salt-sproxy Documentation

Mircea Ulinic

Feb 20, 2020

Contents

1 Install 3

2 Quick Start 5

3 Usage 7

4 Docker 9

5 More usage examples 11

6 Extension Modules 31

7 See Also 47

Python Module Index 91

Index 93

Salt plugin to automate the management and configuration of network devices at scale, without running (Proxy) Minions.

Using `salt-sproxy`, you can continue to benefit from the scalability, flexibility and extensibility of Salt, while you don't have to manage thousands of (Proxy) Minion services. However, you are able to use both `salt-sproxy` and your (Proxy) Minions at the same time.

Note: This is NOT a SaltStack product.

CHAPTER 1

Install

Install this package where you would like to manage your devices from. In case you need a specific Salt version, make sure you install it beforehand, otherwise this package will bring the latest Salt version available instead.

The package is distributed via PyPI, under the name `salt-sproxy`.

Execute:

```
pip install salt-sproxy
```

See *Installation* for more detailed installation notes.

CHAPTER 2

Quick Start

See this recording for a live quick start:

In the above, `minion1` is a dummy Proxy Minion, that can be used for getting started and make the first steps without connecting to an actual device, but get used to the `salt-sproxy` methodology.

The Master configuration file is `/home/mircea/master`, which is why the command is executed using the `-c` option specifying the path to the directory with the configuration file. In this Master configuration file, the `pillar_roots` option points to `/srv/salt/pillar` which is where `salt-sproxy` is going to load the Pillar data from. Accordingly, the Pillar Top file is under that path, `/srv/salt/pillar/top.sls`:

```
base:
  minion1:
    - dummy
```

This Pillar Top file says that the Minion `minion1` will have the Pillar data from the `dummy.sls` from the same directory, thus `/srv/salt/pillar/dummy.sls`:

```
proxy:
  proxytype: dummy
```

In this case, it was sufficient to only set the `proxytype` field to `dummy`.

`salt-sproxy` can be used in conjunction with any of the available Salt Proxy modules, or others that you might have in your own environment. See https://docs.saltstack.com/en/latest/topics/proxyminion/index.html to understand how to write a new Proxy module if you require.

For example, let's take a look at how we can manage a network device through the NAPALM Proxy:

In the above, in the same Python virtual environment as previously make sure you have `NAPALM` installed, by executing `pip install napalm` (see https://napalm.readthedocs.io/en/latest/installation/index.html for further installation requirements, depending on the platform you're running on). The connection credentials for the `juniper-router` are stored in the `/srv/salt/pillar/junos.sls` Pillar, and we can go ahead and start executing arbitrary Salt commands, e.g., net.arp to retrieve the ARP table, or net.load_config to apply a configuration change on the router.

The Pillar Top file in this example was (under the same path as previously, as the Master config was the same):

```
base:
  juniper-router:
    - junos
```

Thanks to Tesuto for providing the virtual machine for the demos!

CHAPTER 3

Usage

First off, make sure you have the Salt Pillar Top file correctly defined and the `proxy` key is available into the Pillar. For more in-depth explanation and examples, check this tutorial from the official SaltStack docs.

Once you have that, you can start using `salt-sproxy` even without any Proxy Minions or Salt Master running. To check, can start by executing:

```
$ salt-sproxy -L a,b,c --preview-target
- a
- b
- c
```

The syntax is very similar to the widely used CLI command `salt`, however the way it works is completely different under the hood:

`salt-sproxy <target> <function> [<arguments>]`

Usage Example:

```
$ salt-sproxy cr1.thn.lon test.ping
cr1.thn.lon:
    True
```

One of the most important differences between `salt` and `salt-sproxy` is that the former is aware of the devices available, thanks to the fact that the Minions connect to the Master, therefore `salt` has the list of targets already available. `salt-sproxy` does not have this, as it doesn't require the Proxy Minions to be up and connected to the Master. For this reason, you will need to provide it a list of devices, or a Roster file that provides the list of available devices.

The following targeting options are available:

- `-E`, `--pcre`: Instead of using shell globs to evaluate the target servers, use pcre regular expressions.
- `-L`, `--list`: Instead of using shell globs to evaluate the target servers, take a comma or space delimited list of servers.
- `-G`, `--grain`: Instead of using shell globs to evaluate the target use a grain value to identify targets, the syntax for the target is the grain key followed by a globexpression: `"os:Arch*"`.

- `-P`, `--grain-pcre`: Instead of using shell globs to evaluate the target use a grain value to identify targets, the syntax for the target is the grain key followed by a pcre regular expression: "os:Arch.*".
- `-N`, `--nodegroup`: Instead of using shell globs to evaluate the target use one of the predefined nodegroups to identify a list of targets.
- `-R`, `--range`: Instead of using shell globs to evaluate the target use a range expression to identify targets. Range expressions look like %cluster.

Warning: Some of the targeting options above may not be avaialble for some Roster modules.

To use a specific Roster, configure the `proxy_roster` (or simply `roster`) option into your Master config file, e.g.,

```
proxy_roster: ansible
```

Note: It is recommended to prefer the `proxy_roster` option in the favour of `roster` as the latter is used by Salt SSH. In case you want to use both salt-sproxy and Salt SSH, you may want to use different Roster files, which is why there are two different options.

salt-sproxy will evauluate both `proxy_roster` and `roster`, in this order.

With the configuration above, `salt-sproxy` would try to use the ansbile Roster module to compile the Roster file (typically `/etc/salt/roster`) which is structured as a regular Ansible Inventory file. This inventory should only provide the list of devices.

The Roster can also be specified on the fly, using the `-R` or `--roster` options, e.g., `salt-sproxy cr1.thn.lon test.ping --roster=flat`. In this example, we'd be using the flat Roster module to determine the list of devices matched by a specific target.

When you don't specify the Roster into the Master config, or from the CLI, you can use `salt-sproxy` to target on or more devices using the `glob` or `list` target types, e.g., `salt-sproxy cr1.thn.lon test.ping` (glob) or `salt-sproxy -L cr1.thn.lon,cr2.thn.lon test.ping` (to target a list of devices, `cr1.thn.lon` and `cr2.thn.lon`, respectively).

Note that in any case (with or without the Roster), you will need to provide a valid list of Minions.

CHAPTER 4

Docker

There are Docker images available should you need or prefer: https://hub.docker.com/r/mirceaulinic/salt-sproxy.

You can see here the available tags: https://hub.docker.com/r/mirceaulinic/salt-sproxy/tags. `latest` provides the code merged into the `master` branch, and `allinone-latest` is the code merged into the `master` branch with several libraries such as NAPALM, Netmiko, ciscoconfparse, or Ansible which you may need for your modules or Roster (if you'd want to use the Ansible Roster, for example).

These can be used in various scenarios. For example, if you would like to use `salt-proxy` but without installing it, and prefer to use Docker instead, you can define the following convoluted alias:

```
alias salt-sproxy='f(){ docker run --rm --network host -v $SALT_PROXY_PILLAR_DIR:/etc/
↪salt/pillar/ -ti mirceaulinic/salt-sproxy salt-sproxy $@; }; f'
```

And in the `SALT_PROXY_PILLAR_DIR` environment variable, you set the path to the directory where you have the Pillars, e.g.,

```
export SALT_PROXY_PILLAR_DIR=/path/to/pillars/dir
```

With this setup, you would be able to go ahead and execute "as normally" (with the difference that the code is executed inside the container, however from the CLI it won't look different):

```
salt-sproxy minion1 test.ping
```

CHAPTER 5

More usage examples

See the following examples to help getting started with salt-sproxy:

5.1 Usage Examples

5.1.1 salt-sproxy 101

This is the first example from the Quick Start section of the documentation.

Using the Master configuration file under examples/master:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar
```

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/master /etc/salt/master
$ cp salt-sproxy/examples/101/pillar/*.sls /srv/salt/pillar/
```

The contents of these two files:

`/srv/salt/pillar/top.sls`:

```
base:
  mininon1:
    - dummy
```

`/srv/salt/pillar/dummy.sls`:

```
proxy:
  proxytype: dummy
```

Having this setup ready, you can go ahead an execute:

```
$ salt-sproxy minion1 test.ping
minion1:
    True

# let's display the list of packages installed via pip on this computer
$ salt-sproxy minion1 pip.list
minion1:
    ----------
    Jinja2:
        2.10.1
    MarkupSafe:
        1.1.1
    PyNaCl:
        1.3.0
    PyYAML:
        5.1
    Pygments:
        2.4.0
    asn1crypto:
        0.24.0
    bcrypt:
        3.1.6
    bleach:
        3.1.0
    certifi:
        2019.3.9
    cffi:
        1.12.3
```

Alternative setup using Docker

1. Clone the salt-sproxy repository and change dir:

```
$ git clone https://github.com/mirceaulinic/salt-sproxy.git
$ cd salt-sproxy/
```

2. Using the `allinone-latest` Docker image (see *Docker*), you can run from this path:

```
$ docker run --rm -v $PWD/examples/101/pillar/:/srv/salt/pillar/ \
    -ti mirceaulinic/salt-sproxy:allinone-latest bash
root@2c68721d93dc:/# salt-sproxy minion1 test.ping -l error
minion1:
    True
```

5.1.2 Using the Ansible Roster

To be able to use the Ansible Roster, you will need to have `ansible` installed in the same environment as `salt-sproxy`, e.g.,

```
$ pip instal ansible
```

Using the Master configuration file under examples/ansible/master:

/etc/salt/master:

```
pillar_roots:
  base:
    - /srv/salt/pillar

proxy_roster: ansible
roster_file: /etc/salt/roster
```

Notice that compared to the previous examples, 101 and NAPALM, there are two additional options: `roster_file` which specifies the path to the Roster file to use, and `proxy_roster` that tells how to interpret the Roster file - in this case, the Roster file `/etc/salt/roster` is going to be loaded as an Ansible inventory. Let's consider, for example, the following Roster / Ansible inventory which you can find at examples/ansible/roster:

```
all:
  children:
    usa:
      children:
        northeast: ~
        northwest:
          children:
            seattle:
              hosts:
                edge1.seattle
            vancouver:
              hosts:
                edge1.vancouver
        southeast:
          children:
            atlanta:
              hosts:
                edge1.atlanta:
                edge2.atlanta:
            raleigh:
              hosts:
                edge1.raleigh:
        southwest:
          children:
            san_francisco:
              hosts:
                edge1.sfo
            los_angeles:
              hosts:
                edge1.la
```

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/ansible/master /etc/salt/master
$ cp salt-sproxy/examples/ansible/roster /etc/salt/roster
$ cp salt-sproxy/examples/ansible/pillar/*.sls /srv/salt/pillar/
```

The contents of these files:

`/srv/salt/pillar/top.sls`:

```
base:
  'edge1*':
    - junos
  'edge2*':
    - eos
```

With this top file, Salt is going to load the Pillar data from `/srv/salt/pillar/junos.sls` for `edge1.seattle`, `edge1.atlanta`, `edge1.raleigh`, `edge1.sfo`, and `edge1.la`, while loading the data from `/srv/salt/pillar/eos.sls` for `edge2.atlanta` (and anything that would match the `edge2*` expression should you have others).

`/srv/salt/pillar/junos.sls`:

```
proxy:
  proxytype: napalm
  driver: junos
  host: {{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

`/srv/salt/pillar/eos.sls`:

```
proxy:
  proxytype: napalm
  driver: eos
  host: {{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

Note that in both case the `hostname` has been set as `{{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com`. `opts.id` points to the Minion ID, which means that the Pillar data is rendered depending on the name of the device; therefore, the hostname for `edge1.atlanta` will be `edge1-atlanta.salt-sproxy.digitalocean.cloud.tesuto.com`, the hostname for `edge2.atlanta` is `edge2-atlanta.salt-sproxy.digitalocean.cloud.tesuto.com`, and so on.

Having this setup ready, you can go ahead an execute:

```
$ salt-sproxy '*' --preview-target
- edge1.seattle
- edge1.vancouver
- edge1.atlanta
- edge2.atlanta
- edge1.raleigh
- edge1.la
- edge1.sfo

# get the LLDP neighbors from all the edge devices
$ salt-sproxy 'edge*' net.lldp
edge1.vancouver:
    ~~~ snip ~~~
edge1.atlanta:
    ~~~ snip ~~~
edge1.sfo:
    ~~~ snip ~~~
```

(continues on next page)

(continued from previous page)

```
edge1.seattle:
    ~~~ snip ~~~
edge1.la:
    ~~~ snip ~~~
edge1.raleigh:
    ~~~ snip ~~~
edge2.atlanta:
    ~~~ snip ~~~
```

Alternative setup using Docker

1. Clone the salt-sproxy repository and change dir:

```
$ git clone https://github.com/mirceaulinic/salt-sproxy.git
$ cd salt-sproxy/
```

2. Update `examples/ansible/roster` with your Ansible inventory.
3. Update `examples/ansible/top.sls` to ensure your Pillar Top file matches the name of the devices from your Roster / Ansible inventory. Also, update the `examples/ansible/eos.sls`, `examples/ansible/junos.sls` etc. files with your credentials to connect to your device(s).

   To double check that the mapping is correct, you can execute:

```
$ docker run --rm -v $PWD/examples/ansible/master:/etc/salt/master \
      -v $PWD/examples/ansible/roster:/etc/salt/roster \
      -v $PWD/examples/ansible/pillar/:/srv/salt/pillar/ \
      -ti mirceaulinic/salt-sproxy:allinone-latest bash

root@2c68721d93dc:/# salt-run pillar.show_pillar edge1.atlanta
proxy:
    ----------
    proxytype:
        napalm
    driver:
        junos
    hostname:
        edge1-atlanta.salt-sproxy.digitalocean.cloud.tesuto.com
    username:
        test
    password:
        t35t1234
```

4. Using the `allinone-latest` Docker image (see *Docker*), you can run from this path:

```
$ docker run --rm -v $PWD/examples/ansible/master:/etc/salt/master \
    -v $PWD/examples/ansible/roster:/etc/salt/roster \
    -v $PWD/examples/ansible/pillar/:/srv/salt/pillar/ \
    --network host \
    -ti mirceaulinic/salt-sproxy:allinone-latest bash

root@2c68721d93dc:/# salt-sproxy -N southwest test.ping
edge1.la:
    True
edge1.sfo:
    True
```

5.1.3 Using the File Roster

The File Roster allows you to easily manage the list of devices through an SLS file - that being any combination of the available Roster modules: Jinja+YAML, YAML, JSON, pure Python, JSON5, HJSON, etc.

By default, the Roster file is `/etc/salt/roster`, but you can have a different path by configuring `roster_file` (or `--roster-file` on the command line) to point to an alternative absolute path, e.g.,

`/etc/salt/master`

```
roster: file
roster_file: /path/to/roster/file
```

For starters, let's consider the following simple Roster SLS file:

`/etc/salt/roster`

```
device1: {}
device2: {}
```

To check that everything is properly configured, you can execute:

```
$ salt-sproxy \* --preview-target
- device1
- device2
```

As always, you'll need to provide the connection credentials, in the Pillar. That is, you can have a structure as the following Pillar top file:

`/srv/pillar/top.sls`

```
base:
  '*':
    - proxy
```

And the connection credentials - example using NAPALM:

`/srv/pillar/proxy.sls`

```
proxy:
  proxytype: napalm
  driver: junos
  hostname: {{ opts.id }}.example.com
  password: superS3kure
```

With this configuration, `device1` will try to connect to `device1.example.com`, and `device2` to `device2.example.com`, respectively, using the NAPALM Junos driver.

If you want more specific connection options per device, you can manage that in the Roster SLS file (under each device you can specify any connection argument to override the details from the `proxy` Pillar), e.g.,

`/etc/salt/roster`

```
device1:
  driver: eos
  hostname: different-hostname-for-device1.example.com
device2:
  password: m0reS3kure
```

Using the previous example, `device1` will connect to `different-hostname-for-device1.example.com` using the NAPALM EOS driver for Arista, while `device2` uses a different password.

In a similar way, you can provide static Grains per device, under the `grains` key, e.g.,

`/etc/salt/roster`:

```
device1:
  grains:
    site: site1
device2:
  grains:
    site: site2
```

If you prefer to manage a JSON structure instead:

`/etc/salt/roster`:

```
{
  "device1": {
    "grains": {
      "site": "site1"
    }
  },
  "device2": {
    "grains": {
      "site": "site2"
    }
  }
}
```

With that clarified, let's make the Roster SLS file more dynamic, and instead of managing the list of devices manually, have it auto-generated:

`/etc/salt/roster`:

```
{%- for i in range(50) %}
device{{ i }}:
  grains:
    site: site{{ i }}
{%- endfor %}
```

The example above provides a list of 50 devices. Although probably too simplistic for real-world usage, it may be sufficient to exemplify the use-case.

Remember that being interpreted as an SLS, you can also invoke Salt functions, using the `__salt__` global variable. For example, to retrieve and build the list of devices dynamically using an HTTP query, you can do, e.g.,

```
{%- set ret = __salt__.http.query('https://netbox.live/api/dcim/devices/',␣
↪decode=true) %}
{%- for device in ret.dict.results %}
{{ device.name }}:
  grains:
    site: {{ device.site.slug }}
{%- endfor %}
```

Ultimately, for higher complexity, consider using the pure Python Renderer whenever you need to put more business logic in selecting the devices you need to manage.

5.1.4 salt-sproxy with network devices

This is the second example from the Quick Start section of the documentation.

To be able to use this example, make sure you have NAPALM installed - see the complete installation notes from https://napalm.readthedocs.io/en/latest/installation/index.html.

Using the Master configuration file under examples/master:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar
```

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/master /etc/salt/master
$ cp salt-sproxy/examples/napalm/pillar/*.sls /srv/salt/pillar/
```

The contents of these two files:

`/srv/salt/pillar/top.sls`:

```
base:
  juniper-router:
    - junos
```

`/srv/salt/pillar/junos.sls`:

```
proxy:
  proxytype: napalm
  driver: junos
  host: juniper.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

Having this setup ready, after you update the connection details, you can go ahead an execute:

```
$ salt-sproxy juniper-router test.ping
juniper-router:
    True

# retrieve the ARP table from juniper-router
$ salt-sproxy juniper-router net.arp
juniper-router:
    ----------
    comment:
    out:
        |_
          ----------
          age:
              849.0
          interface:
              fxp0.0
          ip:
```

(continues on next page)

(continued from previous page)

```
                10.96.0.1
            mac:
                92:99:00:0A:00:00
          |_
            ----------
            age:
                973.0
            interface:
                fxp0.0
            ip:
                10.96.0.13
            mac:
                92:99:00:0A:00:00
          |_
            ----------
            age:
                738.0
            interface:
                em1.0
            ip:
                128.0.0.16
            mac:
                02:42:AC:13:00:02
    result:
        True

# apply a configuration change: dry run
$ salt-sproxy juniper-router net.load_config text='set system ntp server 10.10.10.1'␣
↪test=True
juniper-router:
    ----------
    already_configured:
        False
    comment:
        Configuration discarded.
    diff:
        [edit system]
        +   ntp {
        +       server 10.10.10.1;
        +   }
    loaded_config:
    result:
        True

# apply the configuration change and commit
$ salt-sproxy juniper-router net.load_config text='set system ntp server 10.10.10.1'
juniper-router:
    ----------
    already_configured:
        False
    comment:
    diff:
        [edit system]
        +   ntp {
        +       server 10.10.10.1;
        +   }
    loaded_config:
```

(continues on next page)

(continued from previous page)

```
    result:
        True
```

If you run into issues when connecting to your device, you might want to go through this checklist: https://github.com/napalm-automation/napalm#faq.

Note: For a better methodology on managing the configuration, you might want to take a look at the State system, one of the most widely used State modules for configuration management through NAPALM being Netconfig.

Alternative setup using Docker

1. Clone the salt-sproxy repository and change dir:

```
$ git clone https://github.com/mirceaulinic/salt-sproxy.git
$ cd salt-sproxy/
```

2. Update the `examples/napalm/junos.sls` file with your credentials to connect to your device.
3. Using the `allinone-latest` Docker image (see *Docker*), you can run from this path:

```
$ docker run --rm -v $PWD/examples/napalm/pillar/:/srv/salt/pillar/ \
    --network host \
    -ti mirceaulinic/salt-sproxy:allinone-latest bash
root@2c68721d93dc:/# salt-sproxy juniper-router test.ping
juniper-router:
    True
root@2c68721d93dc:/# salt-sproxy juniper-router net.load_config \
    text='set system ntp server 10.10.10.1' test=True
juniper-router:
    ----------
    already_configured:
        False
    comment:
        Configuration discarded.
    diff:
        [edit system]
        +   ntp {
        +       server 10.10.10.1;
        +   }
    loaded_config:
    result:
        True
```

5.1.5 Using the NetBox Roster

To be able to use the NetBox Roster, you will need to have the `pynetbox` library installed in the same environment as `salt-sproxy`, e.g.,

```
$ pip install pynetbox
```

Using the Master configuration file under examples/netbox/master:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar

proxy_roster: netbox

netbox:
  url: https://url-to-your-netbox-instance
```

With this configuration, the list of devices is going to be loaded from NetBox, with the connection details provides under the `netbox` key.

Note: To set up a NetBox instance, see the installation notes from https://netbox.readthedocs.io/en/stable/installation/.

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/netbox/master /etc/salt/master
$ cp salt-sproxy/examples/netbox/pillar/*.sls /srv/salt/pillar/
```

The contents of these files highly depend on the device names you have in your NetBox instance. The following examples are crafted for device name starting with `edge1` and `edge2`, e.g., `edge1.atlanta`, `edge1.seattle` etc. If you have different device names in your NetBox instance, you'll have to update these Pillars.

`/srv/salt/pillar/top.sls`:

```
base:
  'edge1*':
    - junos
  'edge2*':
    - eos
```

With this top file, Salt is going to load the Pillar data from `/srv/salt/pillar/junos.sls` for `edge1.seattle`, `edge1.atlanta`, `edge1.raleigh`, `edge1.sfo`, and `edge1.la`, while loading the data from `/srv/salt/pillar/eos.sls` for `edge2.atlanta` (and anything that would match the `edge2*` expression should you have others).

`/srv/salt/pillar/junos.sls`:

```
proxy:
  proxytype: napalm
  driver: junos
  host: {{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

`/srv/salt/pillar/eos.sls`:

```
proxy:
  proxytype: napalm
  driver: eos
  host: {{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

Note that in both case the `hostname` has been set as `{{ opts.id | replace('.', '-') }}.salt-sproxy.digitalocean.cloud.tesuto.com`. `opts.id` points to the Minion ID, which means that the Pillar data is rendered depending on the name of the device; therefore, the hostname for `edge1.atlanta` will be `edge1-atlanta.salt-sproxy.digitalocean.cloud.tesuto.com`, the hostname for `edge2.atlanta` is `edge2-atlanta.salt-sproxy.digitalocean.cloud.tesuto.com`, and so on.

Having this setup ready, you can go ahead an execute:

```
$ salt-sproxy '*' --preview-target
- edge1.seattle
- edge1.vancouver
- edge1.atlanta
- edge2.atlanta
- edge1.raleigh
- edge1.la
- edge1.sfo
~~~ many others ~~~

# get the LLDP neighbors from all the edge devices
$ salt-sproxy 'edge*' net.lldp
edge1.vancouver:
    ~~~ snip ~~~
edge1.atlanta:
    ~~~ snip ~~~
edge1.sfo:
    ~~~ snip ~~~
edge1.seattle:
    ~~~ snip ~~~
edge1.la:
    ~~~ snip ~~~
edge1.raleigh:
    ~~~ snip ~~~
edge2.atlanta:
    ~~~ snip ~~~
```

Alternative setup using Docker

1. Clone the salt-sproxy repository and change dir:

```
$ git clone https://github.com/mirceaulinic/salt-sproxy.git
$ cd salt-sproxy/
```

2. Update `examples/netbox/master` with your NetBox details (URL and token).

   Alternatively, for quick testing, you can also leave the existing values, to use the demo instance available at https://netbox.live.

3. Using the `allinone-latest` Docker image (see *Docker*), you can run from this path (at the repository root):

```
$ docker run --rm -v $PWD/examples/netbox/master:/etc/salt/master \
    -v $PWD/examples/netbox/pillar/:/srv/salt/pillar/ \
    --network host \
    -ti mirceaulinic/salt-sproxy:allinone-latest bash

root@2c68721d93dc:/# salt-sproxy \* --preview-target
- edge1.vlc1
```

5.1.6 Using the Pillar Roster

You can think of the Pillar Roster as a Roster that loads the list of devices / inventory dynamically using the Pillar subsystem. Or, in simpler words, you can use any of these features from here: https://docs.saltstack.com/en/latest/ref/pillar/all/index.html to load the list of your devices, including: JSON / YAML HTTP API, load from MySQL / Postgres database, LDAP, Redis, MongoDB, etcd, Consul, and many others; needless to say that this is another pluggable interface and, in case you have a more specific requirement, you can easily extend Salt in your environment by providing another Pillar module under the `salt://_pillar` directory. For example, see this old yet still accurate article: https://medium.com/@Drew_Stokes/saltstack-extending-the-pillar-494d41ee156d.

The core idea is that you are able to use the data pulled via the Pillar modules once you are able to execute the following command and see the list of devices you're aiming to manage:

```
$ salt-run pillar.show_pillar
devices:
  - name: device1
  ...
```

It really doesn't matter where is Salt pulling this data from.

By default, the Pillar Roster is going to check the Pillar data for `*` (any Minion), and load it from the `devices` key. In other words, when executing `salt-sproxy pillar.show_pillar` the output should have at least the `devices` key. To use different settings, have a look at the documentation: *Pillar Roster*.

Say we want to pull the list of devices from an HTTP API module providing the data in JSON format. In this case, we can use the http_json module.

If the data is available at http://example.com/devices, and you can verify, e.g., using `curl`:

```
$ curl http://example.com/devices
{"devices": [{"name": "router1"}, {"name": "router2"}, {"name": "switch1"}]}
```

That being available, we can configure the `http_json` External Pillar:

`/etc/salt/master`:

```
roster: pillar

ext_pillar:
  - http_json:
      url: http://example.com/devices
```

Now, let's verify that the data is pulled properly into the Pillar:

```
$ salt-run pillar.show_pillar
devices:
  - name: router1
  - name: router2
  - name: switch1
```

That being validated, salt-sproxy is now aware of all the devices to be managed:

```
$ salt-sproxy \* --preview-target
- router1
- router2
- switch1
```

As well as other target types such as `list` or `PCRE`:

```
# target a fixed list of devices:

$ salt-sproxy -L router1,router2 --preview-target
- router1
- router2

# target all devices with the name starting with "router",
# followed by one or more numbers:

$ salt-sproxy -E 'router\d+' --preview-target
- router1
- router2
```

The same methodology applies to any of the other External Pillar modules.

5.1.7 Salt REST API

Important: In the configuration examples below, for simplicity, I've used the auto external authentication, and disabled the SSL for the Salt API. This setup is highly discouraged in production.

Using the Master configuration file under examples/salt_api/master:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar

file_roots:
  base:
    - /srv/salt/extmods

rest_cherrypy:
  port: 8080
  disable_ssl: true

external_auth:
  auto:
    '*':
      - '@runner'
```

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/salt_api/master /etc/salt/master
$ cp salt-sproxy/examples/salt_api/pillar/*.sls /srv/salt/pillar/
```

The contents of Pillar files:

`/srv/salt/pillar/top.sls`:

```
base:
  mininon1:
```

(continues on next page)

(continued from previous page)

```
    - dummy
  juniper-router:
    - junos
```

/srv/salt/pillar/dummy.sls:

```
proxy:
  proxytype: dummy
```

/srv/salt/pillar/junos.sls:

```
proxy:
  proxytype: napalm
  driver: junos
  host: juniper.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

Note: The `top.sls`, `dummy.sls`, and `junos.sls` are a combination of the previous examples, 101 and napalm, which is going to allow use to execute against both the dummy device and a real network device.

In the example Master configuration file above, there's also a section for the `file_roots`. As documented in The Proxy Runner section of the documentation, you are going to reference the proxy Runner, e.g.

```
$ mkdir -p /srv/salt/extmods/_runners
$ cp salt-sproxy/salt_sproxy/_runners/proxy.py /srv/salt/extmods/_runners/
```

Or symlink:

```
$ ln -s /path/to/git/clone/salt-sproxy/salt_sproxy /srv/salt/extmods
```

With the `rest_cherrypy` section, the Salt API will be listening to HTTP requests over port 8080, and SSL being disabled (not recommended in production):

```
rest_cherrypy:
  port: 8080
  disable_ssl: true
```

One another part of the configuration is the external authentication:

```
external_auth:
  auto:
    '*':
      - '@runner'
```

This grants access to anyone to execute any Runner (again, don't do this in production).

With this setup, we can start the Salt Master and the Salt API (running in background):

```
$ salt-master -d
$ salt-api -d
```

To verify that the REST API is ready, execute:

```
$ curl -i localhost:8080
HTTP/1.1 200 OK
Content-Type: application/json
Server: CherryPy/18.1.1
Date: Wed, 05 Jun 2019 07:58:32 GMT
Allow: GET, HEAD, POST
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: GET, POST
Access-Control-Allow-Credentials: true
Vary: Accept-Encoding
Content-Length: 146

{"return": "Welcome", "clients": ["local", "local_async", "local_batch", "local_subset
↪", "runner", "runner_async", "ssh", "wheel", "wheel_async"]}
```

Now we can go ahead and execute the CLI command from example 101, by making an HTTP request:

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='auto' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='minion1' \
  -d function='test.ping' \
  -d sync=True
return:
- minion1: true
```

Notice that `eauth` field in this case is `auto` as this is what we've configured in the `external_auth` on the Master.

Similarly, you can now execute the Salt functions from the NAPALM example, against a network device, by making an HTTP request:

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='auto' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='juniper-router' \
  -d function='net.arp' \
  -d sync=True
return:
- juniper-router:
    comment: ''
    out:
    - age: 891.0
      interface: fxp0.0
      ip: 10.96.0.1
      mac: 92:99:00:0A:00:00
    - age: 1001.0
      interface: fxp0.0
      ip: 10.96.0.13
      mac: 92:99:00:0A:00:00
    - age: 902.0
      interface: em1.0
      ip: 128.0.0.16
```

(continues on next page)

(continued from previous page)

```
      mac: 02:42:AC:12:00:02
    result: true
```

5.1.8 salt-sapi

Note: This functionality makes use of the `sproxy` and `sproxy_async` clients added in release 2020.2.0 through the `salt-sapi` entry point. See https://salt-sproxy.readthedocs.io/en/latest/salt_api.html and https://salt-sproxy.readthedocs.io/en/latest/salt_sapi.html for more details.

Important: In the configuration examples below, for simplicity, I've used the auto external authentication, and disabled the SSL for the Salt API. This setup is highly discouraged in production.

Using the Master configuration file under examples/salt_sapi/master:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar

file_roots:
  base:
    - /srv/salt/extmods

rest_cherrypy:
  port: 8080
  disable_ssl: true

external_auth:
  auto:
    '*':
      - '@runner'
```

The `pillar_roots` option points to `/srv/salt/pillar`, so to be able to use this example, either create a symlink to the `pillar` directory in this example, or copy the files. For example, if you just cloned this repository:

```
$ mkdir -p /srv/salt/pillar
$ git clone git@github.com:mirceaulinic/salt-sproxy.git
$ cp salt-sproxy/examples/salt_sapi/master /etc/salt/master
$ cp salt-sproxy/examples/salt_sapi/pillar/*.sls /srv/salt/pillar/
```

The contents of Pillar files:

`/srv/salt/pillar/top.sls`:

```
base:
  mininon1:
    - dummy
  juniper-router:
    - junos
```

`/srv/salt/pillar/dummy.sls`:

```
proxy:
  proxytype: dummy
```

`/srv/salt/pillar/junos.sls`:

```
proxy:
  proxytype: napalm
  driver: junos
  host: juniper.salt-sproxy.digitalocean.cloud.tesuto.com
  username: test
  password: t35t1234
```

Note: The `top.sls`, `dummy.sls`, and `junos.sls` are a combination of the previous examples, 101 and napalm, which is going to allow use to execute against both the dummy device and a real network device.

In the example Master configuration file above, there's also a section for the `file_roots`. As documented in The Proxy Runner section of the documentation, you are going to reference the proxy Runner, e.g.

```
$ mkdir -p /srv/salt/extmods/_runners
$ cp salt-sproxy/salt_sproxy/_runners/proxy.py /srv/salt/extmods/_runners/
```

Or symlink:

```
$ ln -s /path/to/git/clone/salt-sproxy/salt_sproxy /srv/salt/extmods
```

With the `rest_cherrypy` section, the Salt API will be listening to HTTP requests over port 8080, and SSL being disabled (not recommended in production):

```
rest_cherrypy:
  port: 8080
  disable_ssl: true
```

One another part of the configuration is the external authentication:

```
external_auth:
  auto:
    '*':
      - '@runner'
```

This grants access to anyone to execute any Runner (again, don't do this in production).

With this setup, we can start the Salt Master and the Salt API (running in background):

```
$ salt-master -d
$ salt-sapi -d
```

To verify that the REST API is ready, execute:

```
$ curl -i localhost:8080
HTTP/1.1 200 OK
Content-Type: application/json
Server: CherryPy/18.1.1
Date: Wed, 01 Jan 2020 07:58:32 GMT
Allow: GET, HEAD, POST
Access-Control-Allow-Origin: *
```

(continues on next page)

(continued from previous page)

```
Access-Control-Expose-Headers: GET, POST
Access-Control-Allow-Credentials: true
Vary: Accept-Encoding
Content-Length: 146

{"return": "Welcome", "clients": ["local", "local_async", "local_batch", "local_subset
↪", "runner", "runner_async", "sproxy", "sproxy_async", "ssh", "wheel", "wheel_async
↪"]}
```

Now we can go ahead and execute the CLI command from example 101, by making an HTTP request:

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='auto' \
  -d username='mircea' \
  -d password='pass' \
  -d client='sproxy' \
  -d tgt='minion1' \
  -d fun='test.ping'
return:
- minion1: true
```

Notice that `eauth` field in this case is `auto` as this is what we've configured in the `external_auth` on the Master.

Similarly, you can now execute the Salt functions from the NAPALM example, against a network device, by making an HTTP request:

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='auto' \
  -d username='mircea' \
  -d password='pass' \
  -d client='sproxy' \
  -d tgt='juniper-router' \
  -d fun='net.arp'
return:
- juniper-router:
    comment: ''
    out:
    - age: 891.0
      interface: fxp0.0
      ip: 10.96.0.1
      mac: 92:99:00:0A:00:00
    - age: 1001.0
      interface: fxp0.0
      ip: 10.96.0.13
      mac: 92:99:00:0A:00:00
    - age: 902.0
      interface: em1.0
      ip: 128.0.0.16
      mac: 02:42:AC:12:00:02
    result: true
```

CHAPTER 6

Extension Modules

salt-sproxy is delivered together with a few extension modules that are dynamically loaded and immediately available. Please see below the documentation for these modules:

6.1 Extension Roster Modules

6.1.1 Ansible Roster

Read in an Ansible inventory file or script

Flat inventory files should be in the regular ansible inventory format.

```
[servers]
salt.gtmanfred.com ansible_ssh_user=gtmanfred ansible_ssh_host=127.0.0.1 ansible_ssh_
↪port=22 ansible_ssh_pass='password'

[desktop]
home ansible_ssh_user=gtmanfred ansible_ssh_host=12.34.56.78 ansible_ssh_port=23␣
↪ansible_ssh_pass='password'

[computers:children]
desktop
servers

[names:vars]
http_port=80
```

then salt-ssh can be used to hit any of them

```
[~]# salt-ssh -N all test.ping
salt.gtmanfred.com:
    True
home:
```

(continues on next page)

(continued from previous page)

```
    True
[~]# salt-ssh -N desktop test.ping
home:
    True
[~]# salt-ssh -N computers test.ping
salt.gtmanfred.com:
    True
home:
    True
[~]# salt-ssh salt.gtmanfred.com test.ping
salt.gtmanfred.com:
    True
```

There is also the option of specifying a dynamic inventory, and generating it on the fly

```
#!/bin/bash
echo '{
  "servers": [
    "salt.gtmanfred.com"
  ],
  "desktop": [
    "home"
  ],
  "computers": {
    "hosts": [],
    "children": [
      "desktop",
      "servers"
    ]
  },
  "_meta": {
    "hostvars": {
      "salt.gtmanfred.com": {
        "ansible_ssh_user": "gtmanfred",
        "ansible_ssh_host": "127.0.0.1",
        "ansible_sudo_pass": "password",
        "ansible_ssh_port": 22
      },
      "home": {
        "ansible_ssh_user": "gtmanfred",
        "ansible_ssh_host": "12.34.56.78",
        "ansible_sudo_pass": "password",
        "ansible_ssh_port": 23
      }
    }
  }
}'
```

This is the format that an inventory script needs to output to work with ansible, and thus here.

```
[~]# salt-ssh --roster-file /etc/salt/hosts salt.gtmanfred.com test.ping
salt.gtmanfred.com:
        True
```

Any of the [groups] or direct hostnames will return. The 'all' is special, and returns everything.

`_roster.ansible.`**`targets`**(*tgt*, *tgt_type='glob'*, ***kwargs*)

Return the targets from the ansible inventory_file Default: /etc/salt/roster

6.1.2 File Roster

Load the list of devices from an arbitrary SLS file.

To use this module, you only need to configure the –roster option to `file` (on the CLI or Master config), and if the Roster SLS file is in a different location than `/etc/salt/roster`, you'd also need to specify `--roster-file` (or `roster_file` in the Master config).

`_roster.file.`**`targets`**(*tgt*, *tgt_type='glob'*, ***kwargs*)

Return the targets from the sls file, checks opts for location but defaults to /etc/salt/roster

6.1.3 NetBox Roster

Load devices from NetBox, and make them available for salt-ssh or salt-sproxy (or any other program that doesn't require (Proxy) Minions running).

Make sure that the following options are configured on the Master:

```
netbox:
  url: <NETBOX_URL>
  token: <NETBOX_USERNAME_API_TOKEN (OPTIONAL)>
  keyfile: </PATH/TO/NETBOX/KEY (OPTIONAL)>
```

If you want to pre-filter the devices, so it won't try to pull the whole database available in NetBox, you can configure another key, `filters`, under `netbox`, e.g.,

```
netbox:
  url: <NETBOX_URL>
  filters:
    site: <SITE>
    status: <STATUS>
```

Hint: You can use any NetBox field as a filter.

Important: In NetBox v2.6 the default view permissions changed, so `salt-sproxy` may not able to get the device list from NetBox by default.

Add `EXEMPT_VIEW_PERMISSIONS = ['*']` to the `configuration.py` NetBox file to change this behavior. See https://github.com/netbox-community/netbox/releases/tag/v2.6.0 for more information

`_roster.netbox.`**`targets`**(*tgt*, *tgt_type='glob'*, ***kwargs*)

Return the targets from NetBox.

6.1.4 Pillar Roster

Load the list of devices from the Pillar.

Simply configure the `roster` option to point to this module, while making sure that the data is available. As the Pillar is data associated with a specific Minion ID, you may need to ensure that the Pillar is correctly associated with the Minion configured (default `*`), under the exact key required (default `devices`). To adjust these options, you can provide the following under the `roster_pillar` option in the Master configuration:

minion_id: * The ID of the Minion to compile the data for. Default: `*` (any Minion).

pillar_key: `devices` The Pillar field to pull the list of devices from. Default: `devices`.

saltenv: `base` The Salt environment to use when compiling the Pillar data.

pillarenv The Pillar environment to use when compiling the Pillar data.

Configuration example:

```
roster: pillar
roster_pillar:
  minion_id: sproxy
  pillar_key: minions
```

With the following configuration, when executing `salt-run pillar.show_pillar sproxy` you should have under `minions` the list of devices / Minions you want to manage.

Hint: The Pillar data can either be provided as files, or using one or more External Pillars. Check out https://docs.saltstack.com/en/latest/ref/pillar/all/index.html for the complete list of available Pillar modules you can use.

`_roster.pillar.`**`targets`**(*tgt*, *tgt_type='glob'*, ***kwargs*)
Return the targets from External Pillar requested.

6.2 Extension Runners

6.2.1 Proxy Runner

Salt Runner to invoke arbitrary commands on network devices that are not managed via a Proxy or regular Minion. Therefore, this Runner doesn't necessarily require the targets to be up and running, as it will connect to collect the Grains, compile the Pillar, then execute the commands.

class `_runners.proxy.`**`SProxyMinion`**(*opts*, *context=None*)
Create an object that has loaded all of the minion module functions, grains, modules, returners etc. The SProxyMinion allows developers to generate all of the salt minion functions and present them with these functions for general use.

`gen_modules`(*initial_load=False*)
Tell the minion to reload the execution modules.

CLI Example:

```
salt '*' sys.reload_modules
```

class `_runners.proxy.`**`StandaloneProxy`**(*opts*, *unreachable_devices=None*) `_runners.proxy.`**`execute`**(*tgt*, *function=None*, *tgt_type='glob'*, *roster=None*, *preview_target=False*, *target_details=False*, *timeout=60*, *with_grains=True*, *with_pillar=True*, *preload_grains=True*, *preload_pillar=True*, *default_grains=None*, *default_pillar=None*, *args=()*, *batch_size=10*, *batch_wait=0*, *static=False*, *events=True*, *cache_grains=False*, *cache_pillar=False*, *use_cached_grains=True*, *use_cached_pillar=True*, *use_existing_proxy=False*, *no_connect=False*, *test_ping=False*, *target_cache=True*, *target_cache_timeout=60*, *preload_targeting=False*, *invasive_targeting=False*, *failhard=False*, *summary=True*, *verbose=False*, *show_jid=False*, *progress=False*, *hide_timeout=False*, *saltenv='base'*, *sync_roster=False*, *sync_modules=False*, *sync_grains=False*, *sync_all=False*, *returner=''*, *returner_config=''*, *returner_kwargs={}*, ***kwargs*)

Invoke a Salt function on the list of devices matched by the Roster subsystem.

tgt The target expression, e.g., `*` for all devices, or `host1,host2` for a list, etc. The `tgt_list` argument must be used accordingly, depending on the type of this expression.

function The name of the Salt function to invoke.

tgt_type: `glob` The type of the `tgt` expression. Choose between: `glob` (default), `list`, `pcre`, `rage`, or `nodegroup`.

roster: `None` The name of the Roster to generate the targets. Alternatively, you can specify the name of the Roster by configuring the `proxy_roster` option into the Master config.

preview_target: `False` Return the list of Roster targets matched by the `tgt` and `tgt_type` arguments.

preload_grains: `True` Whether to preload the Grains before establishing the connection with the remote network device.

default_grains: Dictionary of the default Grains to make available within the functions loaded.

with_grains: `True` Whether to load the Grains modules and collect Grains data and make it available inside the Execution Functions. The Grains will be loaded after opening the connection with the remote network device.

default_pillar: Dictionary of the default Pillar data to make it available within the functions loaded.

with_pillar: `True` Whether to load the Pillar modules and compile Pillar data and make it available inside the Execution Functions.

arg The list of arguments to send to the Salt function.

kwargs Key-value arguments to send to the Salt function.

batch_size: `10` The size of each batch to execute.

static: `False` Whether to return the results synchronously (or return them as soon as the device replies).

events: `True` Whether should push events on the Salt bus, similar to when executing equivalent through the `salt` command.

use_cached_pillar: `True` Use cached Pillars whenever possible. If unable to gather cached data, it falls back to compiling the Pillar.

use_cached_grains: `True` Use cached Grains whenever possible. If unable to gather cached data, it falls back to collecting Grains.

cache_pillar: `False` Cache the compiled Pillar data before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

cache_grains: `False` Cache the collected Grains before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

use_existing_proxy: `False` Use the existing Proxy Minions when they are available (say on an already running Master).

no_connect: `False` Don't attempt to initiate the connection with the remote device. Default: `False` (it will initiate the connection).

test_ping: `False` When using the existing Proxy Minion with the `use_existing_proxy` option, can use this argument to verify also if the Minion is responsive.

target_cache: `True` Whether to use the cached target matching results.

target_cache_timeout: 60 The duration to cache the target results for (in seconds).

CLI Example:

```
salt-run proxy.execute_roster edge* test.ping
salt-run proxy.execute_roster junos-edges test.ping tgt_type=nodegroup
```

`_runners.proxy.`**`execute_devices`**(*minions*, *function*, *with_grains=True*, *with_pillar=True*, *preload_grains=True*, *preload_pillar=True*, *default_grains=None*, *default_pillar=None*, *args=()*, *batch_size=10*, *batch_wait=0*, *static=False*, *tgt=None*, *tgt_type=None*, *jid=None*, *events=True*, *cache_grains=False*, *cache_pillar=False*, *use_cached_grains=True*, *use_cached_pillar=True*, *use_existing_proxy=False*, *no_connect=False*, *roster_targets=None*, *test_ping=False*, *preload_targeting=False*, *invasive_targeting=False*, *failhard=False*, *timeout=60*, *summary=False*, *verbose=False*, *progress=False*, *hide_timeout=False*, *returner="*, *returner_config="*, *returner_kwargs={}*, ***kwargs*)

Execute a Salt function on a group of network devices identified by their Minion ID, as listed under the `minions` argument.

minions A list of Minion IDs to invoke `function` on.

function The name of the Salt function to invoke.

preload_grains: `True` Whether to preload the Grains before establishing the connection with the remote network device.

default_grains: Dictionary of the default Grains to make available within the functions loaded.

with_grains: `False` Whether to load the Grains modules and collect Grains data and make it available inside the Execution Functions. The Grains will be loaded after opening the connection with the remote network device.

preload_pillar: `True` Whether to preload Pillar data before opening the connection with the remote network device.

default_pillar: Dictionary of the default Pillar data to make it available within the functions loaded.

with_pillar: `True` Whether to load the Pillar modules and compile Pillar data and make it available inside the Execution Functions.

args The list of arguments to send to the Salt function.

kwargs Key-value arguments to send to the Salt function.

batch_size: `10` The size of each batch to execute.

static: `False` Whether to return the results synchronously (or return them as soon as the device replies).

events: `True` Whether should push events on the Salt bus, similar to when executing equivalent through the `salt` command.

use_cached_pillar: `True` Use cached Pillars whenever possible. If unable to gather cached data, it falls back to compiling the Pillar.

use_cached_grains: `True` Use cached Grains whenever possible. If unable to gather cached data, it falls back to collecting Grains.

cache_pillar: `False` Cache the compiled Pillar data before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

cache_grains: `False` Cache the collected Grains before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

use_existing_proxy: `False` Use the existing Proxy Minions when they are available (say on an already running Master).

no_connect: `False` Don't attempt to initiate the connection with the remote device. Default: `False` (it will initiate the connection).

test_ping: `False` When using the existing Proxy Minion with the `use_existing_proxy` option, can use this argument to verify also if the Minion is responsive.

CLI Example:

```
salt-run proxy.execute "['172.17.17.1', '172.17.17.2']" test.ping driver=eos␣
↪username=test password=test123
```

`_runners.proxy.`**`salt_call`**(*minion_id*, *function=None*, *unreachable_devices=None*, *failed_devices=None*, *with_grains=True*, *with_pillar=True*, *preload_grains=True*, *preload_pillar=True*, *default_grains=None*, *default_pillar=None*, *cache_grains=False*, *cache_pillar=False*, *use_cached_grains=True*, *use_cached_pillar=True*, *use_existing_proxy=False*, *no_connect=False*, *jid=None*, *roster_opts=None*, *test_ping=False*, *tgt=None*, *tgt_type=None*, *preload_targeting=False*, *invasive_targeting=False*, *failhard=False*, *timeout=60*, *returner=''*, *returner_config=''*, *returner_kwargs={}*, *args=()*, ***kwargs*)

Invoke a Salt Execution Function that requires or invokes an NAPALM functionality (directly or indirectly).

minion_id: The ID of the Minion to compile Pillar data for.

function The name of the Salt function to invoke.

preload_grains: `True` Whether to preload the Grains before establishing the connection with the remote network device.

default_grains: Dictionary of the default Grains to make available within the functions loaded.

with_grains: `True` Whether to load the Grains modules and collect Grains data and make it available inside the Execution Functions. The Grains will be loaded after opening the connection with the remote network device.

preload_pillar: `True` Whether to preload Pillar data before opening the connection with the remote network device.

default_pillar: Dictionary of the default Pillar data to make it available within the functions loaded.

with_pillar: `True` Whether to load the Pillar modules and compile Pillar data and make it available inside the Execution Functions.

use_cached_pillar: `True` Use cached Pillars whenever possible. If unable to gather cached data, it falls back to compiling the Pillar.

use_cached_grains: `True` Use cached Grains whenever possible. If unable to gather cached data, it falls back to collecting Grains.

cache_pillar: `False` Cache the compiled Pillar data before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

cache_grains: `False` Cache the collected Grains before returning.

> **Warning:** This option may be dangerous when targeting a device that already has a Proxy Minion associated, however recommended otherwise.

use_existing_proxy: `False` Use the existing Proxy Minions when they are available (say on an already running Master).

no_connect: `False` Don't attempt to initiate the connection with the remote device. Default: `False` (it will initiate the connection).

jid: `None` The JID to pass on, when executing.

test_ping: `False` When using the existing Proxy Minion with the `use_existing_proxy` option, can use this argument to verify also if the Minion is responsive.

arg The list of arguments to send to the Salt function.

kwargs Key-value arguments to send to the Salt function.

CLI Example:

```
salt-run proxy.salt_call bgp.neighbors junos 1.2.3.4 test test123
salt-run proxy.salt_call net.load_config junos 1.2.3.4 test test123 text='set␣
↪system ntp peer 1.2.3.4'
```

6.3 Execution Modules

6.3.1 NetBox Execution Module

NetBox

Module to query NetBox

codeauthor Zach Moody <zmoody@do.co>

maturity new

depends pynetbox

Note: This code, distributed as part of `salt-sproxy`, has been copied from the main Salt project, maintained by SaltStack, to provide various enhancements and fixes to the original module.

The following config should be in the minion config file. In order to work with `secrets` you should provide a token and path to your private key file:

```
netbox:
  url: <NETBOX_URL>
  token: <NETBOX_USERNAME_API_TOKEN (OPTIONAL)>
  keyfile: </PATH/TO/NETBOX/KEY (OPTIONAL)>
```

New in version 2018.3.0: This module has been introduced in Salt release 2018.3.0.

In `salt-sproxy`, this module has been included beginning with version 2019.10.0.

`_modules.netbox.`**`create_circuit`**(*name*, *provider_id*, *circuit_type*, *description=None*)

New in version 2019.2.0.

Create a new Netbox circuit

name Name of the circuit

provider_id The netbox id of the circuit provider

circuit_type The name of the circuit type

asn The ASN of the circuit provider

description The description of the circuit

CLI Example:

```
salt myminion netbox.create_circuit NEW_CIRCUIT_01 Telia Transit 1299 "New Telia␣
↪circuit"
```

`_modules.netbox.`**`create_circuit_provider`**(*name*, *asn=None*)

New in version 2019.2.0.

Create a new Netbox circuit provider

name The name of the circuit provider

asn The ASN of the circuit provider

CLI Example:

```
salt myminion netbox.create_circuit_provider Telia 1299
```

_modules.netbox.**create_circuit_termination**(*circuit*, *interface*, *device*, *speed*, *xconnect_id=None*, *term_side='A'*)

New in version 2019.2.0.

Terminate a circuit on an interface

circuit The name of the circuit

interface The name of the interface to terminate on

device The name of the device the interface belongs to

speed The speed of the circuit, in Kbps

xconnect_id The cross-connect identifier

term_side The side of the circuit termination

CLI Example:

```
salt myminion netbox.create_circuit_termination NEW_CIRCUIT_01 xe-0/0/1 myminion␣
↪10000 xconnect_id=XCON01
```

_modules.netbox.**create_circuit_type**(*name*)

New in version 2019.2.0.

Create a new Netbox circuit type.

name The name of the circuit type

CLI Example:

```
salt myminion netbox.create_circuit_type Transit
```

_modules.netbox.**create_device**(*name*, *role*, *model*, *manufacturer*, *site*)

New in version 2019.2.0.

Create a new device with a name, role, model, manufacturer and site. All these components need to be already in Netbox.

name The name of the device, e.g., `edge_router`

role String of device role, e.g., `router`

model String of device model, e.g., `MX480`

manufacturer String of device manufacturer, e.g., `Juniper`

site String of device site, e.g., `BRU`

CLI Example:

```
salt myminion netbox.create_device edge_router router MX480 Juniper BRU
```

_modules.netbox.**create_device_role**(*role*, *color*)

New in version 2019.2.0.

Create a device role

role String of device role, e.g., `router`

CLI Example:

```
salt myminion netbox.create_device_role router
```

_modules.netbox.**create_device_type**(*model*, *manufacturer*)

New in version 2019.2.0.

Create a device type. If the manufacturer doesn't exist, create a new manufacturer.

model String of device model, e.g., `MX480`

manufacturer String of device manufacturer, e.g., `Juniper`

CLI Example:

```
salt myminion netbox.create_device_type MX480 Juniper
```

_modules.netbox.**create_interface**(*device_name*, *interface_name*, *mac_address=None*, *description=None*, *enabled=None*, *lag=None*, *lag_parent=None*, *form_factor=None*)

New in version 2019.2.0.

Attach an interface to a device. If not all arguments are provided, they will default to Netbox defaults.

device_name The name of the device, e.g., `edge_router`

interface_name The name of the interface, e.g., `TenGigE0/0/0/0`

mac_address String of mac address, e.g., `50:87:89:73:92:C8`

description String of interface description, e.g., `NTT`

enabled String of boolean interface status, e.g., `True`

lag: Boolean of interface lag status, e.g., `True`

lag_parent String of interface lag parent name, e.g., `ae13`

form_factor Integer of form factor id, obtained through _choices API endpoint, e.g., `200`

CLI Example:

```
salt myminion netbox.create_interface edge_router ae13 description="Core uplink"
```

_modules.netbox.**create_interface_connection**(*interface_a*, *interface_b*)

New in version 2019.2.0.

Create an interface connection between 2 interfaces

interface_a Interface id for Side A

interface_b Interface id for Side B

CLI Example:

```
salt myminion netbox.create_interface_connection 123 456
```

_modules.netbox.**create_inventory_item**(*device_name*, *item_name*, *manufacturer_name=None*, *serial="*, *part_id="*, *description="*)

New in version 2019.2.0.

Add an inventory item to an existing device.

device_name The name of the device, e.g., `edge_router`.

item_name String of inventory item name, e.g., `Transceiver`.

manufacturer_name String of inventory item manufacturer, e.g., `Fiberstore`.

serial String of inventory item serial, e.g., `FS1238931`.

part_id String of inventory item part id, e.g., `740-01234`.

description String of inventory item description, e.g., `SFP+-10G-LR`.

CLI Example:

```
salt myminion netbox.create_inventory_item edge_router Transceiver part_id=740-
↪01234
```

_modules.netbox.**create_ipaddress**(*ip_address*, *family*, *device=None*, *interface=None*)
New in version 2019.2.0.

Add an IP address, and optionally attach it to an interface.

ip_address The IP address and CIDR, e.g., `192.168.1.1/24`

family Integer of IP family, e.g., `4`

device The name of the device to attach IP to, e.g., `edge_router`

interface The name of the interface to attach IP to, e.g., `ae13`

CLI Example:

```
salt myminion netbox.create_ipaddress 192.168.1.1/24 4 device=edge_router␣
↪interface=ae13
```

_modules.netbox.**create_manufacturer**(*name*)
New in version 2019.2.0.

Create a device manufacturer.

name The name of the manufacturer, e.g., `Juniper`

CLI Example:

```
salt myminion netbox.create_manufacturer Juniper
```

_modules.netbox.**create_platform**(*platform*)
New in version 2019.2.0.

Create a new device platform

platform String of device platform, e.g., `junos`

CLI Example:

```
salt myminion netbox.create_platform junos
```

_modules.netbox.**create_site**(*site*)
New in version 2019.2.0.

Create a new device site

site String of device site, e.g., `BRU`

CLI Example:

```
salt myminion netbox.create_site BRU
```

_modules.netbox.**delete_interface**(*device_name*, *interface_name*)
New in version 2019.2.0.

Delete an interface from a device.

device_name The name of the device, e.g., `edge_router`.

interface_name The name of the interface, e.g., `ae13`

CLI Example:

```
salt myminion netbox.delete_interface edge_router ae13
```

`_modules.netbox.`**`delete_inventory_item`**`(`*item_id*`)`

New in version 2019.2.0.

Remove an item from a devices inventory. Identified by the netbox id

item_id Integer of item to be deleted

CLI Example:

```
salt myminion netbox.delete_inventory_item 1354
```

`_modules.netbox.`**`delete_ipaddress`**`(`*ipaddr_id*`)`

New in version 2019.2.0.

Delete an IP address. IP addresses in Netbox are a combination of address and the interface it is assigned to.

id The Netbox id for the IP address.

CLI Example:

```
salt myminion netbox.delete_ipaddress 9002
```

`_modules.netbox.`**`filter_`**`(`*app*, *endpoint*, ***kwargs*`)`

Get a list of items from NetBox.

app String of netbox app, e.g., `dcim`, `circuits`, `ipam`

endpoint String of app endpoint, e.g., `sites`, `regions`, `devices`

kwargs Optional arguments that can be used to filter. All filter keywords are available in Netbox, which can be found by surfing to the corresponding API endpoint, and clicking Filters. e.g., `role=router`

Returns a list of dictionaries

```
salt myminion netbox.filter dcim devices status=1 role=router
```

`_modules.netbox.`**`get_`**`(`*app*, *endpoint*, *id=None*, ***kwargs*`)`

Get a single item from NetBox.

app String of netbox app, e.g., `dcim`, `circuits`, `ipam`

endpoint String of app endpoint, e.g., `sites`, `regions`, `devices`

Returns a single dictionary

To get an item based on ID.

```
salt myminion netbox.get dcim devices id=123
```

Or using named arguments that correspond with accepted filters on the NetBox endpoint.

```
salt myminion netbox.get dcim devices name=my-router
```

_modules.netbox.**get_circuit_provider**(*name*, *asn=None*)

New in version 2019.2.0.

Get a circuit provider with a given name and optional ASN.

name The name of the circuit provider

asn The ASN of the circuit provider

CLI Example:

```
salt myminion netbox.get_circuit_provider Telia 1299
```

_modules.netbox.**get_interfaces**(*device_name=None*, ***kwargs*)

New in version 2019.2.0.

Returns interfaces for a specific device using arbitrary netbox filters

device_name The name of the device, e.g., `edge_router`

kwargs Optional arguments to be used for filtering

CLI Example:

```
salt myminion netbox.get_interfaces edge_router name="et-0/0/5"
```

_modules.netbox.**get_ipaddresses**(*device_name=None*, ***kwargs*)

New in version 2019.2.0.

Filters for an IP address using specified filters

device_name The name of the device to check for the IP address

kwargs Optional arguments that can be used to filter, e.g., `family=4`

CLI Example:

```
salt myminion netbox.get_ipaddresses device_name family=4
```

_modules.netbox.**make_interface_child**(*device_name*, *interface_name*, *parent_name*)

New in version 2019.2.0.

Set an interface as part of a LAG.

device_name The name of the device, e.g., `edge_router`.

interface_name The name of the interface to be attached to LAG, e.g., `xe-1/0/2`.

parent_name The name of the LAG interface, e.g., `ae13`.

CLI Example:

```
salt myminion netbox.make_interface_child xe-1/0/2 ae13
```

_modules.netbox.**make_interface_lag**(*device_name*, *interface_name*)

New in version 2019.2.0.

Update an interface to be a LAG.

device_name The name of the device, e.g., `edge_router`.

interface_name The name of the interface, e.g., `ae13`.

CLI Example:

```
salt myminion netbox.make_interface_lag edge_router ae13
```

_modules.netbox.**openconfig_interfaces**(*device_name=None*)
New in version 2019.2.0.

Return a dictionary structured as standardised in the openconfig-interfaces YANG model, containing physical and configuration data available in Netbox, e.g., IP addresses, MTU, enabled / disabled, etc.

device_name: `None` The name of the device to query the interface data for. If not provided, will use the Minion ID.

CLI Example:

```
salt '*' netbox.openconfig_interfaces
salt '*' netbox.openconfig_interfaces device_name=cr1.thn.lon
```

_modules.netbox.**openconfig_lacp**(*device_name=None*)
New in version 2019.2.0.

Return a dictionary structured as standardised in the openconfig-lacp YANG model, with configuration data for Link Aggregation Control Protocol (LACP) for aggregate interfaces.

Note: The `interval` and `lacp_mode` keys have the values set as `SLOW` and `ACTIVE` respectively, as this data is not currently available in Netbox, therefore defaulting to the values defined in the standard. See interval and lacp-mode for further details.

device_name: `None` The name of the device to query the LACP information for. If not provided, will use the Minion ID.

CLI Example:

```
salt '*' netbox.openconfig_lacp
salt '*' netbox.openconfig_lacp device_name=cr1.thn.lon
```

_modules.netbox.**slugify**(*value*)
' Slugify given value. Credit to Djangoproject https://docs.djangoproject.com/en/2.0/_modules/django/utils/text/#slugify

_modules.netbox.**update_device**(*name*, ***kwargs*)
New in version 2019.2.0.

Add attributes to an existing device, identified by name.

name The name of the device, e.g., `edge_router`

kwargs Arguments to change in device, e.g., `serial=JN2932930`

CLI Example:

```
salt myminion netbox.update_device edge_router serial=JN2932920
```

_modules.netbox.**update_interface**(*device_name*, *interface_name*, ***kwargs*)
New in version 2019.2.0.

Update an existing interface with new attributes.

device_name The name of the device, e.g., `edge_router`

interface_name The name of the interface, e.g., `ae13`

kwargs Arguments to change in interface, e.g., `mac_address=50:87:69:53:32:D0`

CLI Example:

```
salt myminion netbox.update_interface edge_router ae13 mac_
↪address=50:87:69:53:32:D0
```

CHAPTER 7

See Also

7.1 Quick Start

This is a configuration example to quickly get started with `salt-sproxy`.

7.1.1 1. Install `salt-sproxy`

Run `pip install salt-sproxy` either at root, or within a virtual environment.

If you don't know how to install `pip`, see this document: https://pip.pypa.io/en/stable/installing/.

For setting up a virtual environment, check out https://virtualenv.pypa.io/en/stable/installation/.

If you have more specific requirements for the `salt-sproxy` installation, see *Installation*.

7.1.2 2. Build the list of devices

Say you have a list of devices you want to manage. For ease, you can put them into a file:

`/etc/salt/roster`

```
router1:
  driver: junos
router2:
  driver: iosxr
switch1:
  driver: eos
fw1:
  driver: panos
  host: fw1.firewall.as1234.net
```

Note: The `/etc/salt/roster` file can use any of the available SLS formats (combinations of the Salt Renderer modules) - Jinja + YAML, YAML, JSON, pure Python, JSON5, HJSON, etc.

For more examples, see also *Using the File Roster*.

7.1.3 3. Configure

Apply the following configuration:

/etc/salt/master

```
roster: file
```

This is all you need at minimum, however, you may have more specific requirements which you can customise using the configuration options documented in https://docs.saltstack.com/en/latest/ref/configuration/master.html.

7.1.4 4. Prepare the connection credentials

In a file, say /srv/pillar/proxy.sls, you'll need the following structure:

```
proxy:
  proxytype: <proxy type>
  username: <username>
  password: <password>
  host: <host>
```

Where proxy type is the name of one of the available Proxy modules, either Salt native (https://docs.saltstack.com/en/latest/ref/proxy/all/index.html), or developed in your own environment.

Note: Either of these fields (i.e., proxytype, username, password, host) can be specified in the list of devices in the Pillar above (step 2). Generally, in this file, you put the list of parameters that are globally available to any devices. For example, if you're using the same username to manage all devices, you don't need to put it in the Pillar defined at *step 2*, but rather set it here.

Example:

```
proxy:
  proxytype: napalm
  username: salt
  password: SaltSPr0xyRocks!
  host: {{ opts.id }}.as1234.net
```

The trick in the SLS above is the host field, which is rendered differently for each device; for instance, the hostname for the device router1 would be router1.as1234.net, and so on. As an exception, at *step 2*, for fw2 we defined a most specific host field, so salt-sproxy is going to use that one instead.

In the same way you can build custom dynamically rendered fields, as your business logic requires, making use of the flexibility of the SLS file format (which is by default Jinja + YAML, see this for more information).

Tip: If you want to use your own username / SSH key for authentication, you can configure the following:

```
username: {{ salt.environ.get('USER') }}
```

The configuration above, would dynamically use the username currently logged in, which could be particularly useful for shared environments where multiple users (with potentially different access levels) can log in and run Salt commands.

To authenticate using your SSH key, you need to set the `password` field blank / empty string (i.e., `password: ''`).

As for using a custom private SSH key, you should check the documentation of the Proxy module of choice. For example, if you're using NAPALM, the location of the SSH key would be configured under the `optional_args` key, e.g.,

```
proxy:
  proxytype: napalm
  username: {{ salt.environ.get('USER') }}
  password: ''
  host: {{ opts.id }}.as1234.net
  optional_args:
    key_file: /path/to/priv/key
```

Granted you have the structure above in the `/srv/pillar/proxy.sls` file, as a last step, you only need to include it into the Pillar top file:

`/srv/pillar/top.sls`

```
base:
  '*':
    - proxy
```

7.1.5 5. Happy automating!

With these three files (`/etc/salt/roster`, `/etc/salt/master`, and `/srv/pillar/proxy.sls`) configured as described, you can now start automating your network, e.g.,

```
$ salt-sproxy router1 net.arp
# ... snip ...

$ salt-sproxy -L router1,router2 net.load_config \
    text='set system ntp server 10.10.10.1'
# ... snip ...

$ salt-sproxy router2 napalm.junos_rpc 'get-validation-statistics'
# ... snip ...

$ salt-sproxy \* net.cli 'request system zeroize'
```

7.2 Installation

The base installation is pretty much straightforward, `salt-sproxy` is installable using `pip`. See https://packaging.python.org/tutorials/installing-packages/ for a comprehensive guide on the installing Python packages.

Either when installing in a virtual environment, or directly on the base system, execute the following:

```
$ pip install salt-sproxy
```

If you would like to install a specific Salt version, you will firstly need to instal Salt (via pip) pinning to the desired version, e.g.,

```
$ pip install salt==2018.3.4
$ pip install salt-sproxy
```

7.2.1 Easy installation

We also provide a script to install the system requirements: https://raw.githubusercontent.com/mirceaulinic/salt-sproxy/master/install.sh

Usage example:

- Using curl

```
$ curl sproxy-install.sh -L https://raw.githubusercontent.com/mirceaulinic/salt-
↪sproxy/master/install.sh
# check the contents of sproxy-install.sh
$ sudo sh sproxy-install.sh
```

- Using wget

```
$ wget -O sproxy-install.sh https://raw.githubusercontent.com/mirceaulinic/salt-
↪sproxy/master/install.sh
# check the contents of sproxy-install.sh
$ sudo sh sproxy-install.sh
```

- Using fetch (on FreeBSD)

```
$ fetch -o sproxy-install.sh https://raw.githubusercontent.com/mirceaulinic/salt-
↪sproxy/master/install.sh
# check the contents of sproxy-install.sh
$ sudo sh sproxy-install.sh
```

One liner:

Warning: This method can be dangerous and it is not recommended on production systems.

```
$ curl -L https://raw.githubusercontent.com/mirceaulinic/salt-sproxy/master/install.
↪sh | sudo sh
```

See https://gist.github.com/mirceaulinic/bdbbbcfbc3588b1c8b1ec7ef63931ac6 for a sample one-line installation on a fresh Fedora server.

The script ensures Python 3 is installed on your system, together with the virtualenv package, and others required for Salt, in a virtual environment under the `$HOME/venvs/salt-sproxy` path. In fact, when executing, you will see that the script will tell where it's going to try to install, e.g.,

```
$ sudo sh install.sh

Installing salt-sproxy under /home/mircea/venvs/salt-sproxy

Reading package lists... Done

~~~ snip ~~~

Installation complete, now you can start using by executing the following command:
. /home/mircea/venvs/salt-sproxy/bin/activate
```

After that, you can start using it:

```
$ . /home/mircea/venvs/salt-sproxy/bin/activate
(salt-sproxy) $
(salt-sproxy) $ salt-sproxy -V
Salt Version:
           Salt: 2019.2.0
    Salt SProxy: 2019.6.0b1

Dependency Versions:
        Ansible: Not Installed
           cffi: 1.12.3
       dateutil: Not Installed
      docker-py: Not Installed
          gitdb: Not Installed
      gitpython: Not Installed
         Jinja2: 2.10.1
     junos-eznc: 2.2.1
       jxmlease: 1.0.1
        libgit2: Not Installed
       M2Crypto: Not Installed
           Mako: Not Installed
   msgpack-pure: Not Installed
 msgpack-python: 0.6.1
         NAPALM: 2.4.0
       ncclient: 0.6.4
        Netmiko: 2.3.3
       paramiko: 2.4.2
      pycparser: 2.19
       pycrypto: 2.6.1
   pycryptodome: Not Installed
         pyeapi: 0.8.2
         pygit2: Not Installed
       PyNetBox: 4.0.6
          PyNSO: Not Installed
         Python: 3.6.7 (default, Oct 22 2018, 11:32:17)
   python-gnupg: Not Installed
         PyYAML: 5.1
          PyZMQ: 18.0.1
            scp: 0.13.2
          smmap: Not Installed
        textfsm: 0.4.1
        timelib: Not Installed
        Tornado: 4.5.3
            ZMQ: 4.3.1

System Versions:
           dist: Ubuntu 18.04 bionic
         locale: UTF-8
        machine: x86_64
        release: 4.18.0-20-generic
         system: Linux
        version: Ubuntu 18.04 bionic
```

7.2.2 Upgrading

To install a newer version, you can execute `pip install -U salt-sproxy`, however this is also going to upgrade your Salt installation. So in case you would like to use a specific Salt version, it might be a better idea to install the specific salt-sproxy version you want. You can check at https://pypi.org/project/salt-sproxy/#history the list of available salt-sproxy versions.

Example:

```
$ pip install salt-sproxy==2019.6.0
```

7.3 Using the Roster Interface

While from the CLI perspective `salt-sproxy` looks like it works similar to the usual `salt` command, in fact, they work fundamentally different. One of the most important differences is that `salt` is aware of what Minions are connected to the Master, therefore it is easy to know what Minions would be matched by a certain target expression (see https://docs.saltstack.com/en/latest/topics/targeting/ for further details). In contrast, by definition, `salt-sproxy` doesn't suppose there are any (Proxy) Minions running, so it cannot possibly know what Minions would be matched by an arbitrary expression. For this reasoning, we need to "help" it by providing the list of all the devices it should be aware of. This is done through the Roster interface; even though this Salt subsystem has initially been developed for salt-ssh.

There are several Roster modules natively available in Salt, or you may write a custom one in your own environment, under the `salt://_roster` directory.

To make it work, you would need to provide two configuration options (either via the CLI, or through the Master configuration file. See *Command Line and Configuration Options*, in particular `-r` (or `-roster`), and `--roster-file` (when the Roster module loads the list of devices from a file).

For example, let's see how we can use the *Ansible Roster*.

7.3.1 Roster usage example: Ansible

If you already have an Ansible inventory, simply drop it into a file, e.g., `/etc/salt/roster`.

Note: The Ansible inventory file doesn't need to provide any connection details, as they must be configured into the Pillar. If you do provide them however, they could be used to override the data compiled from the Pillar. See *Overriding Pillar data* for an example.

With that in mind, let's consider a very simply inventory, e.g.,

`/etc/salt/roster`:

```
[routers]
router1
router2
router3

[switches]
switch1
switch2
```

Reference this file, and tell `salt-sproxy` to interpret this file as an Ansible inventory:

`/etc/salt/master`:

```
roster: ansible
roster_file: /etc/salt/roster
```

To verify that the inventory is interpreted correctly, run the following command which should display all the possible devices salt-sproxy should be aware of:

```
$ salt-sproxy \* --preview-target
- router1
- router2
- router3
- switch1
- switch2
```

Then you can check that your desired target matches - say run against all the routers:

```
$ salt-sproxy 'router*' --preview-target
- router1
- router2
- router3
```

Hint: If you don't provide the Roster name and the path to the Roster file, into the Master config file, you can specify them on the command line, e.g.,

```
$ salt-sproxy 'router*' --preview-target -r ansible --roster-file /etc/salt/roster
```

The default target matching is `glob` (shell-like globbing) - see *Target Selection* for more details, and other target selection options.

Important: Keep in mind that some Roster modules may not implement all the possible target selection options.

Using the inventory above, we can also use the PCRE (Perl Compatible Regular Expression) matching and target devices using a regular expression, e.g.,

```
$ salt-sproxy -E 'router(1|2).?' --preview-target
- router1
- router2
$ salt-sproxy -E '(switch|router)1' --preview-target
- router1
- switch1
```

The inventory file doesn't necessarily need to be flat, can be as complex as you want, e.g.,

```
all:
  children:
    usa:
      children:
        northeast: ~
        northwest:
          children:
            seattle:
```

(continues on next page)

(continued from previous page)

```
              hosts:
                edge1.seattle
            vancouver:
              hosts:
                edge1.vancouver
        southeast:
          children:
            atlanta:
              hosts:
                edge1.atlanta:
                edge2.atlanta:
            raleigh:
              hosts:
                edge1.raleigh:
        southwest:
          children:
            san_francisco:
              hosts:
                edge1.sfo
            los_angeles:
              hosts:
                edge1.la
```

Using this inventory, you can then run, for example, against all the devices in Atlanta, to gather the LLDP neighbors for every device:

```
$ salt-sproxy '*.atlanta' net.lldp
edge1.atlanta:
   ~~~ snip ~~~
edge2.atlanta:
   ~~~ snip ~~~
```

Targeting using groups

Another very important detail here is that, depending on the structure of the inventory, and how the devices are grouped, you can use these groups to target using the `-N` target type (nodegroup). For example, based on the hierarchical inventory file above, we can use these targets:

```
# All devices in the USA:
$ salt-sproxy -N usa --preview-target
- edge1.seattle
- edge1.vancouver
- edge1.atlanta
- edge2.atlanta
- edge1.raleigh
- edge1.la
- edge1.sfo

# All devices in the North-West region:
$ salt-sproxy -N northwest --preview-target
- edge1.seattle
- edge1.vancouver

# All devices in the Atlanta area:
$ salt-sproxy -N atlanta --preview-target
```

(continues on next page)

(continued from previous page)

```
- edge1.atlanta
- edge2.atlanta
```

The nodegroups you can use for targeting depend on the names you've assigned in your inventory, and sometimes may be more useful to use them vs. the device name (which may not contain the area / region / country name).

Overriding Pillar data

In the Roster file (Ansible inventory) you may prefer to have more specific connection credentials for some particular devices. In this case, you only need to specify them directly under the device, or using `host_vars` as normally; for example, let's consider the inventory from the above, with the difference that now `edge1.raleigh` has more specific details:

```
all:
  children:
    usa:
      children:
        northeast: ~
        northwest:
          children:
            seattle:
              hosts:
                edge1.seattle
            vancouver:
              hosts:
                edge1.vancouver
        southeast:
          children:
            atlanta:
              hosts:
                edge1.atlanta:
                edge2.atlanta:
            raleigh:
              hosts:
                edge1.raleigh:
                  username: different
                  password: not-the-same
        southwest:
          children:
            san_francisco:
              hosts:
                edge1.sfo
            los_angeles:
              hosts:
                edge1.la
```

With this Roster, `salt-sproxy` will try to authenticate using the username and password specified. The same goes to the rest of the other credentials and fields required by the Proxy module you're using, i.e., `port`, `optional_args`, etc. - check the Salt documentation to understand what fields you have available.

Configuring static Grains

In a similar way to overriding Pillar data for authentication (see the paragraph above), you can equally configure static Grains per device, by simply providing them under the `grains` key, e.g.,

```
all:
  children:
    usa:
      children:
        northeast: ~
        northwest:
          children:
            seattle:
              hosts:
                edge1.seattle
            vancouver:
              hosts:
                edge1.vancouver
        southeast:
          children:
            atlanta:
              hosts:
                edge1.atlanta:
                edge2.atlanta:
                  grains:
                    role: transit
                    site: atl01
            raleigh:
              hosts:
                edge1.raleigh:
        southwest:
          children:
            san_francisco:
              hosts:
                edge1.sfo
            los_angeles:
              hosts:
                edge1.la
```

With the Roster above, derived from the previous examples, the `edge2.atlanta` device is going to have two static Grains associated, i.e., `site` and `role` with the values as configured in the Roster.

7.3.2 Loading the list of devices from the Pillar

The Pillar subsystem is powerful and flexible enough to be used as an input providing the list of devices and their properties.

To use the *Pillar Roster* you only need to ensure that you can access the list of devices you want to manage into a Pillar. The Pillar system is designed to provide data (from whatever source, i.e., HTTP API, database, or any file format you may prefer) to one specific Minion (or some / all). That doesn't mean that the Minion must be up and running, but simply just that one or more Minions have access to this data.

In the Master configuration file, configure the `roster` or `proxy_roster`, e.g.,

```
roster: pillar
```

By default, the Pillar Roster is going to check the Pillar data for `*` (any Minion), and load it from the `devices` key. In other words, when executing `salt-sproxy pillar.show_pillar` the output should have at least the `devices` key. To use different settings, have a look at the documentation: *Pillar Roster*.

Consider the following example setup:

`/etc/salt/master`

```
pillar_roots:
  base:
    - /srv/salt/pillar

roster: pillar
```

`/srv/salt/pillar/top.sls`

```
base:
  '*':
    - devices_pillar
  'minion*':
    - dummy_pillar
```

`/srv/salt/pillar/devices_pillar.sls`

```
devices:
  - name: minion1
  - name: minion2
```

`/srv/salt/pillar/dummy_pillar.sls`

```
proxy:
  proxytype: dummy
```

With this configuration, you can verify that the list of expected devices is properly defined:

```
$ salt-run pillar.show_pillar
devices:
    |_
      ----------
      name:
          minion1
    |_
      ----------
      name:
          minion2
```

Having this available, we can now start using salt-sproxy:

```
$ salt-sproxy \* --preview-target
- minion1
- minion2
```

When working with Pillar SLS files, you can provide them in any format, either Jinja + YAML, or pure Python, e.g. generate a longer list of devices, dynamically:

`/srv/salt/pillar/devices_pillar.sls`

```
devices:
  {% for id in range(100) %}
  - name: minion{{ id }}
  {%- endfor %}
```

Or:

`/srv/salt/pillar/devices_pillar.sls`

```
#!py

def run():
    return {
        'devices': [
            'minion{}'.format(id_)
            for id_ in range(100)
        ]
    }
```

Note: The latter Python example would be particularly useful when the data compilation requires more computation, while keeping the code readable, e.g., execute HTTP requests, or anything you can usually do in Python scripts in general.

With either of the examples above, the targeting would match:

```
$ salt-sproxy \* --preview-target
- minion0
- minion1

~~~ snip ~~~

- minion98
- minion99
```

As the Pillar SLS files are flexible enough to allow you to compile the list of devices you want to manage using whatever way you need and possibly coded in Python. Say we would want to gather the list of devices from an HTTP API:

`/srv/salt/pillar/devices_pillar.sls`

```
#!py

import requests

def run():
    ret = requests.post('http://example.com/devices')
    return {'devices': ret.json()}
```

Or another example, slightly more advanced - retrieve the devices from a MySQL database:

`/srv/salt/pillar/devices_pillar.sls`

```
#!py

import mysql.connector

def run():
   devices = []
   mysql_conn = mysql.connector.connect(host='localhost',
                                         database='database',
                                         user='user',
                                         password='password')
   get_devices_query = 'select * from devices'
   cursor = mysql_conn.cursor()
   cursor.execute(get_devices_query)
```

(continues on next page)

(continued from previous page)

```
    records = cursor.fetchall()
    for row in records:
        devices.append({'name': row[1]})
    cursor.close()
    return {'devices': devices}
```

Important: Everything with the Pillar system remains the same as always, so you can very well use also the External Pillar to provide the list of devices - see https://docs.saltstack.com/en/latest/ref/pillar/all/index.html for the list of the available External Pillars modules that allow you to load data from various sources.

Check also the *Using the Pillar Roster* example on how to load the list of devices from an External Pillar, as the functionaly you may need might already be implemented and available.

Configuring static Grains

Using the `devices_pillar.sls` file from the previous examples, you can provide static Grains per device, under the `grains` key, e.g.,

`/srv/salt/pillar/devices_pillar.sls`

```
devices:
  {% for id in range(100) %}
  - name: minion{{ id }}
    grains:
      site: site{{ id }}
  {%- endfor %}
```

In this case, the Grains data is dynamically generated through the Jinja loop, however it could be provided in any way you'd prefer. Executing the following command, you can check that the Grains data is properly distributed:

```
$ salt-sproxy minion17 grains.get site
minion17:
    site17
```

7.3.3 Roster usage example: NetBox

The *NetBox Roster* is a good example of a Roster modules that doesn't work with files, rather gathers the data from NetBox via the API.

Note: The NetBox Roster module is currently not available in the official Salt releases, and it is distributed as part of the `salt-sproxy` package and dynamically loaded on runtime, so you don't need to worry about that, simply reference it, configure the details as documented and start using it straight away.

To use the NetBox Roster, simply put the following details in the Master configuration you want to use (default `/etc/salt/master`):

```
roster: netbox

netbox:
 url: <NETBOX_URL>
```

You can also specify the `token`, and the `keyfile` but for this Roster specifically, the `url` is sufficient.

To verify that you are indeed able to retrieve the list of devices from your NetBox instance, you can, for example, execute:

```
$ salt-run salt.cmd netbox.filter dcim devices
# ~~~ should normally return all the devices ~~~

# Or with some specific filters, e.g.:
$ salt-run salt.cmd netbox.filter dcim devices site=<SITE> status=<STATUS>
```

Once confirmed this works well, you can verify that the Roster is able to pull the data:

```
$ salt-sproxy '*' --preview-target
```

In the same way, you can then start executing Salt commands targeting using expressions that match the name of the devices you have in NetBox:

```
$ salt-sproxy '*atlanta' net.lldp
edge1.atlanta:
    ~~~ snip ~~~
edge2.atlanta:
    ~~~ snip ~~~
```

Enhanced Grain targeting

When NetBox Roster pulls the data from NetBox via the API, from the `dcim` app, `devices` endpoint, it retrieves additional information about the device, e.g.,

```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 1,
            "name": "edge1.vlc1",
            "display_name": "edge1.vlc1",
            "device_type": {
                "id": 1,
                "url": "https://netbox.live/api/dcim/device-types/1/",
                "manufacturer": {
                    "id": 5,
                    "url": "https://netbox.live/api/dcim/manufacturers/5/",
                    "name": "Juniper",
                    "slug": "juniper"
                },
                "model": "MX960",
                "slug": "mx960",
                "display_name": "Juniper MX960"
            },
            "device_role": {
                "id": 7,
                "url": "https://netbox.live/api/dcim/device-roles/7/",
                "name": "Router",
                "slug": "router"
```

(continues on next page)

(continued from previous page)

```
            },
            "tenant": null,
            "platform": {
                "id": 3,
                "url": "https://netbox.live/api/dcim/platforms/3/",
                "name": "Juniper Junos",
                "slug": "juniper-junos"
            },
            "serial": "",
            "asset_tag": null,
            "site": {
                "id": 1,
                "url": "https://netbox.live/api/dcim/sites/1/",
                "name": "VLC1",
                "slug": "vlc1"
            },
            "rack": {
                "id": 1,
                "url": "https://netbox.live/api/dcim/racks/1/",
                "name": "R1",
                "display_name": "R1"
            },
            "position": 1,
            "face": {
                "value": 0,
                "label": "Front"
            },
            "parent_device": null,
            "status": {
                "value": 1,
                "label": "Active"
            },
            "primary_ip": null,
            "primary_ip4": null,
            "primary_ip6": null,
            "cluster": null,
            "virtual_chassis": null,
            "vc_position": null,
            "vc_priority": null,
            "comments": "",
            "local_context_data": null,
            "tags": [],
            "custom_fields": {},
            "created": "2019-08-12",
            "last_updated": "2019-08-12T11:08:21.706641Z"
        }
    ]
}
```

All this data is by default available in the Grains when targeting, so you can use the *Grain* to match the devices you want to run against.

Examples:

- Select devices under the `router` role:

```
salt-sproxy -G netbox:device_role:role test.ping
```

- Select devices from the `vlc1` site:

```
salt-sproxy -G netbox:site:slug:vlc1 test.ping
```

7.3.4 Other Roster modules

If you may need to load your data from various other data sources, that might not be covered in the existing Roster modules. Roster modules are easy to write, and you only need to drop them into your `salt://_roster` directory, then it would be great if you could open source them for the benefit of the community (either submit them to this repository, at https://github.com/mirceaulinic/salt-sproxy, or to the official Salt repository on GitHub)

7.4 Managing Static Grains

Grains are generally a delicate topic in Salt, particularly on Proxy Minions which need to be able to connect to the remote device to collect the Grains, while the connection credentials may depend on the Grains themselves - that becomes and chicken and egg type problem!

In *salt-sproxy*, you can configure static Grains, in different ways. One of the easiest is adding static data under the `grains` (or `sproxy_grains` or `default_grains`) key in the Master config file, for example:

`/etc/salt/master`

```
grains:
  salt:
    role: proxy
```

The static Grains configured in this way are going to be shared among all the devices / Minions managed via *salt-sproxy*.

Important: The static Grains configured in these ways are available to be used in your target expressions. For example, the above can be used, e.g., `salt-sproxy -G salt:role:proxy --preview-target`.

To configure more specific Grains per device, or groups of devices, you have the following options:

7.4.1 Static Grains in File

To configure static Grains for one specific device, you can put your data as described in https://docs.saltstack.com/en/latest/topics/grains/#grains-in-etc-salt-grains, more specifically under the `/etc/salt/proxy.d/` directory. For example, if you'd want to configure for the device `router1`, you'd have the following file:

`/etc/salt/proxy.d/router1/grains`

```
role: router
```

7.4.2 Static Grains in Roster

Some *Extension Roster Modules* modules allow you to put static Grains granularly. See, for example *Configuring static Grains* (for the *Pillar Roster*) or *Configuring static Grains* (for the *Ansible Roster*).

7.5 Targeting

Targeting devices is largely based on the Roster interface. This is the starting point, where *salt-sproxy* know what devices you want to manage. The Roster interface can be a Pillar file, an Ansible inventory file, a NetBox instance and so on. See *Using the Roster Interface* for more details, usage examples and documentation for each of the available Roster options.

To put it in other words, the Roster provides the totality (or the universe) of devices you have. When you're executing a command, you may want to execute a command against all these devices, or only a subset of them. There are several targeting selection mechanisms, as presented below.

Targeting in *salt-sproxy*, from an user perspective, is very similar to the native Salt targeting - however, the implementation is fundamentally different (again, please see *Using the Roster Interface* for more details on this); that's why the targeting in *salt-sproxy* comes with some caveats you should be aware of.

Tip: Before executing any command, it may be a good idea to check that your target matches the devices you want to run against, by using the `--preview-target` CLI option, e.g.,

```
salt-sproxy -G netbox:role:router --preview-target
```

See also:

When targeting making use of Grains or Pillar data that depend on the device characteristics (such as interfaces, IP addresses, OS version, platform details, and so on), or other properties retrieved from other systems, such as APIs, databases, etc., you may want to look at –invasive-targeting or –preload-targeting options.

7.5.1 Glob

Shell-style globbing on the device name / Minion ID.

See https://docs.saltstack.com/en/latest/topics/targeting/globbing.html#globbing

Examples:

- Match all the devices *salt-sproxy* knows about:

```
salt-sproxy '*' test.ping
```

- Match `edge1` and `edge3` devices:

```
salt-sproxy 'edge[1,3]' test.ping
```

7.5.2 PCRE

PCRE stands for Perl Compatible Regular Expression, so you can target against devices with the name matching the regular expression.

See also: https://docs.saltstack.com/en/latest/topics/targeting/globbing.html#regular-expressions

Example: match top of rack switches with the name ending in a digit:

```
salt-sproxy -E '.*-tor\d' napalm.junos_rpc get-route-summary-information table=mpls.0
```

7.5.3 List

A list of device names.

Example: execute a command on three devices `edge1`, `edge2`, and `edge3`:

```
salt-sproxy -L 'edge1,edge2,edge3' net.arp
```

7.5.4 Grain

Targeting using Grain data.

This is a tricky subject. Unlike the native Salt, *salt-sproxy* doesn't have access to device data before connecting to it (i.e., it can't possibly know device details before even connecting to it). You can however target using Grain data, but there are some caveats, and it's up to you to decide whether you want performance or limit the resource consumption.

See also:

See also: *Managing Static Grains*. Static Grains are always available, and can be anytime used in your targeting, without any restrictions.

An exception is the *NetBox Roster* module which provides an additional set of Grains you can use, under the `netbox` key. See the *Enhanced Grain targeting* section for more details.

Examples: match devices on their role:

```
salt-sproxy -G role:router test.ping
```

7.5.5 Grain PCRE

As the `grain` targeting, but instead of exact matching, can match on a regular expression on the Grain value.

Example: match the devices from multiple sites (e.g., `lon1`, `lon2`, etc.)

```
salt-sproxy -P site:lon\d test.ping
```

7.5.6 Compound

You can mix all the matchers above. See https://docs.saltstack.com/en/latest/topics/targeting/compound.html for more details and notes.

Example: match edge routers 1 and 3 from multiple sites

```
salt-sproxy -C 'edge[1,3] and G@role:router and P@site:lon\d' net.lldp
```

7.6 Command Line and Configuration Options

There are a few options specific for `salt-sproxy`, however you might be already familiar with a vast majority of them from the salt or salt-run Salt commands.

Hint: Many of the CLI options are available to be configured through the file you can specify through the `-c` (`-config-dir`) option, with the difference that in the file you need to use the longer name and underscore instead of hyphen. For example, the `--roster-file` option would be configured as `roster_file: /path/to/roster/file` in the config file.

`--version`
Print the version of Salt and Salt SProxy that is running.

`--versions-report`
Show program's dependencies and version number, and then exit.

`-h, --help`
Show the help message and exit.

`-c` `CONFIG_DIR`**`, --config-dir`**`=CONFIG_dir`
The location of the Salt configuration directory. This directory contains the configuration files for Salt master and minions. The default location on most systems is `/etc/salt`.

`--config-dump`
New in version 2020.2.0.

Print the complete salt-sproxy configuration values (with the defaults), as YAML.

`-r, --roster`
The Roster module to use to compile the list of targeted devices.

`--roster-file`
Absolute path to the Roster file to load (when the Roster module requires a file). Default: `/etc/salt/roster`.

`--invasive-targeting`
New in version 2020.2.0.

The native *salt-sproxy* targeting highly depends on the data your provide mainly through the Roster system (see also *Using the Roster Interface*). Through the Roster interface and other mechanisms, you are able to provide static Grains (see also *Managing Static Grains*), which you can use in your targeting expressions. There are situations when you may want to target using more dynamic Grains that you probably don't want to manage statically.

In such case, the `--invasive-targeting` targeting can be helpful as it connects to the device, retrieves the Grains, then executes the requested command, *only* on the devices matched by your target.

Important: The maximum set of devices you can query is the devices you have defined in your Roster – targeting in this case helps you select a subset of the devices *salt-sproxy* is aware of, based on their properties.

Caution: While this option can be very helpful, bear in mind that in order to retrieve all this data, *salt-sproxy* initiates the connection with ALL the devices provided through the Roster interface. That means, not only that resources consumption is expected to increase, but also the execution time would similarlly be higher. Depending on your setup and use case, you may want to consider using `--cache-grains` and / or `--cache-pillar`. The idea is to firstly run `--invasive-targeting` together with `--cache-grains` and / or `--cache-pillar`, in order to cache your data, and the subsequent executions through *salt-sproxy* are going to use that data, device target matching included.

`--preload-targeting`
New in version 2020.2.0.

This is a lighter derivative of the `--invasive-targeting` option (see above), with the difference that *salt-sproxy* is not going to establish the connection with the remote device to gather the data, but will just load all the possible data without the connection. In other words, you can look at it like a combination of both `--invasive-targeting` and `-no-connect` used together.

This option is useful when the Grains and Pillars you want to use in your targeting expression don't depend on the connection with the device itself, but they are dynamically pulled from various systems, e.g., from an HTTP API, database, etc.

`--sync`
Deprecated since version 2020.2.0: This option has been replaced by `--static` (see below).

Whether should return the entire output at once, or for every device separately as they return.

`-s, --static`
New in version 2020.2.0: Starting with this release, `--static`, replaces the previous CLI option `--sync`, with the same functionality.

Whether should return the entire output at once, or for every device separately as they return.

`--cache-grains`
Cache the collected Grains. Beware that this option overwrites the existing Grains. This may be helpful when using the `salt-sproxy` only, but may lead to unexpected results when running in *Mixed Environments*.

`--cache-pillar`
Cache the collected Pillar. Beware that this option overwrites the existing Pillar. This may be helpful when using the `salt-sproxy` only, but may lead to unexpected results when running in *Mixed Environments*.

`--no-cached-grains`
Do not use the cached Grains (i.e., always collect Grains).

`--no-cached-pillar`
Do not use the cached Pillar (i.e., always re-compile the Pillar).

`--no-grains`
Do not attempt to collect Grains at all. While it does reduce the runtime, this may lead to unexpected results when the Grains are referenced in other subsystems.

`--no-pillar`
Do not attempt to compile Pillar at all. While it does reduce the runtime, this may lead to unexpected results when the Pillar data is referenced in other subsystems.

`-b, --batch, --batch-size`
The number of devices to connect to in parallel.

`--batch-wait`
New in version 2020.2.0.

Wait a specific number of seconds after each batch is done before executing the next one.

`-p, --progress`
New in version 2020.2.0.

Display a progress graph to visually show the execution of the command across the list of devices.

Note: As of release 2020.2.0, the best experience of using the progress graph is in conjunction with the `-s` / `--static` option, otherwise there's a small display issue.

`--hide-timeout`
New in version 2020.2.0.

Hide devices that timeout.

--failhard

New in version 2020.2.0.

Stop the execution at the first error.

--summary

New in version 2020.2.0.

Display a summary of the command execution:

- Total number of devices targeted.
- Number of devices that returned without issues.
- Number of devices that timed out executing the command. See also `-t` or `--timeout` argument to adjust the timeout value.
- Number of devices with errors (i.e., there was an error while executing the command).
- Number of unreachable devices (i.e., couldn't establish the connection with the remote device).

In `-v` / `--verbose` mode, this output is enahnced by displaying the list of devices that did not return / with errors / unreachable.

Example:

```
-------------------------------------------
Summary
-------------------------------------------
# of devices targeted: 10
# of devices returned: 3
# of devices that did not return: 5
# of devices with errors: 0
# of devices unreachable: 2
-------------------------------------------
```

--show-jid

New in version 2020.2.0.

Display jid without the additional output of –verbose.

-v, --verbose

New in version 2020.2.0.

Turn on command verbosity, display jid, devices per batch, and detailed summary.

--preview-target

Show the devices expected to match the target, without executing any function (i.e., just print the list of devices matching, then exit).

--sync-roster

Synchronise the Roster modules (both salt-sproxy native and provided by the user in their own environment). Default: `True`.

--sync-modules

New in version 2019.10.0.

Load the Execution modules provided together with salt-sproxy. Beware that it may override the Salt native modules, or your own extension modules. Default: `False`.

You can also add `sync_modules:  true` into the Master config file, if you want to always ensure that salt-sproxy is using the Execution modules delivered with this package.

--sync-grains

New in version 2019.10.0.

Synchronise the Grains modules you may have in your own environment.

--sync-all

New in version 2020.2.0.

Load the all extension modules provided with salt-sproxy, as well as your own extension modules from your environment.

--saltenv

New in version 2020.2.0.

The Salt environment name where to load extension modules and files from.

--events

Whether should put the events on the Salt bus (mostly useful when having a Master running). Default: `False`.

Important: See *Event-Driven Automation and Orchestration* for further details.

--use-existing-proxy

Execute the commands on an existing Proxy Minion whenever available. If one or more Minions matched by the target don't exist (or the key is not accepted by the Master), salt-sproxy will fallback and execute the command locally, and, implicitly, initiate the connection to the device locally.

Note: This option requires a Master to be up and running. See *Mixed Environments* for more information.

Important: When using this option in combination with a Roster, `salt-sproxy` will firstly try to match your target based on the provided Roster, and then only after that will execute the Salt function on the targets, and on the existing Proxy Minions, best efforts. For example, if your target matches two devices, say `router1` and `switch1`, and there's an available Proxy Minion running for `router1`, then the Salt function would be executed on the `router1` existing Minion, over the already established connection, while for `switch1` the connection is going to be initialised during run time.

If you want to bypass the Roster matching, and target *only* existing (Proxy) Minions, make sure you don't have the `roster` or `proxy_roster` options configured, or execute with `-r None`, e.g.,

```
$ salt-sproxy \* --preview-target --use-existing-proxy -r None
```

The command above would be the equivalent of the following Salt command: `salt \* --preview-target`.

--no-connect

New in version 2019.10.0.

Do not initiate the connection with the remote device. Please use this option with care, as it may lead to unexptected results. The main use case (although not limited to) is executing Salt functions that don't necessarily require the connection, however they may need Pillar or Grains that are associated with each individual device. Such examples include HTTP requests, working with files, and so on. Keep in mind that, as the connection is not established, it won't re-compile fresh Grains, therefore it'll be working with cached data. Make sure that the data you have available is already cached before executing with `--no-connect`, by executing `grains.items` and / or `pillar.items`. The point of this functionality is to speed up the execution when dealing with a large volume of execution events (either from the CLI or through the *The Proxy Runner*), and when the connection is not actually absolutely necessary.

--test-ping
New in version 2019.10.0.

When executing with `--use-existing-proxy`, you can use this option to verify whether the Minion is responsive, and only then attempt to send out the command to be executed on the Minion, otherwise executed the function locally.

Note: Keep in mind that this option generates an additional event on the bus for every execution.

--no-target-cache
New in version 2019.10.0.

Avoid loading the list of targets from the cache.

--pillar-root
New in version 2020.2.0.

Set a specific directory as the base pillar root.

--file-root
New in version 2020.2.0.

Set a specific directory as the base file root.

--states-dir
New in version 2020.2.0.

Set a specific directory to search for additional States.

-m, --module-dirs
New in version 2020.2.0.

Specify one or more directories where to load the extension modules from. Multiple directories can be provided by passing `-m` or `--module-dirs` multiple times.

--file-roots, --display-file-roots
Display the location of the salt-sproxy installation, where you can point your `file_roots` on the Master, to use the *Proxy Runner* and other extension modules included in the salt-sproxy package. See also *The Proxy Runner*.

--save-file-roots
Save the configuration for the `file_roots` in the Master configuration file, in order to start using the *Proxy Runner* and other extension modules included in the salt-sproxy package. See also *The Proxy Runner*. This option is going to add the salt-sproxy installation path to your existing `file_roots`.

7.6.1 Logging Options

Logging options which override any settings defined on the configuration files.

-l LOG_LEVEL, **--log-level**=LOG_LEVEL
Console logging log level. One of `all`, `garbage`, `trace`, `debug`, `info`, `warning`, `error`, `quiet`. Default: `error`.

--log-file=LOG_FILE
Log file path. Default: `/var/log/salt/master`.

--log-file-level=LOG_LEVEL_LOGFILE
Logfile logging log level. One of `all`, `garbage`, `trace`, `debug`, `info`, `warning`, `error`, `quiet`. Default: `error`.

7.6.2 Target Selection

The default matching that Salt utilizes is shell-style globbing around the minion id. See https://docs.python.org/2/library/fnmatch.html#module-fnmatch.

See also:

Targeting

`-E, --pcre`
The target expression will be interpreted as a PCRE regular expression rather than a shell glob.

`-L, --list`
The target expression will be interpreted as a comma-delimited list; example: server1.foo.bar,server2.foo.bar,example7.quo.qux

`-G, --grain`
The target expression matches values returned by the Salt grains system on the minions. The target expression is in the format of '<grain value>:<glob expression>'; example: 'os:Arch*'

This was changed in version 0.9.8 to accept glob expressions instead of regular expression. To use regular expression matching with grains, use the –grain-pcre option.

`-P, --grain-pcre`
The target expression matches values returned by the Salt grains system on the minions. The target expression is in the format of '<grain value>:< regular expression>'; example: 'os:Arch.*'

`-N, --nodegroup`
Use a predefined compound target defined in the Salt master configuration file.

`-R, --range`
Instead of using shell globs to evaluate the target, use a range expression to identify targets. Range expressions look like %cluster.

Using the Range option requires that a range server is set up and the location of the range server is referenced in the master configuration file.

7.6.3 Output Options

`--out`
Pass in an alternative outputter to display the return of data. This outputter can be any of the available outputters:

> `highstate`, `json`, `key`, `overstatestage`, `pprint`, `raw`, `txt`, `yaml`, `table`, and many others.

Some outputters are formatted only for data returned from specific functions. If an outputter is used that does not support the data passed into it, then Salt will fall back on the `pprint` outputter and display the return data using the Python `pprint` standard library module.

Note: If using `--out=json`, you will probably want `--static` as well. Without the sync option, you will get a separate JSON string per minion which makes JSON output invalid as a whole. This is due to using an iterative outputter. So if you want to feed it to a JSON parser, use `--static` as well.

`--out-indent` `OUTPUT_INDENT`, **`--output-indent`** `OUTPUT_INDENT`
Print the output indented by the provided value in spaces. Negative values disable indentation. Only applicable in outputters that support indentation.

`--out-file``=OUTPUT_FILE`, **`--output-file`**`=OUTPUT_FILE`
Write the output to the specified file.

--out-file-append, --output-file-append
Append the output to the specified file.

--no-color
Disable all colored output

--force-color
Force colored output

Note: When using colored output the color codes are as follows:

`green` denotes success, `red` denotes failure, `blue` denotes changes and success and `yellow` denotes a expected future change in configuration.

--state-output=STATE_OUTPUT, **--state_output**=STATE_OUTPUT
Override the configured state_output value for minion output. One of 'full', 'terse', 'mixed', 'changes' or 'filter'. Default: 'none'.

--state-verbose=STATE_VERBOSE, **--state_verbose**=STATE_VERBOSE
Override the configured state_verbose value for minion output. Set to True or False. Default: none.

7.7 The Proxy Runner

The *Proxy Runner* is the core functionality of `salt-sproxy` and can be used to trigger jobs as *Reactions to external events*, or invoked when *Using the Salt REST API*.

In both cases mentioned above you are going to need to have a Salt Master running, that allows you to set up the Reactors and the Salt API; that means, the `proxy` Runner needs to be available on your Master. To do so, you have two options:

7.7.1 1. Reference it from the salt-sproxy installation

After installing salt-sproxy, you can execute the following command:

```
$ salt-sproxy --file-roots
salt-sproxy is installed at: /home/mircea/venvs/salt-sproxy/lib/python3.6/site-
↪packages/salt_sproxy

You can configure the file_roots on the Master, e.g.,

file_roots:
  base:
    - /home/mircea/venvs/salt-sproxy/lib/python3.6/site-packages/salt_sproxy

Or only for the Runners:

runner_dirs:
  - /home/mircea/venvs/salt-sproxy/lib/python3.6/site-packages/salt_sproxy/_runners
```

1.a. Update the `file_roots` and / or `runner_dirs` manually

As suggested in the output, you can directly reference the salt-sproxy installation path to start using the `proxy` Runner (and other extension modules included in the package).

After updating the master configuration file, make sure to execute `salt-run saltutil.sync_all` or `salt-run saltutil.sync_runners`.

1.b. Use the `--save-file-roots` CLI argument to update the master config

A simpler alternative is executing with `--save-file-roots` which adds the path for you, and synchronizes the extension modules provided together with e.g.,

```
$ salt-sproxy --save-file-roots
/home/mircea/venvs/salt-sproxy/lib/python3.6/site-packages/salt_sproxy added to the␣
↪file_roots:

file_roots:
  base:
    - /home/mircea/venvs/salt-sproxy/lib/python3.6/site-packages/salt_sproxy

Now you can start using salt-sproxy for event-driven automation, and the Salt REST␣
↪API.
See https://salt-sproxy.readthedocs.io/en/latest/salt_api.html
and https://salt-sproxy.readthedocs.io/en/latest/events.html for more details.
```

Note: While this option will preserve the configuration you have (but appending another path to `file_roots` and / or `runner_dirs`), it may re-arrange (visually) the contents - however without any side effects.

7.7.2 2. Copy the source file

You can either download it from https://github.com/mirceaulinic/salt-sproxy/blob/master/salt_sproxy/_runners/proxy.py, e.g., if your `file_roots` configuration on the Master looks like:

```
file_roots:
  base:
    - /srv/salt
```

You are going to need to create a directory under `/srv/salt/_runners`, then download the `proxy` Runner there:

```
$ mkdir -p /srv/salt/_runners
$ curl -o /srv/salt/_runners/proxy.py -L \
  https://raw.githubusercontent.com/mirceaulinic/salt-sproxy/master/salt_sproxy/_
↪runners/proxy.py
```

Note: In the above I've used the *raw* like from GitHub to ensure the source code is preserved.

Alternatively, you can also put it under an arbitrary path, e.g., (configuration on the Master)

```
runner_dirs:
  - /path/to/runners
```

Downloading the `proxy` Runner under that specific path:

```
$ curl -o /path/to/runners/proxy.py -L \
  https://raw.githubusercontent.com/mirceaulinic/salt-sproxy/master/salt_sproxy/_
↪runners/proxy.py
```

7.8 Event-Driven Automation and Orchestration

7.8.1 Execution Events

Even though `salt-sproxy` has been designed to be an on-demand executed process (as in opposite to an always running service), you still have the possibility to monitor what is being executed, and potentially export these events or trigger a Reactor execution in response.

Note: To be able to have events, you will need to have a Salt Master running, and preferrably using the same Master configuration file as salt-sproxy, to ensure that they are both sharing the same socket object.

Using the `--events` option on the CLI (or by configuring `events: true` in the Master configuration file), `salt-sproxy` is going to inject events on the Salt bus as you're running the usual Salt commands.

For example, running the following command (from the salt-sproxy with network devices example):

```
$ salt-sproxy juniper-router net.arp --events
```

Watching the event bus on the Master, you should notice the following events:

```
$ salt-run state.event pretty=True
20190529143434052740        {
    "_stamp": "2019-05-29T14:34:34.053900",
    "minions": [
        "juniper-router"
    ]
}
proxy/runner/20190529143434054424/new       {
    "_stamp": "2019-05-29T14:34:34.055386",
    "arg": [],
    "fun": "net.arp",
    "jid": "20190529143434054424",
    "minions": [
        "juniper-router"
    ],
    "tgt": "juniper-router",
    "tgt_type": "glob",
    "user": "mircea"
}
proxy/runner/20190529143434054424/ret/juniper-router        {
    "_stamp": "2019-05-29T14:34:36.937409",
    "fun": "net.arp",
    "fun_args": [],
    "id": "juniper-router",
    "jid": "20190529143434054424",
    "return": {
        "out": [
            {
                "interface": "fxp0.0",
                "mac": "92:99:00:0A:00:00",
                "ip": "10.96.0.1",
                "age": 926.0
            },
            {
                "interface": "fxp0.0",
```

(continues on next page)

(continued from previous page)

```
                "mac": "92:99:00:0A:00:00",
                "ip": "10.96.0.13",
                "age": 810.0
            },
            {
                "interface": "em1.0",
                "mac": "02:42:AC:13:00:02",
                "ip": "128.0.0.16",
                "age": 952.0
            }
        ],
        "result": true,
        "comment": ""
    },
    "success": true
}
```

As in the example, above, every execution pushes at least three events:

- Job creation. The tag is the JID of the execution.
- Job payload with the job details, i.e., function name, arguments, target expression and type, matched devices, etc.
- One separate return event from every device.

A more experienced Salt user may have already noticed that the structure of these events is *very* similar to the usual Salt native events when executing a regular command using the usual `salt`. Let's take an example for clarity:

```
$ salt 'test-minion' test.ping
test-minion:
    True
```

The event bus:

```
$ salt-run state.event pretty=True
20190529144939496567        {
    "_stamp": "2019-05-29T14:49:39.496954",
    "minions": [
        "test-minion"
    ]
}
salt/job/20190529144939496567/new   {
    "_stamp": "2019-05-29T14:49:39.498021",
    "arg": [],
    "fun": "test.ping",
    "jid": "20190529144939496567",
    "minions": [
        "test-minion"
    ],
    "missing": [],
    "tgt": "test-minion",
    "tgt_type": "glob",
    "user": "sudo_mulinic"
}
salt/job/20190529144939496567/ret/test-minion       {
    "_stamp": "2019-05-29T14:49:39.905727",
    "cmd": "_return",
```

(continues on next page)

(continued from previous page)

```
    "fun": "test.ping",
    "fun_args": [],
    "id": "test-minion",
    "jid": "20190529144939496567",
    "retcode": 0,
    "return": true,
    "success": true
}
```

That said, if you already have Reactors matching Salt events, in order to trigger them in response to salt-sproxy commands, you would only need to update the tag matching expression (i.e., besides `salt/job/20190529144939496567/new` should also match `proxy/runner/20190529143434054424/new` tags, etc.).

In the exact same way with other Engine types – if you already have Engines exporting events, they should be able to export salt-sproxy events as well, which is a great easy win for PCI compliance, and generally to monitor who executes what.

7.8.2 Reactions to external events

Using the *The Proxy Runner*, you can configure a Reactor to execute a Salt function on a (network) device in response to an event.

For example, let's consider network events from napalm-logs. To import the napalm-logs events on the Salt bus, simply enable the napalm_syslog Salt Engine on the Master.

In response to an INTERFACE_DOWN notification, say we define the following reaction, in response to events with the `napalm/syslog/*/INTERFACE_DOWN/*` pattern (i.e., matching events such as `napalm/syslog/iosxr/INTERFACE_DOWN/edge-router1`, `napalm/syslog/junos/INTERFACE_DOWN/edge-router2`, etc.):

`/etc/salt/master`

```
reactor:
  - 'napalm/syslog/*/INTERFACE_DOWN/*':
    - salt://reactor/if_down_shutdown.sls
```

The `salt://reactor/if_down_shutdown.sls` translates to `/etc/salt/reactor/if_down_shutdown.sls` when `/etc/salt` is one of the configured `file_roots`. To apply a configuration change on the device with the interface down, we can use the `_runner.proxy.execute()` Runner function:

```
shutdown_interface:
  runner.proxy.execute:
    - tgt: {{ data.host }}
    - function: net.load_template
    - kwarg:
        template_name: salt://templates/shut_interface.jinja
        interface_name: {{ data.yang_message.interfaces.interface.keys()[0] }}
```

This Reactor would apply a configuration change as rendered in the Jinja template `salt://templates/shut_interface.jinja` (physical path `/etc/salt/templates/shut_interface.jinja`). Or, to have an end-to-end overview of the system: when the device sends a notification that one interface is down, in response, Salt is automatically going to try and remediate the problem (in the `shut_interface.jinja` template you can define the business logic you need). Similarly, you can have other concurrent reactions to the same, e.g. to send a Slack notification, and email and so on.

For reactions to `napalm-logs` events specifically, you can continue reading more at https://mirceaulinic.net/2017-10-19-event-driven-network-automation/ for a more extensive introduction and the napalm-logs documentation available at https://napalm-logs.readthedocs.io/en/latest/, with the difference that instead of calling a Salt function directly, you go through the `_runner.proxy.execute()` or `_runner.proxy.execute_devices()` Runner functions.

7.9 Using the Salt REST API

To be able to use the Salt HTTP API, similarly to *Event-Driven Automation and Orchestration*, you will need to have the Salt Master running, and, of course, also the Salt API service.

As the core functionality if based on the *Proxy Runner*, check out first the notes from *The Proxy Runner* to understand how to have the `proxy` Runner available on your Master.

The Salt API configuration is unchanged from the usual approaches: see https://docs.saltstack.com/en/latest/ref/netapi/all/salt.netapi.rest_cherrypy.html how to configure and https://docs.saltstack.com/en/latest/ref/cli/salt-api.html how to start up the salt-api process.

Suppose we have the following configuration:

`/etc/salt/master`

```
rest_cherrypy:
  port: 8080
  ssl_crt: /etc/pki/tls/certs/localhost.crt
  ssl_key: /etc/pki/tls/certs/localhost.key
```

Hint: Consider looking at the *Salt REST API* and *salt-sapi* examples for end-to-end examples on configuring the Salt API or `salt-sapi`, however the official Salt documentation should always be used as the reference.

7.9.1 Starting with salt-sproxy 2020.2.0

Beginning with the *salt-sproxy* release 2020.2.0, the usage has been simplified compared to previous versions, and a new API client has been added, named `sproxy`, together with its counter-part `sproxy_async` for asynchronous requests.

See also:

salt-sapi

In order to do so, instead of starting the usual `salt-api` process, you'd need to start a separate application named `salt-sapi` which is shipped together with *salt-sproxy*. Everything stay the exact same as usually, the only difference being the special `sproxy` and `sproxy_async` clients for simplified usage.

A major advantage of using the `sproxy` / `sproxy_async` clients is that the usage is very similar to the `local` / `local_async` clients (see https://docs.saltstack.com/en/latest/ref/netapi/all/salt.netapi.rest_cherrypy.html#usage), the arguments you'd need to being in-line with the ones from LocalClient: `tgt` (target expression) and `fun` (the name of the Salt function to execute) as mandatory arguments, plus a number of optional arguments documented at https://salt-sproxy.readthedocs.io/en/latest/runners/proxy.html#_runners.proxy.execute. See an usage example below.

Hint: If you are already using Salt API, and would like to make use of the `sproxy` / `sproxy_async` client(s), you may want to use the `salt-sapi` instead of the `salt-api` program, and you'll be able to use the Salt API as always, armed with the *salt-sproxy* clients as well.

Tip: As mentioned in https://docs.saltstack.com/en/latest/ref/netapi/all/salt.netapi.rest_cherrypy.html#best-practices,

> Running asynchronous jobs results in being able to process [. . .] 17x more commands per second (as the `sproxy_async` requests make use of the `RunnerClient` interface).

Running with `sproxy_async` will return you a JID with you can then later use to gather the job returns:

> Job returns can be fetched from Salt's job cache via the `/jobs/<jid>` endpoint, or they can be collected into a data store using Salt's Returner system.

See https://docs.saltstack.com/en/latest/ref/netapi/all/salt.netapi.rest_cherrypy.html#jobs for further details.

After starting the `salt-sapi` process, you should get the following:

```
$ curl -i localhost:8080
HTTP/1.1 200 OK
Content-Type: application/json
Server: CherryPy/18.3.0
Date: Thu, 02 Jan 2020 23:13:28 GMT
Allow: GET, HEAD, POST
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: GET, POST
Access-Control-Allow-Credentials: true
Vary: Accept-Encoding
Content-Length: 172

{"return": "Welcome", "clients": ["local", "local_async", "local_batch", "local_subset
↪", "runner", "runner_async", "sproxy", "sproxy_async", "ssh", "wheel", "wheel_async
↪"]}
```

That means the *salt-sproxy* Salt API is ready to receive requests.

Usage examples:

```
 $ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
     -d eauth='pam' \
     -d username='mircea' \
     -d password='pass' \
     -d client='sproxy' \
     -d tgt='minion1' \
     -d fun='test.ping'
return:
- minion1: true
```

```
$ curl -sS localhost:8080/run -H 'Accept: application/json' \
    -d eauth='pam' \
    -d username='mircea' \
    -d password='pass' \
    -d client='sproxy_async' \
    -d tgt='minion\d' \
    -d tgt_type='pcre' \
    -d fun='test.ping' \
{"return": [{"tag": "salt/run/20200103001109995573", "jid": "20200103001109995573"}]}
```

7.9.2 Before salt-sproxy 2020.2.0

After starting the `salt-api` process, we should get the following:

```
$ curl -i localhost:8080
HTTP/1.1 200 OK
Content-Type: application/json
Server: CherryPy/18.1.1
Date: Wed, 05 Jun 2019 07:58:32 GMT
Allow: GET, HEAD, POST
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: GET, POST
Access-Control-Allow-Credentials: true
Vary: Accept-Encoding
Content-Length: 146

{"return": "Welcome", "clients": ["local", "local_async", "local_batch", "local_subset
↪", "runner", "runner_async", "ssh", "wheel", "wheel_async"]}
```

That means the Salt API is ready to receive requests.

To invoke a command on a (network) device managed through Salt, you can use the `proxy` Runner to invoke commands on, e.g.,

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='pam' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='minion1' \
  -d function='test.ping' \
  -d sync=True
return:
- minion1: true
```

Note that the execution is at the `/run` endpoint, with the following details:

- `username`, `password`, `eauth` as configured in the `external_auth`. See https://docs.saltstack.com/en/latest/topics/eauth/index.html for more details and how to configure external authentication.
- `client` is *runner*, as we're going to use the `proxy` Runner.
- `fun` is the name of the Runner function, in this case `_runners.proxy.execute()`.
- `tgt` is the Minion ID / device name to target.
- `function` is the Salt function to execute on the targeted device(s).
- `sync` is set as `True` as the execution must be synchronous because we're waiting for the output to be returned back over the API. Otherwise, if we only need to invoke the function without expecting an output, we don't need to pass this argument.

This HTTP request is the equivalent of CLI from the example *salt-sproxy 101*:

```
$ salt-sproxy minion1 test.ping
```

It works in the same way when execution function on actual devices, for instance when gathering the ARP table from a Juniper router (the equivalent of the `salt-sproxy juniper-router net.arp` CLI from the example *salt-sproxy with network devices*):

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='pam' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='juniper-router' \
  -d function='net.arp' \
  -d sync=True
return:
- juniper-router:
    comment: ''
    out:
    - age: 891.0
      interface: fxp0.0
      ip: 10.96.0.1
      mac: 92:99:00:0A:00:00
    - age: 1001.0
      interface: fxp0.0
      ip: 10.96.0.13
      mac: 92:99:00:0A:00:00
    - age: 902.0
      interface: em1.0
      ip: 128.0.0.16
      mac: 02:42:AC:12:00:02
    result: true
```

Or when updating the configuration:

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='pam' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='juniper-router' \
  -d function='net.load_config' \
  -d text='set system ntp server 10.10.10.1' \
  -d test=True \
  -d sync=True
return:
- juniper-router:
    already_configured: false
    comment: Configuration discarded.
    diff: '[edit system]
      +   ntp {
      +       server 10.10.10.1;
      +   }'
    loaded_config: ''
    result: true

$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='pam' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
```

(continues on next page)

(continued from previous page)

```
  -d tgt='juniper-router' \
  -d function='net.load_config' \
  -d text='set system ntp server 10.10.10.1' \
  -d sync=True
return:
- juniper-router:
    already_configured: false
    comment: ''
    diff: '[edit system]
      +   ntp {
      +       server 10.10.10.1;
      +   }'
    loaded_config: ''
    result: true
```

You can follow the same methodology with any other Salt function (including States) that you might want to execute against a device, without having a (Proxy) Minion running.

7.9.3 See Also

salt-sapi

New in version 2020.2.0.

`salt-sapi` is a program distributed together with *salt-sproxy*, to ease the usage of the Salt API by providing two additional clients: `sproxy` and `sproxy_async`.

The usage is the exact same as the native `salt-api` entry point, just enhanced with the mentioned clients for the `/run` URI.

See *Using the Salt REST API* or https://salt-sproxy.readthedocs.io/en/latest/salt_api.html for more details and usage examples.

Important: At the time being, `salt-sapi` is simply available as a Python program entry point, without providing the system service files. That said, in order for you to use the *salt-sapi* clients, you wlll need to provide a service file or edit the one you might have for `salt-api` already by configuring the path to `salt-sapi` (run `$ which salt-sapi` to find the installation path), e.g., `ExecStart=/usr/local/bin/salt-sapi`.

Example - start `salt-sapi` in debug mode:

```
$ salt-sapi -l debug
```

See the complete list of options by executing `salt-sapi --help`:

```
$ salt-sapi --help
Usage: salt-sapi [options]
salt-sapi is an enhanced Salt API system that provides additional sproxy and
sproxy_async clients, to simplify the usage of salt-sproxy through the Salt
REST API

Options:
  --version             show program's version number and exit
  -V, --versions-report
                        Show program's dependencies version number and exit.
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -c CONFIG_DIR, --config-dir=CONFIG_DIR
                        Pass in an alternative configuration directory.
                        Default: '/etc/salt'.
  -d, --daemon          Run the salt-sapi as a daemon.
  --pid-file=PIDFILE    Specify the location of the pidfile. Default:
                        '/var/run/salt-sapi.pid'.

  Logging Options:
    Logging options which override any settings defined on the
    configuration files.

    -l LOG_LEVEL, --log-level=LOG_LEVEL
                        Console logging log level. One of 'all', 'garbage',
                        'trace', 'debug', 'profile', 'info', 'warning',
                        'error', 'critical', 'quiet'. Default: 'warning'.
    --log-file=API_LOGFILE
                        Log file path. Default: '/var/log/salt/api'.
    --log-file-level=LOG_LEVEL_LOGFILE
                        Logfile logging log level. One of 'all', 'garbage',
                        'trace', 'debug', 'profile', 'info', 'warning',
                        'error', 'critical', 'quiet'. Default: 'warning'.

You can find additional help about salt-sapi issuing "man salt-sapi" or on
https://salt-sproxy.readthedocs.io and
https://docs.saltstack.com/en/latest/ref/cli/salt-api.html.
```

7.10 Mixed Environments

When running in a mixed environment (you already have (Proxy) Minions running, and you would also like to use the salt-sproxy), it is highly recommended to ensure that salt-sproxy is using the same configuration file as your Master, and the Master is up and running.

Using the `--use-existing-proxy` option on the CLI, or configuring `use_existing_proxy: true` in the Master configuration file, `salt-sproxy` is going to execute the command on the Minions that are connected to this Master (and matching your target), otherwise the command is going to be executed locally.

For example, suppose we have two devices, identified as `minion1` and `minion2`, extending the example *salt-sproxy 101*:

`/srv/salt/pillar/top.sls`:

```
base:
  'minion*':
    - dummy
```

`/srv/salt/pillar/dummy.sls`:

```
proxy:
  proxytype: dummy
```

The Master configuration remains the same:

`/etc/salt/master`:

```
pillar_roots:
  base:
    - /srv/salt/pillar
```

Starting up the Master, and the `minion1` Proxy:

```
# start the Salt Master
$ salt-master -d

# start the Proxy Minion for ``minion1``
$ salt-proxy --proxyid minion1 -d

# accept the key of minion1
$ salt-key -y -a minion1

# check that minion1 is now up and running
$ salt minion1 test.ping
minion1:
    Test
```

In a different terminal window, you can start watching the Salt event bus (and leave it open, as I'm going to reference the events below):

```
$ salt-run state.event pretty=True
# here you will see the events flowing
```

Executing the following command, notice that the execution takes place locally (you can identify using the `proxy/runner` event tag):

```
$ salt-sproxy -L minion1,minion2 test.ping --events
minion1:
    True
minion2:
    True
```

The event bus:

```
20190603145654312094        {
    "_stamp": "2019-06-03T13:56:54.312664",
    "minions": [
        "minion1",
        "minion2"
    ]
}
proxy/runner/20190603145654313680/new       {
    "_stamp": "2019-06-03T13:56:54.314249",
    "arg": [],
    "fun": "test.ping",
    "jid": "20190603145654313680",
    "minions": [
        "minion1",
        "minion2"
    ],
    "tgt": [
        "minion1",
        "minion2"
    ],
```

(continues on next page)

(continued from previous page)

```
    "tgt_type": "list",
    "user": "sudo_mircea"
}
proxy/runner/20190603145654313680/ret/minion1       {
    "_stamp": "2019-06-03T13:56:54.406816",
    "fun": "test.ping",
    "fun_args": [],
    "id": "minion1",
    "jid": "20190603145654313680",
    "return": true,
    "success": true
}
proxy/runner/20190603145654313680/ret/minion2       {
    "_stamp": "2019-06-03T13:56:54.538850",
    "fun": "test.ping",
    "fun_args": [],
    "id": "minion2",
    "jid": "20190603145654313680",
    "return": true,
    "success": true
}
```

As presented in *Event-Driven Automation and Orchestration*, there is one event for the job creating, then one for job start, and one event for each device separately (i.e., `proxy/runner/20190603145654313680/ret/minion1` and `proxy/runner/20190603145654313680/ret/minion2`, respectively).

Now, if we want to execute the same, but use the already running Proxy Minion for `minion1` (started previously), simply pass the `--use-existing-proxy` option:

```
$ salt-sproxy -L minion1,minion2 test.ping --events --use-existing-proxy
minion2:
    True
minion1:
    True
```

In this case, the event bus would look like below:

```
proxy/runner/20190603150335939481/new       {
    "_stamp": "2019-06-03T14:03:35.940128",
    "arg": [],
    "fun": "test.ping",
    "jid": "20190603150335939481",
    "minions": [
        "minion1",
        "minion2"
    ],
    "tgt": [
        "minion1",
        "minion2"
    ],
    "tgt_type": "list",
    "user": "sudo_mircea"
}
salt/job/20190603150335939481/new   {
    "_stamp": "2019-06-03T14:03:36.047971",
    "arg": [],
    "fun": "test.ping",
```

(continues on next page)

(continued from previous page)

```
    "jid": "20190603150335939481",
    "minions": [
        "minion1"
    ],
    "missing": [],
    "tgt": "minion1",
    "tgt_type": "glob",
    "user": "sudo_mircea"
}
salt/job/20190603150335939481/ret/minion1    {
    "_stamp": "2019-06-03T14:03:36.147398",
    "cmd": "_return",
    "fun": "test.ping",
    "fun_args": [],
    "id": "minion1",
    "jid": "20190603150335939481",
    "retcode": 0,
    "return": true,
    "success": true
}
proxy/runner/20190603150335939481/ret/minion2        {
    "_stamp": "2019-06-03T14:03:36.245592",
    "fun": "test.ping",
    "fun_args": [],
    "id": "minion2",
    "jid": "20190603150335939481",
    "return": true,
    "success": true
}
proxy/runner/20190603150335939481/ret/minion1        {
    "_stamp": "2019-06-03T14:03:36.247206",
    "fun": "test.ping",
    "fun_args": [],
    "id": "minion1",
    "jid": "20190603150335939481",
    "return": true,
    "success": true
}
```

In this sequence of events, you can notice that, in addition to the events from the previous example, there are two additional events: `salt/job/20190603150335939481/new` - which is for the job start against the `minion1` Proxy Minion, and `salt/job/20190603150335939481/ret/minion1` - which is the return from the `minion1` Proxy Minion. The presence of the `salt/job` event tags proves that the execution goes through the already existing Proxy Minion.

If you would like to always execute through the available Minions, whenever possible, you can add the following option to the Master configuration file:

```
use_existing_proxy: true
```

7.11 Large Scale Settings

The reference document remains https://docs.saltstack.com/en/latest/topics/tutorials/intro_scale.html with some small differences. Note however that if you're running in *Mixed Environments*, the notes from the *Using Salt at Scale* document must be followed in order to manage a large number of devices (i.e., thousands or tens of thousands).

When running salt-sproxy only - without relying on other existing Minions, it is still highly encouraged to use the batch mode when executing: https://docs.saltstack.com/en/latest/topics/tutorials/intro_scale.html#too-many-minions-returning-at-once Usage example:

```
$ salt-sproxy '*' state.highstate -b 20
```

This will only execute on 20 devices at once, while looping through all the targeted devices.

When running in an environment with a Salt Master running and pushing events on the bus as detailed in *Execution Events*, targeting a large number of devices may lead to a higher density of events which requires to increase the size of the event bus and other specific options, e.g., the ZeroMQ high-water mark and backlog - see https://docs.saltstack.com/en/latest/ref/configuration/master.html#master-large-scale-tuning-settings for more details and options.

7.12 Release Notes

7.12.1 Latest Release

Release 2020.2.0

This is considered the first mature release, with significant improvements around the targeting, new CLI options as well as other improvements and features.

Static Grains

With this release, static Grains can be configured easier for large (or all) groups of devices by having a `grains` section in the Master configuration file, e.g.,

`/etc/salt/master`

```
grains:
  salt:
    role: proxy
```

For more details check out the new section *Managing Static Grains*.

Improved targeting

Targeting mechanisms have been revisited and rewrote almost from scratch, for a better user experience similar to when managing Proxy Minions and executing via the usual *salt* command.

On this occasion, there are two new CLI options added in this release: `--invasive-targeting` and `--preload-targeting`. The reasoning for adding these is that the native *salt-sproxy* targeting highly depends on the data you provide mainly through the *Roster* system (see also *Extension Roster Modules*). Through the Roster interface and other mechanisms, you are able to provide static Grains, which you can use in your targeting expressions. There are situations when you may want to target using more dynamic Grains that you probably don't want to manage statically, which may depend on various attributes retrieved *after* connecting to the device (e.g., hardware model, OS version, etc.). In such case, the `--invasive-targeting` targeting can be helpful as it connects to the device, retrieves these attributes / Grains, then executes the requested command, only on the devices matched by your target.

`--preload-targeting` works in a similar way, with the distinction that it doesn't establish the connection with the remote device, however your target expression depends on number of attributes retrieved from various systems depending on each individual device (or group of devices).

Using `--invasive-targeting` together with `--cache-grains` and / or `--cache-pillar` can speed up the run time when you execute next time (next run would be without `--invasive-targeting`), as the Grains / Pillar data is already available and will be used in determining the targets from your expression.

New Roster module: `file`

Using the new *File Roster*, you can provide the universe of devices *salt-sproxy* can possibly manage through an arbitrary SLS file (therefore this file can be provided in any of the supported format: Jinja+YAML, YAML, JSON, Python, etc. - see the list of available Renderers for more options). The path to this file defaults to `/etc/salt/roster`, or you can override it using the `roster_file` configuration option (or from the command line using `--roster-file`), providing the absolute path.

Example File Roster (as YAML):

`/etc/salt/roster`

```
device1:
  grains:
    site: site1
device2:
  grains:
    site: site2
```

Example File Roster (as Jinja+YAML) - manage 100 device, with a simple Jinja + YAML auto-generated Roster:

`/etc/salt/roster`

```
{%- for i in range(100) %}
device{{ i }}:
  grains:
    site: site1
{%- endfor %}
```

Example File Roster (as JSON):

`/etc/salt/roster`

```
{
  "device1": {
    "grains": {
      "site": "site1"
    }
  },
  "device2": {
    "grains": {
      "site": "site1"
    }
  }
}
```

Using any of these, you'll be able to execute `salt-sproxy -G site:site1 test.ping` (to target all devices that have the `site` Grain set as `site1`) or `salt-sproxy 'device*' test.ping`, etc.

Tip: Remember that being interpreted as an SLS, you can also invoke Salt functions, using the `__salt__` global variable. For example, to retrieve and build the list of devices dynamically using an HTTP query, you can do, e.g.,

```
{%- set ret = __salt__.http.query('https://netbox.live/api/dcim/devices/',␣
↪decode=true) %}
{%- for device in ret.dict.results %}
{{ device.name }}:
  grains:
    site: {{ device.site.slug }}
{%- endfor %}
```

As always, for higher complexity, consider using the pure Python Renderer.

salt-sapi

iIn order to simplify the usage of the REST API calls to devices managed through *salt-sproxy*, beginning with this release, there's an additional program distributed with *salt-sproxy*, `salt-sapi` that leverages the usual Salt API features, and on top, it provides an additional client for *sproxy*.

Note: That means, instead of starting the usual `salt-api`, in order to execute REST calls through *sproxy*, you can start `salt-sapi` instead, using the exact same CLI arguments and configuration options. See *salt-sapi* for further information.

Example call before this release (without *salt-sapi*):

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
  -d eauth='pam' \
  -d username='mircea' \
  -d password='pass' \
  -d client='runner' \
  -d fun='proxy.execute' \
  -d tgt='minion1' \
  -d function='test.ping' \
  -d sync=True
return:
- minion1: true
```

Example call starting with this release (through *salt-sapi*):

```
$ curl -sS localhost:8080/run -H 'Accept: application/x-yaml' \
     -d eauth='pam' \
     -d username='mircea' \
     -d password='pass' \
     -d client='sproxy' \
     -d tgt='minion1' \
     -d fun='test.ping'
return:
- minion1: true
```

Notice in the later call the client invoked is `sproxy`, while the `fun` field points straight to the Execution Function you want to execute (as in opposite to a more convoluted usage of both `fun` and `function` as previously).

See also:

Check out the *salt-sapi* example for configuring and using the *salt-sapi* interface.

New CLI options

New CLI options added in this release, to provide similar functionality to the usual `salt` command:

`--batch-wait`: Wait a specific number of seconds after each batch is done before executing the next one.

`--hide-timeout`: Hide devices that timeout.

`--failhard`: Stop the execution at the first error.

`--progress` / `-p`: Display a progress graph to visually show the execution of the command across the list of devices.

`--summary`: Display a summary of the command execution:

- Total number of devices targeted.
- Number of devices that returned without issues.
- Number of devices that timed out executing the command. See also `-t` or `--timeout` argument to adjust the timeout value.
- Number of devices with errors (i.e., there was an error while executing the command).
- Number of unreachable devices (i.e., couldn't establish the connection with the remote device).

In `-v` / `--verbose` mode, this output is enahnced by displaying the list of devices that did not return / with errors / unreachable.

Example:

```
-------------------------------------------
Summary
-------------------------------------------
# of devices targeted: 10
# of devices returned: 3
# of devices that did not return: 5
# of devices with errors: 0
# of devices unreachable: 2
-------------------------------------------
```

`--show-jid`: Display the JID (Job ID).

`--verbose` / `-v`: Turn on command verbosity, display jid, devices per batch, and detailed summary.

`--pillar-root`: Set a specific directory as the base pillar root.

`--states-dir`: Set a specific directory to search for additional States.

`--module-dirs` / `-m`: Specify one or more directories where to load the extension modules from. Multiple directories can be provided by passing `-m` or `--module-dirs` multiple times.

`--saltenv`: The Salt environment name where to load extension modules and files from.

`--config-dump`: Print the complete salt-sproxy configuration values (with the defaults), in YAML format.

Returners

Using the `--return`, `--return-config`, and `--return-kwargs` new CLI options, you can forward the execution results to various systems such as SQL databases, Slack, Syslog, or NoSQL systems, etc. - see here the list of natively available Returner modules you can use.

7.12.2 Previous Releases

2019.10.0

This release includes several new features:

- Improved the granularity of the options that are loaded from the Roster. As such, this can be used to provide more specific connection parameters per device (or groups of devices). In other words, if you have one of more devices that need a more specific, username / password / port / etc. to establish the connection, you can put those into the Roster.

  The available fields that you can use depend on what the Proxy module of choice requires, see https://docs.saltstack.com/en/latest/ref/proxy/all/index.html and check out the documentation of the Proxy module you're using.

  You can also override the `proxytype` value, to use a different Proxy module per device.

  For example, if you're using the *Pillar Roster*, you would typically have a structure as following:

```
devices:
  - name: device1
  - name: device2
  - name: device3
```

  Where all three devices would be managed, say using the napalm Proxy module.

  Say, if you'd like to change `device2` to be managed using the junos Proxy module instead, you can update the above as:

```
devices:
  - name: device1
  - name: device2
    proxytype: junos
  - name: device3
```

  In a similar way, if you require to authenticate to `device3` using a different username, you can override that as:

```
devices:
  - name: device1
  - name: device2
    proxytype: junos
  - name: device3
    username: test-username
```

  While the examples above are using the *Pillar Roster*, they would work in the same way with other Rosters, such as *Ansible Roster*, etc.

  For a more complete example, make sure to take a look at *Quick Start*.

- Added `--no-connect` command line option, to be able to invoke Salt functions without necessarily establishing the connection with the remote device. See –no-connect for more details.
- New option –test-ping which can be used in combination with –use-existing-proxy to ensure that the existing (Proxy) Minion is alive / usable, before attempting to execute the command; when non-responsive, `salt-sproxy` will try to execute the code locally.
- Starting with this release, when targeting through a Roster, by default, the list of targets determined using your Roster of choice, is going to be cached locally. To deactivate this behaviour and re-compute the target at every

  execution, you can use the –no-target-cache option. This option can also be set in the configuration file as `no_target_cache: false`.

- Two new options –sync-grains and –sync-modules to re-sync the Execution or Grain modules that are delivered with the `salt-sproxy` package or from your own environment.

Important: If you are using the *NetBox Roster*, you might want to keep in mind that in Netbox v2.6 the default view permissions changed, so `salt-sproxy` may not able to get the device list from Netbox by default.

Add `EXEMPT_VIEW_PERMISSIONS = ['*']` to the NetBox `configuration.py` file to change this behavior. See https://github.com/netbox-community/netbox/releases/tag/v2.6.0 for more information.

Python Module Index

_

_modules.netbox, 39
_roster.ansible, 31
_roster.file, 33
_roster.netbox, 33
_roster.pillar, 33
_runners.proxy, 34

Index

Symbols

-batch-wait
  command line option, 66
-cache-grains
  command line option, 66
-cache-pillar
  command line option, 66
-config-dump
  command line option, 65
-events
  command line option, 68
-failhard
  command line option, 66
-file-root
  command line option, 69
-file-roots, -display-file-roots
  command line option, 69
-force-color
  command line option, 71
-hide-timeout
  command line option, 66
-invasive-targeting
  command line option, 65
-log-file-level=LOG_LEVEL_LOGFILE
  command line option, 69
-log-file=LOG_FILE
  command line option, 69
-no-cached-grains
  command line option, 66
-no-cached-pillar
  command line option, 66
-no-color
  command line option, 71
-no-connect
  command line option, 68
-no-grains
  command line option, 66
-no-pillar
  command line option, 66
-no-target-cache
  command line option, 69
-out
  command line option, 70
-out-file-append, -output-file-append
  command line option, 71
-out-file=OUTPUT_FILE,
    -output-file=OUTPUT_FILE
  command line option, 70
-out-indent OUTPUT_INDENT,
    -output-indent OUTPUT_INDENT
  command line option, 70
-pillar-root
  command line option, 69
-preload-targeting
  command line option, 65
-preview-target
  command line option, 67
-roster-file
  command line option, 65
-saltenv
  command line option, 68
-save-file-roots
  command line option, 69
-show-jid
  command line option, 67
-state-output=STATE_OUTPUT,
    -state_output=STATE_OUTPUT
  command line option, 71
-state-verbose=STATE_VERBOSE,
    -state_verbose=STATE_VERBOSE
  command line option, 71
-states-dir
  command line option, 69
-summary
  command line option, 67
-sync
  command line option, 66
-sync-all
  command line option, 68

-sync-grains
  command line option, 67
-sync-modules
  command line option, 67
-sync-roster
  command line option, 67
-test-ping
  command line option, 69
-use-existing-proxy
  command line option, 68
-version
  command line option, 65
-versions-report
  command line option, 65
-E, -pcre
  command line option, 70
-G, -grain
  command line option, 70
-L, -list
  command line option, 70
-N, -nodegroup
  command line option, 70
-P, -grain-pcre
  command line option, 70
-R, -range
  command line option, 70
-b, -batch, -batch-size
  command line option, 66
-c CONFIG_DIR, -config-dir=CONFIG_dir
  command line option, 65
-h, -help
  command line option, 65
-l LOG_LEVEL, -log-level=LOG_LEVEL
  command line option, 69
-m, -module-dirs
  command line option, 69
-p, -progress
  command line option, 66
-r, -roster
  command line option, 65
-s, -static
  command line option, 66
-v, -verbose
  command line option, 67
_modules.netbox (*module*), 39
_roster.ansible (*module*), 31
_roster.file (*module*), 33
_roster.netbox (*module*), 33
_roster.pillar (*module*), 33
_runners.proxy (*module*), 34

C

command line option
  -batch-wait, 66
  -cache-grains, 66
  -cache-pillar, 66
  -config-dump, 65
  -events, 68
  -failhard, 66
  -file-root, 69
  -file-roots, -display-file-roots, 69
  -force-color, 71
  -hide-timeout, 66
  -invasive-targeting, 65
  -log-file-level=LOG_LEVEL_LOGFILE, 69
  -log-file=LOG_FILE, 69
  -no-cached-grains, 66
  -no-cached-pillar, 66
  -no-color, 71
  -no-connect, 68
  -no-grains, 66
  -no-pillar, 66
  -no-target-cache, 69
  -out, 70
  -out-file-append, -output-file-append, 71
  -out-file=OUTPUT_FILE, -output-file=OUTPUT_FILE, 70
  -out-indent OUTPUT_INDENT, -output-indent OUTPUT_INDENT, 70
  -pillar-root, 69
  -preload-targeting, 65
  -preview-target, 67
  -roster-file, 65
  -saltenv, 68
  -save-file-roots, 69
  -show-jid, 67
  -state-output=STATE_OUTPUT, -state_output=STATE_OUTPUT, 71
  -state-verbose=STATE_VERBOSE, -state_verbose=STATE_VERBOSE, 71
  -states-dir, 69
  -summary, 67
  -sync, 66
  -sync-all, 68
  -sync-grains, 67
  -sync-modules, 67
  -sync-roster, 67
  -test-ping, 69
  -use-existing-proxy, 68
  -version, 65
  -versions-report, 65
  -E, -pcre, 70
  -G, -grain, 70
  -L, -list, 70

-N, -nodegroup, 70
-P, -grain-pcre, 70
-R, -range, 70
-b, -batch, -batch-size, 66
-c CONFIG_DIR, -config-dir=CONFIG_dir, 65
-h, -help, 65
-l LOG_LEVEL, -log-level=LOG_LEVEL, 69
-m, -module-dirs, 69
-p, -progress, 66
-r, -roster, 65
-s, -static, 66
-v, -verbose, 67
create_circuit() (*in module _modules.netbox*), 39
create_circuit_provider() (*in module _modules.netbox*), 39
create_circuit_termination() (*in module _modules.netbox*), 40
create_circuit_type() (*in module _modules.netbox*), 40
create_device() (*in module _modules.netbox*), 40
create_device_role() (*in module _modules.netbox*), 40
create_device_type() (*in module _modules.netbox*), 41
create_interface() (*in module _modules.netbox*), 41
create_interface_connection() (*in module _modules.netbox*), 41
create_inventory_item() (*in module _modules.netbox*), 41
create_ipaddress() (*in module _modules.netbox*), 42
create_manufacturer() (*in module _modules.netbox*), 42
create_platform() (*in module _modules.netbox*), 42
create_site() (*in module _modules.netbox*), 42

D

delete_interface() (*in module _modules.netbox*), 42
delete_inventory_item() (*in module _modules.netbox*), 43
delete_ipaddress() (*in module _modules.netbox*), 43

E

execute() (*in module _runners.proxy*), 34
execute_devices() (*in module _runners.proxy*), 36

F

filter_() (*in module _modules.netbox*), 43

G

gen_modules() (*_runners.proxy.SProxyMinion method*), 34
get_() (*in module _modules.netbox*), 43
get_circuit_provider() (*in module _modules.netbox*), 43
get_interfaces() (*in module _modules.netbox*), 44
get_ipaddresses() (*in module _modules.netbox*), 44

M

make_interface_child() (*in module _modules.netbox*), 44
make_interface_lag() (*in module _modules.netbox*), 44

O

openconfig_interfaces() (*in module _modules.netbox*), 45
openconfig_lacp() (*in module _modules.netbox*), 45

S

salt_call() (*in module _runners.proxy*), 37
slugify() (*in module _modules.netbox*), 45
SProxyMinion (*class in _runners.proxy*), 34
StandaloneProxy (*class in _runners.proxy*), 34

T

targets() (*in module _roster.ansible*), 32
targets() (*in module _roster.file*), 33
targets() (*in module _roster.netbox*), 33
targets() (*in module _roster.pillar*), 34

U

update_device() (*in module _modules.netbox*), 45
update_interface() (*in module _modules.netbox*), 45