 02:52:39 -0500] [21840] [INFO] Starting gunicorn 19.3.0
[2016-04-29 02:52:39 -0500] [21840] [INFO] Listening at: http://127.0.0.1:8990 (21840)
[2016-04-29 02:52:39 -0500] [21840] [INFO] Using worker: gevent
[2016-04-29 02:52:39 -0500] [21843] [INFO] Booting worker with pid: 21843
[2016-04-29 02:52:39 -0500] [21844] [INFO] Booting worker with pid: 21844
[2016-04-29 02:52:39 -0500] [21845] [INFO] Booting worker with pid: 21845
[2016-04-29 02:52:39 -0500] [21846] [INFO] Booting worker with pid: 21846
```

Upgrade the database

This will initialize a new database or run any pending database migrations to an existing database.

```
$ nsot-server upgrade
Operations to perform:
  Synchronize unmigrated apps: django_filters, staticfiles, messages, smart_selects, rest_framework_s
  Apply all migrations: admin, contenttypes, nsot, auth, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying nsot.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying nsot.0002_auto_20150810_1718... OK
  Applying nsot.0003_auto_20150810_1751... OK
  Applying nsot.0004_auto_20150810_1806... OK
  Applying nsot.0005_auto_20150810_1847... OK
  Applying nsot.0006_auto_20150810_1947... OK
  Applying nsot.0007_auto_20150811_1201... OK
  Applying nsot.0008_auto_20150811_1222... OK
  Applying nsot.0009_auto_20150811_1245... OK
```

```
  Applying nsot.0010_auto_20150921_2120... OK
  Applying nsot.0011_auto_20150930_1557... OK
  Applying nsot.0012_auto_20151002_1427... OK
  Applying nsot.0013_auto_20151002_1443... OK
  Applying nsot.0014_auto_20151002_1653... OK
  Applying nsot.0015_move_attribute_fields... OK
  Applying nsot.0016_move_device_data... OK
  Applying nsot.0017_move_network_data... OK
  Applying nsot.0018_move_interface_data... OK
  Applying nsot.0019_move_assignment_data... OK
  Applying nsot.0020_move_value_data... OK
  Applying nsot.0021_remove_resource_object... OK
  Applying nsot.0022_auto_20151007_1847... OK
  Applying nsot.0023_auto_20151008_1351... OK
  Applying nsot.0024_network_state... OK
  Applying nsot.0025_value_site... OK
  Applying sessions.0001_initial... OK
```

Reverse proxy

Start an authenticating reverse proxy for use in development.

You must install MrProxy first: `pip install mrproxy`.

```
$ nsot-server user_proxy
```

Generate a secret_key

Generate a URL-safe base64-encoded 36-byte secret key suitable for use inside of `settings.py`. This key is used for encryption/decryption of sessions and API auth tokens.

Note: A unique key is randomly generated for you when you utilize `nsot-server init`.

This must be kept secret! Anyone with this key is able to create and read messages.

```
$ nsot-server generate_key
R2gasBVJKmU5ZgkrlBljyZJrLP_B6EwZ3S7k28-SkIs=
```

Python shell

This will drop you into an interactive iPython shell with all of the database models and various other utilities already imported for you. This is immensely useful for direct access to manipulating database objects.

> **Warning:** This is an advanced feature that gives you direct access to the Django ORM database models. Use this very cautiously as you can cause irreparable damage to your NSoT installation.

```
$ nsot-server shell_plus
# Shell Plus Model Imports
from django.contrib.admin.models import LogEntry
```

```
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType
from django.contrib.sessions.models import Session
from nsot.models import Assignment, Attribute, Change, Device, Interface, Network, Site, User, Value
# Shell Plus Django Imports
from django.utils import timezone
from django.conf import settings
from django.core.cache import cache
from django.db.models import Avg, Count, F, Max, Min, Sum, Q, Prefetch
from django.core.urlresolvers import reverse
from django.db import transaction
Python 2.7.8 (default, Oct 19 2014, 16:02:00)
Type "copyright", "credits" or "license" for more information.

IPython 3.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Database shell

This will drop you to a shell for your configured database. This can be very handy for troubleshooting database issues.

> **Warning:** This is an advanced feature that gives you direct access to the database to run raw SQL queries. database. Use this very cautiously as you can cause irreparable damage to your NSoT installation.

```
$ nsot-server dbshell
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite>
```

CHAPTER 6

Data Model

The Network Source of Truth is composed of various object types with which it is important to be familiarized. This document describes each object type.

Sites

Sites function as unique namespaces that can contain all other objects. Sites allow an organization to have multiple instances of potentially conflicting objects. For example, this could be beneficial for isolating corporate vs. production environments, or pulling in the IP space of an acquisition.

Every object must be related to a site and therefore the `site_id` field is used frequently to scope object lookups.

A Site cannot be deleted unless it contains no other objects.

A typical Site object might look like this:

```
{
    "id": 1,
    "name": "Demo Site",
    "description": "This is a demonstration site for NSoT."
}
```

Attributes

Attributes are arbitrary key/value pairs that can be assigned to various resources. Attributes have various flags and constraints to control how they may be used.

Attributes are bound to a *resource name* (e.g. Device). You may have multiple Attributes with the same name bound to different resource types.

When assigned to objects, think of an Attribute as an instance of of an Attribute object with a Value object assigned to it. Objects may be looked up by their attribute/value pairs using set queries.

Attribute/value pairs are cached locally on on the containing object on write to improve read performance.

A typical Attribute object might look like this:

```
{
    "multi": false,
    "resource_name": "Device",
    "description": "The device manufacturer.",
```

```
    "display": true,
    "required": true,
    "site_id": 1,
    "id": 2,
    "constraints": {
        "pattern": "",
        "valid_values": [
            "arista",
            "cisco",
            "juniper"
        ],
        "allow_empty": false
    },
    "name": "vendor"
}
```

Important: Changes to attribute flags and constraints are not retroactive. Existing resources will not be forcefully validated until updated.

Values

Values contain atttribute values. These are never directly manipulated, but they are accessible from the API for utilty.

All attribute values must be strings. If an attribute is a list type (`multi=True`), then the values for that attribute will be a list of strings.

A typical Value object might look like:

```
{
    "id": 8,
    "name": "owner",
    "value": "jathan",
    "attribute": 5,
    "resource_name": "Device",
    "resource_id": 2
}
```

Flags

required If an attribute is required then additions/updates for that resource will require that attribute be present.

display Whether to display the attribute in the web UI. Required attributes are always displayed.

multi Whether the attribute values should be treated as a list type

Constraints

pattern A regex pattern. If set, values for this attribute must match the pattern.

allow_empty Whether the attribute should require a value. This causes the attribute to behave like a tag.

valid_values Valid values for this attribute. This causes the attribute to behave like an enum.

Set Queries

All Resource types support set query operations. Set queries are a powerful part of the data model that allow you to perform complex lookups of objects by attribute/value pairs.

Set queries can be performed using a simple string-based syntax.

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, when using set queries to lookup Device objects:

- `"vendor=juniper"` would return the set intersection of objects with `vendor=juniper`.
- `"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`).
- `"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).

For how set queries can be performed, please see the REST API documentation on *Performing Set Queries*.

Resources

A Resource object is any object that can have attributes. The primary resource types are:

Devices

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

Devices in their most basic form are represented by a hostname.

Devices can contain zero or more Interfaces.

A typical Device object might look like:

```
{
    "attributes": {
        "owner": "jathan",
        "vendor": "juniper",
        "hw_type": "router",
        "metro": "lax"
    },
    "hostname": "lax-r1",
    "site_id": 1,
```

```
    "id": 1
}
```

Networks

Networks in NSoT are designed to provide IP Address Management (IPAM) features. A Network represents an IPv4 or IPv6 Network or IP address. Working with networks is usually done with CIDR notation.

Networks may be assigned to Interfaces by way of an *Assignment* relationship.

A typical Network object might look like:

```
{
    "parent_id": null,
    "state": "allocated",
    "prefix_length": 8,
    "is_ip": false,
    "ip_version": "4",
    "network_address": "10.0.0.0",
    "attributes": {
        "type": "internal"
    },
    "site_id": 1,
    "id": 1
}
```

Tree Traversal

Networks are represented as tree objects. Anytime a network is added or deleted, the tree is automatically updated to reparent networks appropriately.

Networks support all of the common tree traversal methods that you may expect from this type of object:

parent The parent of this network

ancestors All parents of the parent of this network

siblings Networks with the same parent as this network

children The child networks of this network

descendents Deprecated since version 1.1.

Use *descendants* instead, which is the correctly spelled version of the same method.

descendants All children of the children of this network

closest_parent If this network doesn't exist, who might its parent be if it did?

subnets Subnetworks of this network

supernets Supernets of this network

State

Network state represents whether the Network is in use or not. The states are:

allocated The default state for any newly-created Network. It is implied that this address is in use some how, but it is not a busy state.

assigned Used to represent a Network assigned to an Interface. This is a busy state.

reserved Used to represent that the Network is reserved for future use. This is a busy state.

orphaned Used to represent a Network that was previously assigned or reserved but has since drifted.

Allocation

Networks can be used to allocate child networks or addresses.

next_network Given a prefix_length, return the next available child Network of this length.

next_address Given a number of addresses, return that many next available IP addresses.

Interfaces

An Interface represents a physical or logical network interface such as an ethernet port. Interfaces must always be associated with a device. Zero or more addresses may be assigned to an Interface, although the same address may not be assigned to more than one interface on the same device.

A typical Interface object might look like:


```
{
    "addresses": [
        "10.10.10.1/32"
    ],
    "device": 1,
    "speed": 10000,
    "networks": [
        "10.10.10.0/24"
    ],
    "description": "this is eth0",
    "name": "eth0",
    "id": 1,
    "parent_id": null,
    "mac_address": null,
    "attributes": {
        "vlan": "100"
    },
    "type": 6
}
```


Addresses

An address assignment to an Interface is represented by an *Assignment* relationship to a Network object.

If a Network object for the desired IP address assignment does not exist at the time of assignment, one is created and set to the state `assigned`.

If a Network object already exists and is not in a "busy state", then it will be assigned to the Interface.

Assignments

Assignments represent the relationship and constraints for a Network to be associated to an Interface.

The following constraints are enforced:

- An address may not be assigned to a to more than one Interface on any given Device.
- Only a Network containing a host address with a prefix of `/32` (IPv4) or `/128` (IPv6) may be assigned to an Interface.

Networks

The networks for an Interface are the are read-only representation of the derived parent Network objects of any addresses assigned to an Interface.

Changes

All Create/Update/Delete events are logged as a Change. A Change includes information such as the change time, user, and the full object payload after modification.

Changes are immutable and can only be removed by deleting the entire Site.

A typical Change object might look like:


```
{
    "event": "Create",
    "change_at": 1460994054,
    "resource_name": "Attribute",
    "resource": {
        "multi": false,
        "resource_name": "Interface",
        "description": "",
        "required": false,
        "site_id": 1,
        "display": false,
        "constraints": {
            "pattern": "",
            "valid_values": [],
            "allow_empty": false
        },
        "id": 9,
        "name": "foo"
    },
    "user": {
        "id": 1,
        "email": "admin@localhost"
    },
    "resource_id": 9,
    "id": 36,
    "site": {
        "description": "This is a demonstration site for NSoT.",
        "id": 1,
        "name": "Demo Site"
    }
}
```


Users

Users are for logging into stuff. Users in NSoT are represented by an email address.

Users have a "secret key" that can be used for API authentication.

A typical User might look like:

```
{
    "id": 1,
    "email": "admin@localhost",
    "permissions": {
        "1": {
            "user_id": 1,
            "site_id": 1,
            "permissions": [
                "admin"
            ]
        }
    }
}
```

Permissions

Permissions, like other objects, are specific to Sites. There are no permissions that cross over sites. All objects are readable regardless of permissions. There is currently only one type of permissions a User can have in order to make modifications:

- admin
  - Ability to Update/Delete Site
  - Ability to grant permissions within a site
  - All subsequent permissions

Site creation is open to all users. Upon creating a Site you become an admin of that Site with full permissions.

CHAPTER 7

API Reference

If you are looking for information on how to utilize the REST API, or a specific function, class or method, this part of the documentation is for you.

REST API

NSoT is designed as an API-first application so that all possible actions are published as API endpoints.

API Reference

Interactive API reference documentation can be found by browsing to `/docs/` on a running NSoT server instance.

Browsable API

Because NSoT is an API-first application, the REST API is central to the experience. The REST API can support JSON or can also be used directly from your web browser. This version is called the "browsable API" and while it doesn't facilitate automation, it can be very useful.

Visit `/api/` in your browser on your installed instance. How cool is that?!

Authentication

Two methods of authentication are currently supported.

User Authentication Header

This is referred to internally as **auth_header** authentication.

In normal operation NSoT is expected to be run behind an authenticating proxy that passes back a specific header. By default we expect `X-NSoT-Email`, though it is configurable using the `USER_AUTH_HEADER` setting.

The value of this header must be the user's `email` and is formatted like so:

```
X-NSoT-Email: {email}
```

AuthToken

This is referred to internally as **auth_token** authentication.

API authentication requires the `email` and `secret_key` of a user. When a user is first created, a `secret_key` is automatically generated. The user may obtain their `secret_key` from the web interface.

Users make a POST request to `/api/authenticate/` to passing `email` and `secret_key` in `JSON` payload. They are returned an `auth_token` that can then be used to make API calls. The `auth_token` is short-lived (default is 10 minutes and can be change using the `AUTH_TOKEN_EXPIRY` setting). Once the token expires a new one must be obtained.

The `auth_token` must be sent to the API using an `Authorization` header that is formatted like so:

```
Authorization: AuthToken {email}:{secret_key}
```

Requests

In addition to the authentication header above all `POST`, `PUT`, and `PATCH`, requests will be sent as `JSON` rather than form data and should include the header `Content-Type: application/json`

`PUT` requests are of note as they are expected to set the state of all mutable fields on a resource. This means if you don't specificy all optional fields may revert to their default values, depending on the object type.

`PATCH` allows for partial update of objects for most fields, depending on the object type.

`OPTIONS` will provide the schema for any endpoint.

Responses

All responses will be in format along with the header `Content-Type: application/json` set.

The `JSON` payload will be in one of two potential structures and will always contain a `status` field to distinguish between them. If the `status` field has a value of `"ok"`, then the request was successful and the response will be available in the `data` field.

```
{
    ...
}
```

If the `status` field has a value of `"error"` then the response failed in some way. You will have access to the error from the `error` field which will contain an error `code` and `message`.

```
{
    "error": {
        "code": 404,
        "message": "Resource not found."
    }
}
```

Pagination

All responses that return a list of resources will support pagination. If the `results` object on the response has a `count` attribute then the endpoint supports pagination. When making a request against this endpoint `limit` and `offset` query parameters are supported.

The response will also include `next` and `previous` URLs that can be used to retrieve the next set of results. If there are not any more results available, their value will be `null`.

An example response for querying the `sites` endpoint might look like:

Request:

```
GET http://localhost:8990/api/sites/?limit=1&offset=0
```

Response:

```
{
    "count": 1,
    "next": "http://localhost:8990/api/sites/?limit=1&offset=1",
    "previous": null,
    "results": [
        {
            "id": 1
            "name": "Site 1",
            "description": ""
        }
    ]
}
```

Schemas

By performing an `OPTIONS` query on any endpoint, you can obtain the schema of the resource for that endpoint. This includes supported content-types, HTTP actions, the fields allowed for each action, and their attributes.

An example response for the schema for the `devices` endpoint might look like:

Request:

```
OPTIONS http://localhost:8990/api/devices/
```

Response:

```
HTTP 200 OK
Allow: GET, POST, PUT, PATCH, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "name": "Device List",
    "description": "API endpoint that allows Devices to be viewed or edited.",
    "renders": [
        "application/json",
        "text/html"
    ],
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ],
    "actions": {
        "PUT": {
            "id": {
                "type": "integer",
                "required": false,
```

```
                "read_only": true,
                "label": "ID"
            },
            "hostname": {
                "type": "string",
                "required": true,
                "read_only": false,
                "label": "Hostname",
                "max_length": 255
            },
            "attributes": {
                "type": "field",
                "required": true,
                "read_only": false,
                "label": "Attributes",
                "help_text": "Dictionary of attributes to set."
            }
        },
        "POST": {
            "hostname": {
                "type": "string",
                "required": true,
                "read_only": false,
                "label": "Hostname",
                "max_length": 255
            },
            "attributes": {
                "type": "field",
                "required": false,
                "read_only": false,
                "label": "Attributes",
                "help_text": "Dictionary of attributes to set."
            },
            "site_id": {
                "type": "integer",
                "required": true,
                "read_only": false,
                "label": "Site id"
            }
        }
    }
}
```


Performing Set Queries

Set Queries allow you to perform complex lookups of objects by attribute/value pairs and are available on all *Resources* at the `/api/:resource/query/` list endpoint for a given resource type.

To perform a set query you must perform a `GET` request to the query endpoint providing the set query string as a value to the `query` argument.

For example:

Request:

```
GET /api/devices/query/?query=vendor=juniper
```

Response:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "attributes": {
            "owner": "jathan",
            "vendor": "juniper",
            "hw_type": "router",
            "metro": "lax"
        },
        "hostname": "lax-r2",
        "site_id": 1,
        "id": 2
    },
    {
        "attributes": {
            "owner": "jathan",
            "vendor": "juniper",
            "hw_type": "router",
            "metro": "iad"
        },
        "hostname": "iad-r1",
        "site_id": 1,
        "id": 5
    }
]
```

The optional `unique` argument can also be provided in order to ensure only a single object is returned, otherwise an error is returned.

Request:

```
GET /api/devices/query/?query=metro=iad&unique=true
```

Response:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "attributes": {
            "owner": "jathan",
            "vendor": "juniper",
            "hw_type": "router",
            "metro": "iad"
        },
        "hostname": "iad-r1",
        "site_id": 1,
        "id": 5
    }
]
```

If multiple results match the query, when `unique` has been specified, an error will be returned.

Request:

```
GET /api/devices/query/?query=vendor=juniper
```

Response:

```
HTTP 400 Bad Request
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "error": {
        "message": {
            "query": "Query returned 2 results, but exactly 1 expected"
        },
        "code": 400
    }
}
```

Python API

Database Primitives

Models

class nsot.models.**Assignment**(**args*, ***kwargs*)

DB object for assignment of addresses to interfaces (on devices).

This is used to enforce constraints at the relationship level for addition of new address assignments.

clean_address(*value*)

Enforce that new addresses can only be host addresses.

class nsot.models.**Attribute**(**args*, ***kwargs*)

Represents a flexible attribute for Resource objects.

clean_constraints(*value*)
Enforce formatting of constraints.

save(**args*, ***kwargs*)
Always enforce constraints.

class nsot.models.**Change**(**args*, ***kwargs*)
Record of all changes in NSoT.

clean_fields(*exclude=None*)
This will populate the change fields from the incoming object.

class nsot.models.**Circuit**(**args*, ***kwargs*)
Represents two network Interfaces that are connected

addresses
Return addresses associated with this circuit.

clean_site(*value*)
Always enforce that site is set.

devices
Return devices associated with this circuit.

interfaces
Return interfaces associated with this circuit.

class nsot.models.**Device**(**args*, ***kwargs*)
Represents a network device.

circuits
All circuits related to this Device.

class nsot.models.**Interface**(**args*, ***kwargs*)
A network interface.

assign_address(*cidr*)
Assign an address to this interface.

Must have prefix of /32 (IPv4) or /128 (IPv6).

Parameters **cidr** – IPv4/v6 CIDR host address or Network object

circuit
Return the Circuit I am associated with

clean_addresses()
Make sure that addresses/networks are saved as JSON.

clean_device_hostname(*device*)
Extract hostname from device

clean_mac_address(*value*)
Enforce valid mac_address.

clean_name(*value*)
Enforce name.

clean_site(*value*)
Always enforce that site is set.

clean_speed(*value*)
Enforce valid speed.

clean_type(*value*)
Enforce valid type.

get_addresses()
Return a list of assigned addresses.

get_assignments()
Return a list of informatoin about my assigned addresses.

get_mac_address()
Return a serializable representation of mac_address.

get_networks()
Return a list of attached Networks.

networks
Return all the parent Networks for my addresses.

set_addresses(*addresses*, *overwrite=False*, *partial=False*)
Explicitly assign a list of addresses to this Interface.

Parameters

- **addresses** – A list of CIDRs
- **overwrite** – Whether to purge existing assignments before assigning.
- **partial** – Whether this is a partial update.

class nsot.models.**Network**(**args*, ***kwargs*)
Represents a subnet or IP address.

clean_fields(*exclude=None*)
This will enforce correct values on fields.

clean_state(*value*)
Enforce that state is one of the valid states.

get_ancestors(*ascending=False*)
Return my ancestors.

get_children()
Return my immediate children.

get_descendants()
Return all of my children!

get_next_address(*num=None*, *strict=False*, *as_objects=True*)
Return a list of the next available addresses.

If no addresses are available, an empty list will be returned.

Parameters

- **num** – The number of addresses desired
- **as_objects** – Whether to return IPNetwork objects or strings

get_next_network(*prefix_length*, *num=None*, *strict=False*, *as_objects=True*)
Return a list of the next available networks.

If no networks are available, an empty list will be returned.

Parameters

- **prefix_length** – The prefix length of networks

- **num** – The number of networks desired
- **as_objects** – Whether to return IPNetwork objects or strings
- **strict** – Whether to return networks for strict allocation

Returns list(IPNetwork)

get_root()
Returns the root node (the parent of all of my ancestors).

get_siblings(*include_self=False*)
Return my siblings. Root nodes are siblings to other root nodes.

is_child_node()
Returns whether I am a child node.

is_leaf_node()
Returns whether I am leaf node (no children).

is_root_node()
Returns whether I am a root node (no parent).

reparent_subnets()
Determine list of child nodes and set the parent to self.

save(**args*, ***kwargs*)
This is stuff we want to happen upon save.

class nsot.models.**NetworkManager**
Manager for NetworkInterface objects.

get_by_address(*cidr*, *site=None*)
Lookup a Network object by `cidr`.

Parameters

- **cidr** – IPv4/IPv6 CIDR string
- **site** – `Site` instance or `site_id`

get_closest_parent(*cidr*, *prefix_length=0*, *site=None*)
Return the closest matching parent Network for a `cidr` even if it doesn't exist in the database.

Parameters

- **cidr** – IPv4/IPv6 CIDR string
- **prefix_length** – Maximum prefix length depth for closest parent lookup
- **site** – `Site` instance or `site_id`

class nsot.models.**Resource**(**args*, ***kwargs*)
Base for heirarchial Resource objects that may have attributes.

clean_attributes()
Make sure that attributes are saved as JSON.

get_attributes()
Return the JSON-encoded attributes as a dict.

set_attributes(*attributes*, *valid_attributes=None*, *partial=False*)
Validate and store the attributes dict as a JSON-encoded string.

class nsot.models.**ResourceManager**
Manager for Resource objects that adds a special resource methods:

- `.set_query()` - For performing set theory lookups by attribute-value string patterns
- `.by_attribute()` - For looking up objects by attribute name/value.

by_attribute(*name*, *value*, *site_id=None*)
Filter objects by Attribute `name` and `value`.

For example:

```
>>> Interface.objects.by_attribute(name='vlan', value=300)
[<Interface: device=1, name=eth0>]
```

Parameters

- **name** – Attribute name
- **value** – Attribute value
- **site_id** – ID of Site to filter results

queryset_class
alias of *ResourceSetTheoryQuerySet*

set_query(*query*, *site_id=None*, *unique=False*)
Filter objects by set theory attribute-value string patterns.

For example:

```
>>> Network.objects.set_query('owner=jathan +cluster=sjc'}
[<Device: foo-bar1>, <Device: foo-bar3>, <Device: foo-bar4>]

>>> Network.objects.set_query('owner=gary -cluster=sjc'}
[<Device: foo-bar2>]

>>> Network.objects.set_query('owner=jathan foo=baz', unique=True}
[<Device: foo-bar3>]
```

Parameters

- **query** – Set theory query pattern
- **site_id** – ID of Site to filter results
- **unique** – Find exactly one match, error otherwise

class nsot.models.**ResourceSetTheoryQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)
Set theory QuerySet for Resource objects to add `.set_query()` method.

For example:

```
>>> qs = Network.objects.filter(network_address=u'10.0.0.0')
>>> qs.set_query('owner=jathan +metro=lax')
```

You may also search using regex by appending `_regex` to an attribtue name and providing a regex pattern as the value:

```
>>> qs = Device.objects.set_query('role_regex=[bd]r')
```

Which is functionally equivalent to:

```
>>> qs = Device.objects.set_query('role=br +role=dr')
```

by_attribute(*name*, *value*, *site_id=None*)
Lookup objects by Attribute `name` and `value`.

set_query(*query*, *site_id=None*, *unique=False*)
Filter objects by set theory attribute-value `query` patterns.

class nsot.models.**Site**(**args*, ***kwargs*)
A namespace for attribtues, devices, and networks.

class nsot.models.**User**(**args*, ***kwargs*)
A custom user object that utilizes email as the username.

generate_auth_token()
Serialize user data and encrypt token.

classmethod verify_auth_token(*email*, *auth_token*, *expiration=None*)
Verify token and return a User object.

verify_secret_key(*secret_key*)
Validate secret_key

class nsot.models.**Value**(**args*, ***kwargs*)
Represents a value for an attribute attached to a Resource.

clean_site(*value*)
Always enforce that site is set.

nsot.models.**change_api_updated_at**(*sender=None*, *instance=None*, **args*, ***kwargs*)
Anytime the API is updated, invalidate the cache.

nsot.models.**delete_resource_values**(*sender*, *instance*, ***kwargs*)
Delete values when a Resource object is deleted.

nsot.models.**model_class**
alias of *Circuit*

nsot.models.**update_device_interfaces**(*sender*, *instance*, ***kwargs*)
Anytime a device is saved, update device_hostname on its interfaces

Fields

class nsot.fields.**BinaryIPAddressField**(**args*, ***kwargs*)
IP Address field that stores values as varbinary.

get_db_prep_value(*value*, *connection*, *prepared=False*)
Python -> DB.

to_python(*value*)
DB -> Python.

class nsot.fields.**ChainedForeignKey**(*to*, *chained_field=None*, *chained_model_field=None*, *show_all=False*, *auto_choose=False*, *sort=True*, *view_name=None*, ***kwargs*)
chains the choices of a previous combo box with this one

class nsot.fields.**JSONField**(**args*, ***kwargs*)
JSONField is a generic textfield that neatly serializes/unserializes JSON objects seamlessly. Main thingy must be a dict object.

get_db_prep_save(*value*, *connection*, ***kwargs*)
Convert our JSON object to a string before we save

south_field_triple()
Returns a suitable description of this field for South.

to_python(*value*)
Convert our string value to JSON after we load it from the DB

class nsot.fields.**MACAddressField**(**args*, ***kwargs*)
Subclass of base field to raise a DRF ValidationError.

DRF handles Django's default ValidationError, but this is so that we can always expect the DRF version, for better consistency in debugging and testing.

Validators

Validators for validating object fields.

nsot.validators.**validate_cidr**(*value*)
Validate whether `value` is a validr IPv4/IPv6 CIDR.

nsot.validators.**validate_email**(*value*)
Validate whether `value` is an email address.

nsot.validators.**validate_host_address**(*value*)
Validate whether `value` is a host IP address.

nsot.validators.**validate_mac_address**(*value*)
Validate whether `value` is a valid MAC address.

nsot.validators.**validate_name**(*value*)
Validate whether `value` is a valid name.

Exceptions

exception nsot.exc.**Error**(*detail=None*, *code=None*)
Baseclass for NSoT Exceptions.

exception nsot.exc.**ModelError**(*detail=None*, *code=None*)
Base class for NSoT Model Exceptions.

exception nsot.exc.**BaseHttpError**(*detail=None*, *code=None*)
Base HTTP error.

exception nsot.exc.**BadRequest**(*detail=None*, *code=None*)
HTTP 400 error.

exception nsot.exc.**Unauthorized**(*detail=None*, *code=None*)
HTTP 401 error.

exception nsot.exc.**Forbidden**(*detail=None*, *code=None*)
HTTP 403 error.

exception nsot.exc.**NotFound**(*detail=None*, *code=None*)
HTTP 404 error.

exception nsot.exc.**Conflict**(*detail=None*, *code=None*)
HTTP 409 error.

nsot.exc.**DjangoValidationError**
alias of `ValidationError`

exception nsot.exc.**ObjectDoesNotExist**
The requested object does not exist

exception nsot.exc.**MultipleObjectsReturned**
The query returned multiple objects when only one was expected.

REST API Primitives

Views

class nsot.api.views.**AttributeViewSet**(***kwargs*)
API endpoint that allows Attributes to be viewed or edited.

serializer_class
alias of AttributeSerializer

class nsot.api.views.**BaseNsotViewSet**(***kwargs*)
Default viewset for Nsot objects with the following defaults:

•Objects are designed to be nested under site resources, but can also be top-level resources.

get_natural_key_kwargs(*filter_value*)
This method should take value and return a dictionary containing the natural key fields used to filter results.

This is called internally by self.get_object() if a subclass has defined a natural_key.

Parameters **filter_value** – Value to be used to filter by natural_key

get_object()
Enhanced default to support looking up objects for:

•Natural key lookups for resource objects (e.g. Device.hostname)

•Inject of site into filter lookup if site_pk is set.

Currently this does NOT filter the queryset, which should not be a problem as we were never using .get_object() before. See the FIXME comments for more context.

list(*request*, *site_pk=None*, *queryset=None*, **args*, ***kwargs*)
List objects optionally filtered by site.

natural_key = None
Natural key for the resource. If not defined, defaults to pk-only.

not_found(*pk=None*, *site_pk=None*, *msg=None*)
Standard formatting for 404 errors.

retrieve(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Retrieve a single object optionally filtered by site.

success(*data*, *status=None*, *headers=None*)
Return a positive API response.

Parameters

- **data** – Serialized data
- **status** – (Optional) HTTP status code
- **headers** – (Optional) Dict of extra headers

class nsot.api.views.**ChangeViewSet**(***kwargs*)
Read-only API endpoint that allows Changes to be viewed.

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.

serializer_class
alias of ChangeSerializer

class nsot.api.views.**CircuitViewSet**(***kwargs*)
API endpoint that allows Circuits to be viewed or edited.

addresses(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of addresses for the interfaces on this Circuit.

bulk_update(*request*, **args*, ***kwargs*)
Workaround for bulk update of objects with unique constraint.

At this time this is only required by the Circuit object, which is why it is only defined here.

Credit: https://github.com/miki725/django-rest-framework-bulk/issues/30 Source: http://bit.ly/2ilboIQ

devices(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of devices for this Circuit.

filter_class
alias of CircuitFilter

get_natural_key_kwargs(*filter_value*)
Return a dict of kwargs for natural_key lookup.

interfaces(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of interfaces for this Circuit.

list(*request*, **args*, ***kwargs*)
Override default list so we can cache results.

retrieve(*request*, **args*, ***kwargs*)
Override default retrieve so we can cache results.

serializer_class
alias of CircuitSerializer

class nsot.api.views.**DeviceViewSet**(***kwargs*)
API endpoint that allows Devices to be viewed or edited.

circuits(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of Circuits for this Device

filter_class
alias of DeviceFilter

get_natural_key_kwargs(*filter_value*)
Return a dict of kwargs for natural_key lookup.

interfaces(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return all interfaces for this Device.

serializer_class
alias of DeviceSerializer

class nsot.api.views.**InterfaceViewSet**(***kwargs*)
API endpoint that allows Interfaces to be viewed or edited.

addresses(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of addresses for this Interface.

assignments(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return a list of information about my assigned addresses.

circuit(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the Circuit I am associated with

filter_class
alias of `InterfaceFilter`

get_natural_key_kwargs(*filter_value*)
Return a dict of kwargs for natural_key lookup.

list(*request*, **args*, ***kwargs*)
Override default list so we can cache results.

networks(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return all the containing Networks for my assigned addresses.

retrieve(*request*, **args*, ***kwargs*)
Override default retrieve so we can cache results.

serializer_class
alias of `InterfaceSerializer`

class `nsot.api.views.`**NetworkViewSet**(***kwargs*)
API endpoint that allows Networks to be viewed or edited.

ancestors(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return ancestors of this Network.

assignments(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the interface assignments for this Network.

children(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the immediate children of this Network.

closest_parent(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the closest matching parent of this Network even if it doesn't exist in the database.

descendants(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return descendants of this Network.

descendents(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return descendants of this Network.

Deprecated since version 1.1.

This endpoint is pending deprecation. Use the `descendants` endpoint instead.

filter_class
alias of `NetworkFilter`

get_natural_key_kwargs(*filter_value*)
Return a dict of kwargs for natural_key lookup.

next_address(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return next available IPs from this Network.

next_network(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return next available networks from this Network.

parent (*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the parent of this Network.

query (*request*, *site_pk=None*, **args*, ***kwargs*)
Override base query to inherit filtering by query params.

reserved (*request*, *site_pk=None*, **args*, ***kwargs*)
Display all reserved Networks.

root (*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return the parent of all ancestors for this Network.

serializer_class
alias of `NetworkSerializer`

siblings (*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return Networks with the same parent. Root nodes are siblings to other root nodes.

subnets (*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return subnets of this Network.

supernets (*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Return supernets of this Network.

class `nsot.api.views.`**NotFoundViewSet** (***kwargs*)
Catchall for bad API endpoints.

class `nsot.api.views.`**NsotBulkUpdateModelMixin**
The default mixin isn't using super() so multiple-inheritance breaks. This fixes it for our use-case.

class `nsot.api.views.`**NsotViewSet** (***kwargs*)
Generic mutable viewset that logs all change events and includes support for bulk creation of objects.

get_success_headers (*data*)
Overload default to include relative request PATH.

Parameters **data** – Dict of validated serializer data

perform_create (*serializer*)
Support bulk create.

Parameters **serializer** – Serializer instance

perform_destroy (*instance*)
Overload default to handle non-serializer exceptions, and log Change events.

Parameters **instance** – Model instance to delete

perform_update (*serializer*)
Overload default to handle non-serializer exceptions, and log Change events.

Parameters **serializer** – Serializer instance

class `nsot.api.views.`**ResourceViewSet** (***kwargs*)
Resource views that include set query list endpoints.

get_resource_object (*pk*, *site_pk*)
Return a resource object based on pk or site_pk.

query (*request*, *site_pk=None*, **args*, ***kwargs*)
Perform a set query.

class `nsot.api.views.`**SiteViewSet** (***kwargs*)
API endpoint that allows Sites to be viewed or edited.

class nsot.api.views.**UserPkInfo**(*user*, *pk*)
Namedtuple for retrieving pk and user object of current user.

pk
Alias for field number 1

user
Alias for field number 0

class nsot.api.views.**UserViewSet**(***kwargs*)
This viewset automatically provides *list* and *detail* actins.

retrieve(*request*, *pk=None*, *site_pk=None*, **args*, ***kwargs*)
Retreive a single user.

serializer_class
alias of UserSerializer

class nsot.api.views.**ValueViewSet**(***kwargs*)
API endpoint that allows Attribute Values to be viewed or edited.

serializer_class
alias of ValueSerializer

Serializers

class nsot.api.serializers.**AttributeCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for POST on Attributes.

class nsot.api.serializers.**AttributeSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for GET, DELETE on Attributes.

class nsot.api.serializers.**AttributeUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for PUT, PATCH, on Attributes.

Currently because Attributes have only one required field (name), and it may not be updated, there is not much functional difference between PUT and PATCH.

class nsot.api.serializers.**AuthTokenSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
AuthToken authentication serializer to validate username/secret_key inputs.

class nsot.api.serializers.**ChangeSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for displaying Change events.

class nsot.api.serializers.**CircuitCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for POST on Circuits.

class nsot.api.serializers.**CircuitPartialUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for PATCH on Circuits.

class nsot.api.serializers.**CircuitSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for GET, DELETE on Circuits

class nsot.api.serializers.**CircuitUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PUT on Circuits.

class nsot.api.serializers.**DeviceCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for POST on Devices.

class nsot.api.serializers.**DevicePartialUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PATCH on Devices.

class nsot.api.serializers.**DeviceSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for GET, DELETE on Devices.

class nsot.api.serializers.**DeviceUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PUT on Devices.

class nsot.api.serializers.**InterfaceCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for POST on Interfaces.

class nsot.api.serializers.**InterfacePartialUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PATCH on Interfaces.

class nsot.api.serializers.**InterfaceSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for GET, DELETE on Interfaces.

class nsot.api.serializers.**InterfaceUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PUT on Interfaces.

class nsot.api.serializers.**JSONDataField**(*read_only=False*, *write_only=False*, *required=None*, *default=<class rest_framework.fields.empty>*, *initial=<class rest_framework.fields.empty>*, *source=None*, *label=None*, *help_text=None*, *style=None*, *error_messages=None*, *validators=None*, *allow_null=False*)

Base field used to represent attributes as JSON <-> `field_type`.

It is an error if `field_type` is not defined in a subclass.

class nsot.api.serializers.**JSONDictField**(*read_only=False*, *write_only=False*, *required=None*, *default=<class rest_framework.fields.empty>*, *initial=<class rest_framework.fields.empty>*, *source=None*, *label=None*, *help_text=None*, *style=None*, *error_messages=None*, *validators=None*, *allow_null=False*)

Field used to represent attributes as JSON <-> Dict.

field_type
alias of `dict`

class nsot.api.serializers.**JSONListField**(*read_only=False*, *write_only=False*, *required=None*, *default=<class rest_framework.fields.empty>*, *initial=<class rest_framework.fields.empty>*, *source=None*, *label=None*, *help_text=None*, *style=None*, *error_messages=None*, *validators=None*, *allow_null=False*)

Field used to represent attributes as JSON <-> List.

field_type
alias of `list`

class nsot.api.serializers.**MACAddressField**(*read_only=False*, *write_only=False*, *required=None*, *default=<class rest_framework.fields.empty>*, *initial=<class rest_framework.fields.empty>*, *source=None*, *label=None*, *help_text=None*, *style=None*, *error_messages=None*, *validators=None*, *allow_null=False*)

Field used to validate MAC address objects as integer or string.

class nsot.api.serializers.**NetworkCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for POST on Networks.

class nsot.api.serializers.**NetworkPartialUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PATCH on Networks.

class nsot.api.serializers.**NetworkSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for GET, DELETE on Networks.

class nsot.api.serializers.**NetworkUpdateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Used for PUT on Networks.

class nsot.api.serializers.**NsotSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Base serializer that logs change events.

to_internal_value(*data*)
Inject site_pk from view's kwargs if it's not already in data.

to_representation(*obj*)
Always return the dict representation.

class nsot.api.serializers.**ResourceSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
For any object that can have attributes.

create(*validated_data*, *commit=True*)
Create that is aware of attributes.

update(*instance*, *validated_data*, *commit=True*)
Update that is aware of attributes.

This will not set attributes if they are not provided during a partial update.

class nsot.api.serializers.**UserSerializer**(**args*, ***kwargs*)
UserProxy model serializer that takes optional *with_secret_key* argument that controls whether the secret_key for the user should be displayed.

class nsot.api.serializers.**ValueCreateSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for POST on Values.

class nsot.api.serializers.**ValueSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Used for GET, DELETE on Values.

Filters

class nsot.api.filters.**CircuitFilter**(**args*, ***kwargs*)
Filter for Circuit objects.

class nsot.api.filters.**DeviceFilter**(**args*, ***kwargs*)
Filter for Device objects.

class nsot.api.filters.**InterfaceFilter**(**args*, ***kwargs*)
Filter for Interface objects.

Includes a custom override for filtering on mac_address because this is not a Django built-in field.

filter_mac_address(*queryset*, *value*)
Overloads queryset filtering to use built-in.

Doesn't work by default because MACAddressField is not a Django built-in field type.

class nsot.api.filters.**NetworkFilter**(**args*, ***kwargs*)
Filter for Network objects.

filter_cidr(*queryset*, *value*)
Converts `cidr` to network/prefix filter.

filter_include_ips(*queryset*, *value*)
Converts `include_ips` to queryset filters.

filter_include_networks(*queryset*, *value*)
Converts `include_networks` to queryset filters.

filter_root_only(*queryset*, *value*)
Converts `root_only` to null parent filter.

class nsot.api.filters.**ResourceFilter**(**args*, ***kwargs*)
Attribute-aware filtering for Resource objects.

filter_attributes(*queryset*, *value*)
Reads 'attributes' from query params and joins them together as an intersection set query.

Pagination

Routers

class nsot.api.routers.**BulkRouter**(**args*, ***kwargs*)
Map http methods to actions defined on the bulk mixins.

class nsot.api.routers.**BulkNestedRouter**(**args*, ***kwargs*)
Bulk-enabled nested router.

URLs

Authentication

Utilities

nsot.util.**qpbool**(*arg*)
Convert "truthy" strings into Booleans.

```
>>> qpbool('true')
True
```

Parameters **arg** – Truthy string

nsot.util.**normalize_auth_header**(*header*)
Normalize a header name into WSGI-compatible format.

```
>>> normalize_auth_header('X-NSoT-Email')
'HTTP_X_NSOT_EMAIL'
```

Parameters **header** – Header name

nsot.util.**generate_secret_key**()
Return a secret key suitable for use w/ Fernet.

```
>>> generate_secret_key()
'1BpuqeM5d5pi-U2vIsqeQ8YnTrXRRUAfqV-hu6eQ5Gw='
```

nsot.util.**get_field_attr**(*model*, *field_name*, *attr*)
Return the specified field for a model field

class nsot.util.**SetQuery**(*action*, *name*, *value*)

action
Alias for field number 0

name
Alias for field number 1

value
Alias for field number 2

nsot.util.**parse_set_query**(*query*)

Parse a representation of set operations for attribute/value pairs into (action, name, value) and return a list of `SetQuery` objects.

Computes left-to-right evaluation, where the first character indicates the set operation:

- "+" indicates a union
- "-" indicates a difference
- no marker indicates an intersection

For example:

```
>>> parse_set_query('+owner=team-networking')
[SetQuery(action='union', name='owner', value='team-networking')]
>>> parse_set_query('foo=bar')
[SetQuery(action='intersection', name='foo', value='bar')]
>>> parse_set_query('foo=bar -owner=team-networking')
[SetQuery(action='intersection', name='foo', value='bar'),
 SetQuery(action='difference', name='owner', value='team-networking')]
```

Parameters **query** – Set query string

nsot.util.**generate_settings**(*config_template=None*)

Used to emit a generated configuration from `config_template`.

Parameters **config_template** – Config template

nsot.util.**initialize_app**(*config*)

Actions to be performed prior to creating the Application object.

Parameters **config** – Config object

nsot.util.**main**()

CLI application used to manage NSoT.

nsot.util.**cidr_to_dict**(*cidr*)

Take a cidr and return it as a dictionary.

```
>>> cidr_to_dict('192.168.0.0/16')
{'network_address': '192.168.0.0', 'prefix_length': 16}
```

Parameters **cidr** – IPv4/IPv6 CIDR string

nsot.util.**slugify**(*s*)

Slugify a string.

This works in a less-agressive manner than Django's slugify, which simply drops most drops most non-alphanumeric characters and lowercases the entire string. It would likely to cause uniqueness conflicts for things like interface names, such as Eth1/2/3 and Eth12/3, which would slugify to be the same.

```
>>> slugify('switch-foo01:Ethernet1/2')
'switch-foo01:Ethernet1_2'
```

Parameters **s** – String to slugify

CHAPTER 8

Development

Git Branches

On the parent repo for NSoT, there are two important branches:

- `develop` is the branch that all Pull Requests are opened against and represents the bleeding edge of work being done on NSoT.
- `master` is considered to be the 'stable' branch and no PRs should be merged directly in to this branch. As features are merged into `develop`, releases are created off of that branch and then merged into `master`. See *Release Process* for more information on how this is done.

When developing on NSoT, you should be basing your work on the `develop` branch.

Setting up Your Environment

Note: You'll need to have a reasonably recent version of npm to build front-end dependencies. (Minimum version tested is `1.3.24`)

We suggest setting up your test environment in a Python virtualenv:

```
$ virtualenv nsot
$ source nsot/bin/activate
```

Or, if you use virtualenvwrapper:

```
$ mkvirtualenv nsot
```

If you haven't already, make sure you set up git and add an SSH key to your GitHub account before proceeding!

After that, clone the repository into whichever directory you use for development and install the dependencies:

```
$ git clone git@github.com:dropbox/nsot.git
$ cd nsot
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

Running a Test Instance

For developement and testing, it's easiest to run NSoT behind a reverse proxy that handles authentication and sends a username via a *special HTTP header*. We've included a test proxy for running on development instances.

To get started, follow these steps:

```
# Initialize the config
$ nsot-server init

# Setup the database.
$ nsot-server upgrade

# Run the development reverse proxy (where $USER is the desired username)
$ nsot-server user_proxy $USER

# (In another terminal) Run the front-end server, remember to activate your
# virtualenv first if you need to
$ nsot-server start
```

Note: This quick start assumes that you're installing and running NSoT on your local system (aka *localhost*).

Now, point your web browser to http://localhost:8991 and explore the documentation!

Running Unit Tests

All tests will automatically be run on Travis CI when pull requests are sent. However, it's beneficial to run the tests often during development:

```
$ py.test -v tests/
```

Working with Database Migrations

If you make any changes to the database models you'll need to generate a new migration. We use Django's built-in support for database migrations underneath, so for general schema changes is should be sufficient to just run:

```
$ nsot-server makemigrations
```

This will generate a new schema version. You can then sync to the latest version:

```
$ nsot-server migrate
```

Working with Docs

Documentation is generated using Sphinx. If you just want to build and view the docs | you cd into the `docs` directory and run `make html`. Then point your browser | to `docs/\_build/html/index.html` on your local filesystem.

If you're actively modifying the docs it's useful to run the autobuild server:

```
$ sphinx-autobuild docs docs/_build/html/
```

This will start a server listening on a port that you can browse to and will be automatically reloaded when you change any rst files. One downside of this approach is that is doesn't refresh when docstrings are modified.

Front-end Development

We use a combination JavaScript utilities to do front-end development:

- npm - npm is used to manage our build dependencies
- bower - bower to manage our web dependencies
- gulp - gulp for building, linting, testing

Note: You do not have to install these yourself! When you run `setup.py develop`, it will install and build all front-end components for you!

Adding New Build Dependencies

For the most part you shouldn't need to care about these details though if you want to add new build dependencies, for example gulp-concat, you would run the followiing:

```
# Install gulp-concat, updating package.json with a new devDependency
$ npm install gulp-concat --save-dev

# Writes out npm-shrinkwrap.json, including dev dependencies, so consistent
# build tools are used
$ npm shrinkwrap --dev
```

Adding New Web Dependencies

Adding new web dependencies are done through bower:

```
# Install lodaash, updating bower.json with the new dependency
$ bower install lodash --save
```

Unfortunately, bower doesn't have a shrinkwrap/freeze feature so you'll want to update the version string to make the version explicit for repeatable builds.

We make use of bower's "main file" concept to distribute only "main" files. Most packages don't consider consider the minified versions of their project to be their main files so you'll likely also need to update the `overrides` section of `bower.json` with which files to distribute.

Versioning

We use semantic versioning. Version numbers will follow this format:

```
{Major version}.{Minor version}.{Revision number}.{Build number (optional)}
```

Patch version numbers (0.0.x) are used for changes that are API compatible. You should be able to upgrade between minor point releases without any other code changes.

Minor version numbers (0.x.0) may include API changes, in line with the *Deprecation policy*. You should read the release notes carefully before upgrading between minor point releases.

Major version numbers (x.0.0) are reserved for substantial project milestones.

Release Process

When a new version is to be cut from the commits made on the `develop` branch, the following process should be followed. This is meant to be done by project maintainers, who have push access to the parent repository.

1. Create a branch off of the `develop` branch called `release-vX.Y.Z` where `vX.Y.Z` is the version you are releasing
2. Use `bump.sh` to update the version in `nsot/version.py` and the Dockerfile. Example:

```
$ ./bump.sh -v X.Y.Z
```

3. Update `CHANGELOG.rst` with what has changed since the last version. A one-line summary for each change is sufficient, and often the summary from each PR merge works.
4. Commit these changes to your branch.
5. Merge the release branch into `develop` and push that branch up
6. Merge the release branch into `master`
7. Create a new git tag with this verison in the format of `vX.Y.Z`
8. Push the `master` branch up along with the new tag
9. Create a new package and push it up to PyPI:

```
$ python setup.py sdist upload
```

Deprecation policy

NSoT releases follow a formal deprecation policy, which is in line with Django's deprecation policy.

The timeline for deprecation of a feature present in version 1.0 would work as follows:

- Version 1.1 would remain **fully backwards compatible** with 1.0, but would raise Python `PendingDeprecationWarning` warnings if you use the feature that are due to be deprecated. These warnings are **silent by default**, but can be explicitly enabled when you're ready to start migrating any required changes.

  Additionally, a `WARN` message will be logged to standard out from the `nsot-server` process.

  Finally, a `Warning` header will be sent back in any response from the API. For example:

```
Warning: 299 - "The `descendents` API endpoint is pending deprecation. Use
the `descendants` API endpoint instead."
```

- Version 1.2 would escalate the Python warnings to `DeprecationWarning`, which is **loud by default**.
- Version 1.3 would remove the deprecated bits of API entirely and accessing any deprecated API endoints will result in a `404` error.

Note that in line with Django's policy, any parts of the framework not mentioned in the documentation should generally be considered private API, and may be subject to change.

CHAPTER 9

Changelog

Version History

1.1.3 (2017-02-03)

- Bump `django-smart-selects` to version 1.3.x * Fixes a regression that was introduced in version 1.2.9 when an XSS bug

  was fixed

- Change `requirements.txt` to use Compatible Release version specifiers and track patch-level updates on all of our dependencies

1.1.2 (2017-01-25)

- Reimplements the `Network.get_next_network` method
- `Network.get_next_network` now optionally returns networks that can be strictly allocated by passing the `strict_allocation=True` paremeter.
- Fixed a minor bug in `/api/:resource/:id/query/` API endpoints where `unique` would always evaluate to `True` if present in the query parameters.

1.1.1 (2017-01-27)

- Add *name_slug* field to Circuit, make it the natural key to fix a bug with Circuit names that contain slashes

1.1 (2017-01-23)

- A formal *Deprecation policy* has been implemented which dictates a three-feature release cycle for removing deprecated API endpoints. Please see the documentation on this topic for more details.
- Fix #203 - Implementation of Circuits as a resource object.
  - A Circuit has one-to-one relationship with each of A and Z side endpoint Interfaces.
  - Circuits are resource objects and therefore may have attributes and support set query lookups.
  - A circuit must have at least an A-side endpoint defined. For circuits for which you do not own the remote end, you may leave the Z-side empty and specify the remote endpoint by customizing the circuit name.

- Circuits have the following detail routes available in the API:
  - `circuits/:id/devices/` - List peer devices on either end of circuit
  - `circuits/:id/interfaces/` - List interfaces bound to the circuit
  - `circuits/:id/addresses/` - List addresses bound to circuit interfaces
- Interfaces have a new `interfaces/:id/circuit/` detail route that will display the circuit to which an interface is bound.
- Devices have a new `devices/:id/circuits/` detail route that will display all circuits bound to interfaces on the device.
- Fix #191 - The Interface object unicode representation changed to `device_hostname:name` so that it can more easily be used as a slug for computing Circuit slug.
- Fix #230 - The misspelled `networks/:id/descendents/` API endpoint is pending deprecation in exchange for `networks/:id/descendants/`.

1.0.13 (2017-01-12)

- Provides the ability to more efficiently access the device hostname associated with an interface, using the cached `device_hostname` field.
- Provides the ability access interface objects by natural key of `device_hostname:name`. Ex. `foo-bar1:eth1`

1.0.12 (2017-01-12)

- **Fix #252 - Fixes bug in `get_next_network` for assigned networks of** different prefix lengths

1.0.11 (2017-01-10)

- Fix #250 - Improves treatment of `get_next_network` and assigned state
- Fix #238 - Update to Docker instructions
- Fix #219 - Better handling of attempts to create objects in non-existent sites
- Moved Vagrantfile to root of repo

1.0.10 (2016-12-05)

- Fix for handling IPAddress defaults in migrations, to avoid attempting validation of a NULL default.

1.0.9 (2016-11-23)

- Added missing database migrations related to having changed or added the `verbose_name` on a bunch of model fields prior to 1.0 release. No schema changes are actually made in the migration. This is being released so that some pending pull requests can be merged in more cleanly.

1.0.8 (2016-10-24)

- Provides the ability to require uniqueness for results of queries using the optional `unique=true` param. Queries with multiple results that have this flag set will return an error. Implements #221.

1.0.7 (2016-10-24)

- Implemented changes needed to upgrade to Django REST Framework v3.5.0
- Added *fields = '__all__'* to all default model serializers used for displaying objects
- Changes required for django-filter>=0.15 were made for filtersets using custom fields.

1.0.6 (2016-10-18)

- Improve performance in Network.get_next_network() for large prefixes
  - The fix in #224 introduced a notable performance bug due to iterating all descendents vs. only direct children.
  - This patch addresses the performance issue by attempting to pre-seed the list of dirty networks via excluding ones with ineligible prefix lengths as well as immediately checking whether a candidate subnet is dirty BEFORE iterating child networks vs. AFTER.

1.0.5 (2016-10-13)

- Fix #224 - Fixed a bug in `Network.next_network()` where a nested child (descendent) would continually be offered as free, even if it existed in the database. All descendent networks for a parent are now inspected when determining availability.

1.0.4 (2016-09-29)

- Replaced `settings.NETWORK_INTERCONNECT_PREFIXLEN` (an integer) with `settings.NETWORK_INTERCONNECT_PREFIXES` (a tuple) to support IPv6 prefixes, which defaults to prefixes (/31, /127).
- `Network.next_address()` was changed to calculate available addresses differently if the network from which you are allocating is determined to be an interconnect network. For interconnects, gateway and broadcast addresses can be returned. For any other networks, they cannot.

1.0.3 (2016-09-08)

- Fix #216 - Fixed a bug in `Network.next_network()` where networks containing children were being offered as free. Networks are now only offered if they do not have any child networks.
- Fix #212 - Updated requirements to require `djangorestframework>=3.4.4` and removed `nsot.api.serializers.LimitedForeignKeyField` since this functionality is now built into DRF.

1.0.2 (2016-08-31)

- Ubuntu 16.04 is now officially supported.
- Fix #213 - Updated requirements to utilize `cryptography==1.5` so that install will work on Ubuntu versions 12.04 through 16.04. (Credit: @slinderud)
- Finally fixed `bump.sh` to work on both Linux and Darwin. For real this time.

1.0.1 (2016-07-08)

- Fix #209 - Fixed a bug in `Network.closest_parent()` that would sometimes cause an incorrect parent network to be returned when performing a "closest parent" lookup for a CIDR.

1.0 (2016-04-27)

- OFFICIAL VERSION 1.0!!
- Completely documented all object fields including help_text, verbose_names, labels, default values, etc. for every field so that is cascades to serializers and form fields.

0.17.4 (2016-04-22)

- Fixed a bug in `Network.next_address()` and `Network.next_network()` where children w/ busy states were mistakenly being excluded from the filter and therefore causing them to be offered as free. This also addressed a related bug where networks were not offered unless they came after the last prefix of the last matching child.

0.17.3 (2016-04-21)

- Added documentation for set queries for both how they work and for how to use them.
- Fixed a typo in Docker readme
- Added an entry-point for `snot-server` because reasons

0.17.2 (2016-04-17)

- Filtering of Interfaces by `mac_address` can now be done using either the string (e.g. `'00:00:00:00:00:01'`) or integer (e.g. `1`) representations.

0.17.1 (2016-04-07)

- Fixed a bug that would cause set queries lookups of attributes values containing spaces to always fail. When performing a set queries for an `attribute=value` pair, if a value contains a space, it must be quoted, and it will be properly parsed.
- When performing a set query for an attribute that does not exist, an error is raised.
- When performing a set query, if no attribute pairs are found, an empty set is returned.
- Docs: Fixed a typo in data model doc

- Docs: Fixed incorrect year for a bunch of entries in changelog

0.17 (2016-03-31)

- **BACKWARDS INCOMPATIBLE** - API version 1.0 is now the global default.
- Fix #167 - Web UI has been updated to use API v1.0
- Ripped out all pre-v1 code.
- Updated the browsable API renderer to not display "filter forms", so that browsable API views with tons of results and related fields don't deadlock.

0.16 (2016-03-29)

- Finally added a login screen to the web UI.
- Fixes #130 - Redirect to login screen if a 401 is detected
- This adds HTTP interceptor for 401 responses that will redirect to the DRF API login web screen.
- Also skinned the default DRF login screen to match the NSoT theme.
- Stopgap fix in `services.js` to check for `response.status`. This will have to be adjusted as a part of the API version 1.0 migration, along with all of the other JS code.

0.15.10 (2016-03-28)

- Fix #168 - Fix a 500 when assigning address that is in multiple sites

0.15.9 (2016-03-17)

- Bring a lot of documentation up to speed for readthedocs.org
- Added docstrings in places where there were none.
- Added code examples to some docstrings
- Updated requirements: Django==1.8.11

0.15.8 (2016-03-12)

- Fixes #171: Implemented API support for lookup by closest parent
- This implements a new detail route on the Networks endpoint at `networks/{cidr}/closest_parent/`. The Network need not exist in the database and if found, the closest matching parent network will be returned.
- The endpoint also accepts a `prefix_length` argument to optionally restrict how far it will recurse to find possible parents.

0.15.7 (2016-03-12)

- Migrated to built-in filtering of Interface objects in API.
- Also added the ability to filter by `device__hostname`, e.g. `GET /api/interfaces/?device__hostname=foo-bar1`

0.15.6 (2016-03-10)

- Fixes #169: Bugfix when filtering objects by 'attributes' in list view
- Fixed a bug that would result in a 500 crash when filtering by attributes in list view if multiple sites have matching objects.
- Fixes #166: Added a settings toggle to display IPv6 in compressed form. (See: `settings.NSOT_COMPRESS_IPV6`)

0.15.5 (2016-03-08)

- Bugfix to filtering networks in API and bump.sh and update requirements.
- Fixed shebang in `bump.sh` and used it to bump the version!
- Upgrade requirements: certifi==2016.2.28
- Bugfix in API filtering for Network objects that would result in an empty set if both `include_ips` and `include_networks` were set to `True`.
- Added unit tests to extercise `include_ips/include_networks` filters, because come on.

0.15.4 (2016-03-02)

- Made authentication API endpoints version-aware.
  - Overlooked the API authentication endpoints when doing the API versioning.
- Moved API version header to root of tests so that the "API version" message shows up on all executions of unit tests.
- Updated requirements django-rest-swagger==0.3.5.

0.15.3 (2016-02-29)

- Complete overhaul of API filtering to use DRF built-in filtering.
- All overloads in views of .get_queryset() has been removed and replaced with `filter_class` objects stored in `nsot.api.filters`
- All Resource filtering is now done using built-in `DjangoFilterBackend` objects using either `filter_class` or `filter_fields`.

0.15.2 (2016-02-24)

- Fixes #118 - Network objects are now round-trippable in API.
  - You may now provide either `cidr` or `network_address` + `prefix_length` when creating a Network object.
  - A Network object returned by the API may now be full used for create or update, making them round-trippable.
- Verbose names and help text have been added to all Network fields, so that they display all pretty like.

0.15.1 (2016-02-23)

- Added X-Forward-For into request logging.
- Also added an API test for sending X-Forward-For

0.15 (2016-02-22)

- Full support for PATCH in the API and some resultant bug fixes to PUT.
  - Specifically, this means any resource that is allowed to have attributes can now be partially updated using PATCH, because PATCH operations have been made attribute-aware.
  - Attributes themselves cannot YET be partially updated, but we hope to address that in a future... PATCH.
- Serializers
  - PATCH support enabled for complex objects: Attributes, Devices, Interfaces, Networks.
  - ResourceSerializer subclasses now all inherit default behavior for handling attributes. The `.create()` and `.update()` methods now take an optional `commit=` argument to toggle whether to save an object after updating attributes. This is so that descendent serializers subclasses can overload this method and not call save until they choose (such as in Interface serializers).
  - Each resource now has PUT and PATCH serializers broken out explicitly to facilitate the "optional fields" nature of PATCH vs. the "mandatory fields" nature of PUT.
- Attributes
  - All error messages raised when validating attributes include the word "attributes" so that you know it's a validation error specific to attributes.
- Bug Fixes
  - Bugfix in handling PUT requests where attributes would be initialized if not provided. Attributes are now mandatory on any PUT requests and will result in an error if they are missing.
  - Bugfix when assigning more than one IP address from the same network to an Interface that would result in a 500 error (and unit tests now catch this).

0.14.2 (2016-02-19)

- Bugfixes w/ natural_key lookups that would result in a 500 error.
  - Turns out that `site_pk` was incorrectly being dropped when doing natural_key lookups, which would result in a 500 w/ multiple sites.

  - We now detect when multiple objects are returned when looking up resources by natural_key and display a helpful 400 error.
  - Reverted top-level URL router back to Bulk default router because the SimpleRouter base doesn't provide api-root, and we kind of (really) want that.

0.14.1 (2016-02-19)

- Issue #50 - Adds better device name validation
- Fixed regex to match DNS hostname requirements. Added unit tests for device name checking
- Fix device name / attribute name comment

0.14 (2016-02-17)

- Implement GET/PUT objects by natural_key and minor fixes.
- General
  - Upgraded `drf-nested-routers==0.11.1`
  - Re-organized nsot.api.urls to improve readability
  - Implemented natural_key mappings for Device and Network resources
- Networks
  - Updated `Network.objects.get_by_address()` to support optional site= argument for filtering by site_id.
- Serializers
  - Moved `.create()`, `.update()` methods from Device, Network serializers to new `ResourceSerializer` base.
- Change Events
  - Fix when deleting a resource object using the API failed for any reason the "Delete" change event would still be created. The Change event will now only be kept *after* a successful delete.
- Views
  - Implemented `NsotViewSet.get_object()` support for nested serializers
  - Updated Network lookup_value_regex to support loookup by pk or IPv4/IPv6 natural_key.

0.13.2 (2016-02-16)

- Fix #142 - Properly catch non-serializer errors in API views.
- This includes unique constraints and integrity errors.
- Added a regression test for this error case.

0.13.1 (2016-02-11)

- Fixes #82: Implemented for regex-based attribute lookups via the API.
- You now may append `_regex` to an attribute name in a set query and provide a regex pattern as the value to perform regex-based lookups.

0.13 (2016-02-02)

- Implement API filtering for value objects & perf. tweaks (Fixes #137)
- Value objects now have a site_id attribute that is hidden and automatically populated by their parent Attribtue, similarly to Interface objects and their parent Device.
- New API endpoint: `/api/sites/:site_id/values/`
- Improved performance when creating/updating Interface objects by not explicitly looking up the parent Device object EVERY time.

0.12.7 (2015-12-23)

- Small tweaks to web UI
- Site index page
  - Interface count now added to Site dashboard
  - Links for ipv4/6 and network usage now link to filtered Network list
  - "Changes" renamed to "Recent Changes"
  - Under "Network Usage", "In Use" renamed to "Assigned"
- Networks list
  - Added "ip_version" filter
  - Columns now link to filtered Network list

0.12.5 (2015-12-18)

- Upgrade to Django==1.8.7 and DRF==3.3.2
- Filter fields now implemented in Browsable API (new in DRF 3.3)
- Added django-crispy-forms as a dependency
- Bootstrap JS updated to v3.3.5
- Bower updated to include Bootstrap fonts (DRF 3.3. needs this)

0.12.4 (2015-12-09)

- Made `include_ips=True` default when retrieving Networks (fix #120)

0.12.3 (2015-12-04)

- Implemented basic support for Interfaces in Web UI.
  - Create, update, delete all work
  - Device still only showing by id, should be displayed by hostname
  - Type only showing by id, should be displayed as selection of human-readable names derived from the schema.

0.12.2 (2015-12-03)

- Use native 'inet' type for storing IPs in PostgreSQL. (fixes #128)

0.12.1 (2015-11-19)

- Fix 500 crash when querying OPTIONS to view schema in API (fixes #126)
  - The bulk update mixin had to be subclassed to utilize super(), because it does not extend a pre-existing django-rest-framework mixin.
  - The inheritance order of the bulk mixins used in the Resource viewset also had to be changed because of this.
- Cleanup: The viewset for Attributes now inherits from ResourceViewSet.
- Cleanup: The viewset for Sites has been moved before ResourceViewSet for better readability (because Site is not a Resource type).

0.12 (2015-11-17)

- Basic concept of Network states, one of which is 'reserved'.
- Interaction w/ Interfaces to set state='assigned' on Assignment.
- Basic API endpoint to view `/api/networks/reserved/` to view reserved Networks.

0.11.7 (2015-10-29)

- Implemented more backend gunicorn options for default http service
  - max-requests: Max requests per worker before restart
  - max-requests-jitter - Random jitter in seconds between worker restart
  - preload - Whether to preload app before forking

0.11.6 (2015-10-22)

- Disabled caching by default (set to dummy caching)
- Added a section in the config docs for caching.
- Updated `requirements-dev.txt` to (re-)include `sphinx-autobuild`

0.11.5 (2015-10-20)

- Update Interface serializer to properly encode None as JSON.
  - It was encoding it as a string (`'None'`) vs. objects (`null`)

0.11.4 (2015-10-20)

- Fix to allow null values for MAC address on Interfaces.
- Serializer and model fields now allow MAC to be set to None.
- Also added missing lines to MANIFEST.in causing missing static/templates, which is problematic for new dev. environments or external contributors.

0.11.3 (2015-10-20)

- MAC address bugfix and a little cleanup in exceptions and validation
- Integers are no longer being improperly cast to strings and then back to an incorrect integer representation. (fixes #111)
- Added extra unit tests and regression tests for this bug.
- Moved all references to exceptions into `nsot.exc`.
- Moved email validator to `nsot.validators`.

0.11.2 (2015-10-16)

- Updated nsot-server management commands to Django 1.8 syntax
- Bugfix in user_proxy in string formatting on startup
- Implemented support for `-v/--verbosity` flag in nsot-server commands to adjust loglevel (fix #59)
- Cleaned up the gunicorn service to read from CLI args
- Updated `test_settings.py` to include some of the newer settings.

0.11.1 (2015-10-15)

- Made gunicorn worker timeout configurable by CLI or settings.py
- New setting: `settings.NSOT_NUM_WORKERS` (default: 4) to tweak number of workers
- New setting: `settings.NSOT_WORKER_TIMEOUT` (default: 30) to tweak default worker timeout
- `nsot-server start` now takes a `-t/--timeout` option at runtime to override globald defaults.

0.11 (2015-10-15)

- Enabled caching for Interface API endpoints.
- Cache is invalidated on save or delete of an Interface object.

0.10.6 (2015-10-13)

- Removed stale deps. and updated core deps. to latest stable versions

0.10.5 (2015-10-13)

- Bugfix when explicitly setting `parent_id=None` on Interface create.

0.10.4 (2015-10-13)

- Implemented bulk update of all objects using the REST API.
- Objects can now be bulk-updated using PUT by providing a list of updated objects as the payload.
- Unit tests have been updated accordingly to test for both bulk create and bulk update.

0.10.3 (2015-10-08)

- Added a Vagrantfile, improved documentation, and made some UX fixes.
- Read auth header from settings vs. hard-coded inside of user_proxy command (fix #57)
- User proxy now also defers to default values from within settings.py
- Added a vagrant directory containing a Vagrantfile to bootstrap NSoT in a self-contained virtual machine
- Added a new 'assignments' endpoint for Networks, to tell where they are being assigned to Interfaces.
- Added new nsot.utils.stats and ability to calculate network utilization.

0.10.2 (2015-10-08)

- Always return empty query when set query is invalid (fix #99)

0.10.1 (2015-10-08)

- Improved indexing on common attribute-value lookups.
- All attribute-value lookups are index now by the most commonly used search patterns (name, value, resource_name) and (resource_name, resource_id)
- Moved Interface.get_networks() and Interface.get_addresses() to used concrete JSON cache fields on the objects. This is a huge query-time optimization.
- Tweaked admin panel fields a little bit to remove references to now defunct 'Resource' objects.

0.10 (2015-10-05)

- Overhauled the relationship between Values and Resources.
- Drastic performance improvement and more accurate indexing of attribute Values in databases with millions of rows.
- Got rid of multi-table inheritance from base Resource model that was used to allow a generic foreign key from attribute Values to Resources (Devices, Networks, Interfaces are all resources)

- All Resource subclasses are *abstract* now. Which means the model fields they inherit are concrete on their own table.
- The Value object does not have an FK, and instead has a composite primary key to (resource_name, resource_id) ... for example ('Device', 16999) which is indexed together.
- The Attribute name is now also stored in a concrete field on the Value at creation, eliminating a lookup to the Attribute table.
- All of these changes are accounted for in the database migrations, but need to be done carefully! It's going to be quicker and easier for databases that don't have Interfaces.

0.9.4 (2015-10-02)

- Bug and performance fixes for Interface objects.
- Fix poor performance when there are lots of Interface objects.
- Bugfix to missing interface type 53 (proprietary virtual/internal)
- Added `smart_selects==1.1.1` so that FK lookups on Interface.parent will be limited to owning Device.
- Temporarily convert Interface.parent_id to raw ID field, until an autocomplete feature can be added to the browsable API.
- Updated unit tests to validated CRUD for Interface.parent_id.

0.9.3 (2015-09-30)

- Fix a 500 crash when database `IntegrityError` happens.
  - This will now be treated as a `409 CONFLICT`.

0.9.2 (2015-09-30)

Schema change to fix confusion when selecting parent objects.

- Benchmarks for Network and Interface objects are a *little* faster now too, direct table access for parent.
- Device objects no longer have an extraneous parent attribute.

0.9.1 (2015-09-29)

- Enhanced and clarified sections in README.rst
- Converted README from .md to .rst
- Clarified virtualenvwrapper instructions (fix #90)
- Made use of git clone more explicit (fix #91)
- Updated required version of Django REST Framework to v3.2.4

0.9 (2015-08-06)

- Implemented top-level Interface resource object.
- Addresses are assigned to Interfaces by way of Assignment objects, which are used to enforce relationship-level constraints on the assignment of Network objects to Device Interfaces.
- A Device can zero or more Interfaces; an Interface can have multiple addresses, and addresses are 'assigned' to Interfaces
- Networks are derived as the parent networks of the addresses for each interface.
- Moved hard-coded variable data in models.py into module-global constants.
- Renamed all model "choices" lists to end in "_CHOICES"
- New requirements: django-macaddress v1.3.2, Django v1.8.4
- Updated README.md to include IRC mention.
- All constants moved from `nsot.constants` to `nsot.conf.settings` and `nsot.constants` has been eliminiated. (fix #87)
- All data validators have been moved to `nsot.validators` and added new validators for cidr and host addresses.
- Moved `.to_representation()` methods on all 'resource' serializers to the top-level `nsot.api.serializers.NsotSerializer`
- Fixed a crash when creating `Network` objects without the CIDR being unicode.
- Fixed a bug when looking up a single object in API without providing site_pk
- Moved IP_VERSIONS and HOST_PREFIXES into settings.py
- IP assignments must now be unique to a device/interface tuple.
- Addresses can now be explicitly assigned to an interface, or overwritten
- Added a new `nsot.serializers.JSONListField` type to serialize JSON <-> Python lists
- Added util for deriving attributes from custom model fields that required custom serializer fields.
- Added `tests.api_tests.util.filter_interfaces` for simplifying `Interface` testing.
- Added 'ip_version' as a filter field for `Network` API lookups.

0.8.6 (2015-07-29)

- Add remote IP address in request logger.

0.8.5 (2015-07-24)

- Broke out media (css, etc.), nav, and scripts into their own include files.
- Updated main FeView to inherit default template context
- Added a template context processor to globally modify template context to inject app version.
- Added API and API Reference to dropdown "gear" menu
- Fix #77 - Only collect static files on `nsot-server start` if `settings.SERVE_STATIC_FILES=True`.

0.8.4 (2015-07-20)

- Fix including of static files in setup.py install.
- Also make sure that tests packages aren't included.

0.8.3 (2015-07-20)

- Improvements to managing static files and other server mgmt fixups.
- The default `STATIC_ROOT` setting has been changed back to `$BASE_DIR/staticfiles`
- Added 'staticfiles' to `.gitignore`
- The 'nsot-server start' command has been updated to collect the static files automatically. This can be disabled by passing `--no-collectstatic`.
- Renamed `nsot-server --noupgrade` to `--no-upgrade`
- Added help text to `nsot-server start` arguments.
- Added a URL redirect handler for `favicon.ico` (fixes #73) and included a placeholder favicon and included a `<link>` in the web UI template.
- Replaced package_data in `setup.py` with grafting files in `MANIFEST.in`
- Updated the `setup.py sdist` command to *truly* include the built static files prior to making the distribution.
- Updated Django requirement to v1.8.3

0.8.2 (2015-07-19)

- Large update to FE build/dist!
- We're now using npm to manage our frontend dev dependencies and gulp to manage our front end builds
- Add some node files and built assets to .gitignore
- Gulp added w/ tasks for linting, caching templates, annotating ng DI, concat, minify, etc.
- Setup npm devDependencies and shrinkwrap them for consistent build
- Relocated js/css into src directory that isn't included with dist build
- Updated angular code to not explicitly put DI params twice since that happens at build
- Angular templates are now compiled to javascript and added to the template cache
- Fixed some lint errors (semicolons!)
- setup.py updated to support running all tests (python & javascript)
- setup.py updated to build static on develop/sdist commands
- Removed 3rd party deps from the checked in repo
- Fixed MANIFEST.in to not include pyc's under tests

0.8.1 (2015-07-16)

- Implement network/address allocation endpoints for Network objects.
- For database models the following methods have been added:
  - `get_next_address()` - Returns a list of next available a addresses (fixes #49)
  - `get_next_network()` - Returns a list of next available networks matching the provided prefix_length. (fixes #48)
- For the REST API, the following endpoints have been added to Network objects in detail view (e.g. `GET /api/sites/1/networks/10/:endpoint1`):
  - `next_address` - Returns a list of next available a addresses
  - `next_network` - Returns a list of next available networks matching the provided prefix_length.
  - `parent` - Return the parent Network for this Network
- Updated all of the tree traversal methods to explicitly order results by (network_address, prefix_length) so that results are in tree order.
- Corrected a typo in the README file (fixes #69)
- All new functionality is completely unit-tested!

0.8 (2015-07-16)

- Implement tree traversal endpoints for Network objects.
- For database models the following methods have been added:
  - `is_child_node()` - Returns whether Network is a child node
  - `is_leaf_node()` - Returns whether Network has no children
  - `is_root_node()` - Returns whether Network has no parent
  - `get_ancestors()` - Return all parents for a Network
  - `get_children()` - Return immediate children for a Network
  - `get_descendents()` - Return ALL children for a Network
  - `get_root()` - Return the root node of this Network
  - `get_siblings()` - Returns Networks with the same parent
- For the REST API, the following endpoints have been added to Network objects detail view (e.g. `GET /api/sites/1/networks/10/:endpoint`):
  - `ancestors` - Return all parents for a Network
  - `children` - Return immediate children for a Network
  - `descendents` - Return ALL children for a Network
  - `root` - Return the root node of this Network
  - `siblings` - Returns Networks with the same parent
- All new functionality is completely unit-tested!

0.7.4 (2015-07-14)

- Multiple bug fixes related to looking up Attributes using set queries.
- Fix #66 - Handle 500 error when multiple Sites contain an Attribute of the same name.
- Fix #67 - Bugfix when an Attribute name isn't found when performing a set query.
- Resource.objects.set_query() now takes an optional site_id argument that will always be sent when called internally by the API.
- Added site_id to repr for Attribute objects to make it less confusing when working with multiple sites containing Attributes of the same name.
- Fixed a bug in Attribute.all_by_name() that would cause the last Attribute matching the desired name, even if the site_id conflicted with the parent resource object. Attribute.all_by_name() now requires a site argument.
- If a set query raises an exception (such as when no matching Attribute is found), an empty queryset is returned.

0.7.3 (2015-07-09)

- Fix #58: Typo in permissions docs
- Fix #64: New command to generate key

0.7.2 (2015-07-07)

- Fix #62 - 500 error when API authenticate is malformed.

0.7.1 (2015-07-02)

- Remove need to "collectstatic", remove 'nsot.log' log handler.
  - Static files will default to being served from within the nsot library itself, eliminating the need to colectstatic.
  - nsot-server will no longer drop an empty nsot.log file in the directory from which it is called.

0.7 (2015-07-01)

- Replace backend with Django + Django REST Framework + Gunicorn + Gevent

0.5.6 (2015-06-15)

- Actually pass num_processes down to tornado

0.5.5 (2015-06-11)

- Fix #46: Purge attribute index before a Device object is deleted.

0.5.4 (2015-06-08)

- Update libs and small UI fixes
  - Add filter options to networks page
  - css cleanup
  - Fix bug where all changes were for site id 1. fixes #51
  - Update libraries to later versions to get some new features.

0.5.3 (2015-05-29)

- Bugfix in validating Attribute when constraints are not dict.

0.5.2 (2015-04-13)

- Fix #40 Auth token verification now uses session from request handler
  - This is very difficult to reproduce, so changing the request handler (which is currently the only caller of User.verify_auth_token()) to send its own session when calling is a best guess at solving this.k

0.5.1 (2015-04-13)

- Fix #41 so set queries on networks include optional filter arguments.

0.5 (2015-04-07)

- Add support for logging errors to Sentry if sentry_dsn is set.

0.4.4 (2015-04-02)

- Bugfix for displaying IPs when filtering Networks w/ attrs. (fix #34)
- Added some extra networks to the test fixtures for API tests.
- Updated fixtures for network set queries to reflect extra networks.

0.4.3 (2015-04-01)

- UI Updates
  - fixes #19
  - fixes #32
- Show attributes on Device/Network pages.
- Show latest changes on Device/Network pages.
- Provide NSOT_VERSION to jinja and angular templates.
- Show version in NSoT UI

0.4.1 (2015-03-31)

- Only import mrproxy for user_proxy arg in nsot-ctl. (fixes #24)

0.4 (2015-03-31)

- Add support for filtering networks by cidr/addr/prefix/attrs. (fix #18)

0.3.3 (2015-03-30)

- If restrict_networks is null, treat it as an empty list. (fix #22)

0.3.2 (2015-03-30)

- Explicitly include and order all dependent packages.
  - This is so that enum34 (dependency of cryptography) can be properly installed using an internal PyPI mirror (See: https://github.com/pyca/cryptography/issues/1803)
- Removed six from requirements-dev.txt
- Bumped version to differentiate these underlying changes.

0.3.1 (2015-03-19)

- Allow lookup of Devices by hostname or attributes.

0.3 (2015-03-12)

- Added support for set operation queries on Devices and Networks.
- New "query" endpoint on each of these resources take a "?query=" argument that is a string representation of attribute/value pairs for intersection, difference, and union operations.
- All new functionality unit tested!

0.2.2 (2015-03-06)

- Bugfix for 500 error when creating Network w/ null cidr (fixes #13)

0.2.1 (2015-03-05)

- Bug fix for 500 error when validating null hostname (fixes #11)

0.2.0 (2015-03-04)

- Added support for bulk creation of Attributes, Devices, and Networks
- When creating a collection via POST, a 201 CREATED response is generated without a Location header. The payload includes the created objects.

0.1.0 (2015-02-28)

- Bugfix in string format when validating attribute that doesn't exist.

0.0.9 (2015-02-10)

- Implemented API key (auth_token) authentication
- Cookies are now stored as secure cookies using cookie_secret setting.
- New site setting for storing secret_key used for crypto.
- User has a new .secret_key field which is generated when User is created
  - User should obtain key through web UI (however that is NYI)
  - Secret key is used as user password to generate an auth_token
- Auth token is serialized, and encrypted with server's key and also contains an expiration timestamp (default 10 minutes)
- AuthToken can be done using "Authorization" header or query args.
- New User methods for generating and validating auth_token
- API endpoints still also accept "default" login methods.
- Added a models.get_db_session() function to make getting a session easier!
- Added a Model.query classmethod to make model queries easier!!
- All new changes are unit tested!
- If you're checking out the API auth stuff and want to test it out, see the README.auth.rst file!
- Web views use "default" auth (currently user_auth_header)
- API views use "default" or "auth_token"
- AuthToken can now be done using "Authorization" header or query args.

0.0.2 (2015-01-12)

- Add setting to toggle for checking XSRF cookies on API calls.

0.0.1 (2014-12-03)

- Initial scaffolding for NSoT
- Python packaging
- Inital models
- Support for add/remove/update/list Sites

CHAPTER 10

Support

Network Source of Truth is a primarily a Pacific coast operation, so your best chance of getting a real-time response is during the weekdays, Pacific time.

The best way to get support, provide feedback, ask questions, or to just talk shop is to find us online on Slack.

We have a bi-directional Slack/IRC bridge for our channels so that users can use whichever they prefer.

Slack

We hangout on Slack on the NetworkToCode team.

1. Request to join the team (it's free!)
2. Join us in `#nsot`.

IRC

If you want to keep it old school and use IRC, find us at `#nsot` on Freenode (irc://irc.freenode.net/nsot).

n

A

B

C

D

E

F

G

I

J

T

U

V