Extending Salt's capabilities for event-driven network automation and orchestration

Mircea Ulinic

NANOG 76, June 2019
Washington, D.C.

Agenda

- Brief introduction to Salt
- New features in Salt release Fluorine (2019.2.0)
- Tutorial / live demo setup
- Your first custom module
- Cross-calling Salt functions
- Functions for low-level API calls
- Writing custom modules for interacting with the network devices
- Cross-platform module implementation
- Using the extension modules for event-driven orchestration

Brief Introduction to Salt

Salt is an event-driven and data-driven configuration management and orchestration tool.

"In SaltStack, speed isn't a byproduct, it is a design goal. SaltStack was created as an extremely fast, lightweight communication bus to provide the foundation for a remote execution engine. SaltStack now provides orchestration, configuration management, event reactors, cloud provisioning, and more, all built around the SaltStack high-speed communication bus."

https://docs.saltstack.com/en/getstarted/speed.html

Brief Introduction to Salt: Network Automation

NETWORK AUTOMATION: NAPALM

Beginning with 2016.11.0, network automation is included by default in the core of Salt. It is based on a the NAPALM library and provides facilities to manage the configuration and retrieve data from network devices running widely used operating systems such: JunOS, IOS-XR, eOS, IOS, NX-OS etc.

- see the complete list of supported devices.

The connection is established via the `NAPALM proxy`.

In the current release, the following modules were included:

- `NAPALM grains` - Select network devices based on their characteristics
- `NET execution module` - Networking basic features
- `NTP execution module`
- `BGP execution module`
- `Routes execution module`
- `SNMP execution module`
- `Users execution module`
- `Probes execution module`
- `NTP peers management state`
- `SNMP configuration management state`
- `Users management state`

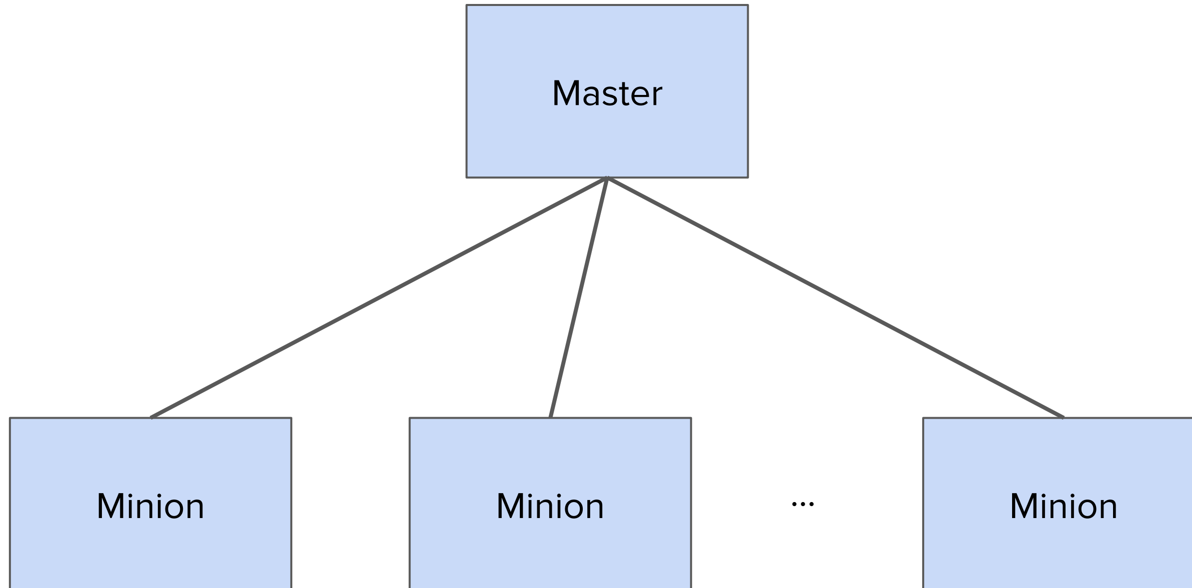
https://docs.saltstack.com/en/develop/topics/releases/2016.11.0.html

Brief Introduction to Salt: Installing Salt

```
$ curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
# ~~~ check the contents of bootstrap-salt.sh ~~~
$ sudo sh bootstrap-salt.sh
```

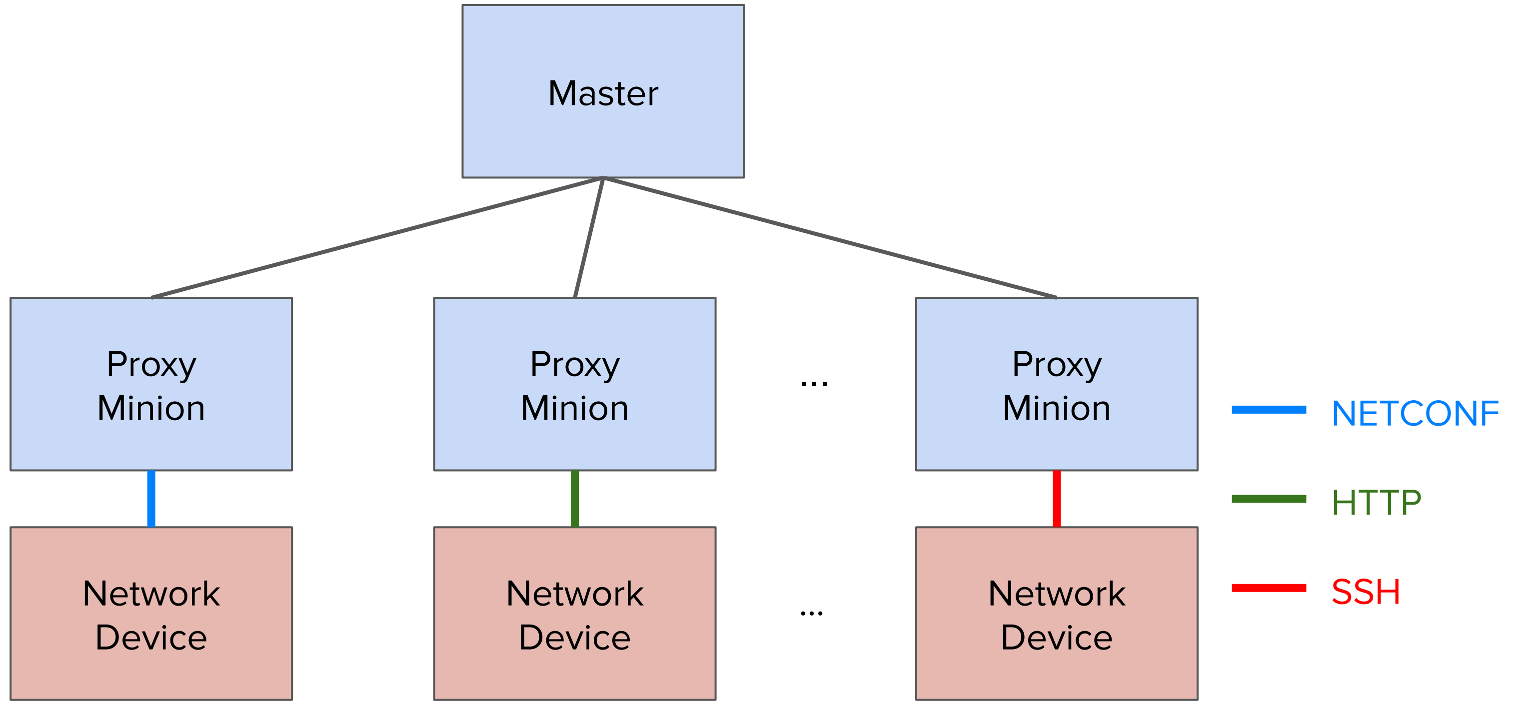
https://github.com/saltstack/salt-bootstrap#install-using-curl

Brief Introduction to Salt: Typical Architecture



https://docs.saltstack.com/en/latest/topics/topology/index.html

Brief Introduction to Salt: Network Automation Topology



https://docs.saltstack.com/en/latest/topics/topology/index.html

Brief Introduction to Salt: Nomenclature

Pillar

Free-form data that can be used to organise configuration values or store sensitive data. This is data managed by the user, in whatever preferred way, including YAML files, JSON files, databases, HTTP API, Excel files, etc. Example of data typically stored in the Pillar: interface configuration, NTP servers, SNMP details, etc.

Grains

Data collected by Salt from the network device (i.e., “what Salt knows about this device”). Can be used to build automation logic, or target devices. Examples of data typically found in the Grains include: device model, serial number, operating system version, etc.

https://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html

Brief Introduction to Salt: file_roots

Salt has its own file server that is used to share files between the Master and the (Proxy) Minions.

`file_roots` is a list of paths from where the Master should serve the files from (and data).

/etc/salt/master

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

https://docs.saltstack.com/en/latest/ref/file_server/

Brief Introduction to Salt: pillar_roots

Similar to the `file_roots` configuration option, `pillar_roots` lists the locations from where Salt should load Pillar data stored as files.

/etc/salt/master

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

https://docs.saltstack.com/en/latest/ref/file_server/

Brief Introduction to Salt: deep dive

- Salt in 10 minutes
- Salt fundamentals
- Configuration management using Salt
- Network automation official docs
- Network automation at scale: up and running in 60 minutes
- Using Salt at scale

New features in Salt release Fluorine (2019.2.0)

NETWORK AUTOMATION

Beginning with this release, Salt provides much broader support for a variety of network operating systems, and features for configuration manipulation or operational command execution.

NETBOX

Added in the previous release, 2018.3.0, the capabilities of the `netbox` Execution Module have been extended, with a much longer list of available features:

- `netbox.create_circuit`
- `netbox.create_circuit_provider`
- `netbox.create_circuit_termination`
- `netbox.create_circuit_type`
- `netbox.create_device`
- `netbox.create_device_role`
- `netbox.create_device_type`
- `netbox.create_interface`
- `netbox.create_interface_connection`
- `netbox.create_inventory_item`
- `netbox.create_ipaddress`
- `netbox.create_manufacturer`
- `netbox.create_platform`
- `netbox.create_site`
- `netbox.delete_interface`
- `netbox.delete_inventory_item`
- `netbox.delete_ipaddress`
- `netbox.get_circuit_provider`
- `netbox.get_interfaces`
- `netbox.get_ipaddresses`
- `netbox.make_interface_child`
- `netbox.make_interface_lag`
- `netbox.openconfig_interfaces`
- `netbox.openconfig_lacp`
- `netbox.update_device`
- `netbox.update_interface`

New features in Salt release Fluorine (2019.2.0)

NETMIKO

Netmiko, the multi-vendor library to simplify Paramiko SSH connections to network devices, is now officially integrated into Salt. The network community can use it via the `netmiko` Proxy Module or directly from any Salt Minions, passing the connection credentials - see the documentation for the `netmiko` Execution Module.

ARISTA

Arista switches can now be managed running under the `pyeapi` Proxy Module, and execute RPC requests via the `pyeapi` Execution Module.

CISCO NEXUS

While support for SSH-based operations has been added in the release codename Carbon (2016.11), the new `nxos_api` Proxy Module and `nxos_api` allow management of Cisco Nexus switches via the NX-API.

It is important to note that these modules don't have third party dependencies, therefore they can be used straight away from any Salt Minion. This also means that the user may be able to install the regular Salt Minion on the Nexus switch directly and manage the network devices like a regular server.

GENERAL-PURPOSE MODULES

The new `ciscoconfparse` Execution Module can be used for basic configuration parsing, audit or validation for a variety of network platforms having Cisco IOS style configuration (one space indentation), as well as brace-delimited configuration style.

The `iosconfig` can be used for various configuration manipulation for Cisco IOS style configuration, such as: `configuration cleanup`, `tree representation of the config`, etc.

New features in Salt release Fluorine (2019.2.0)

NAPALM

COMMIT AT AND COMMIT CONFIRMED

Beginning with this release, NAPALM users are able to execute scheduled commits (broadly known as "commit at") and "commit confirmed" (revert the configuration change unless the user confirms by running another command). These features are available via the `commit_in`, `commit_at`, `revert_in`, or `revert_at` arguments for the `net.load_config` and `net.load_template` execution functions, or `netconfig.managed`.

The counterpart execution functions `net.confirm_commit`, or `net.cancel_commit`, as well as the State functions `netconfig.commit_cancelled`, or `netconfig.commit_confirmed` can be used to confirm or cancel a commit.

Please note that the commit confirmed and commit cancelled functionalities are available for any platform whether the network devices supports the features natively or not. However, be cautious and make sure you read and understand the caveats before using them in production.

New features in Salt release Fluorine (2019.2.0)

NAPALM

COMMIT AT AND COMMIT CONFIRMED

Beginning with this release, NAPALM users are able to execute scheduled commits (broadly known as "commit at") and "commit confirmed" (revert the configuration change unless the user confirms by running another command). These features are available via the `commit_in`, `commit_at`, `revert_in`, or `revert_at` arguments for the `net.load_config` and `net.load_template` execution functions, or `netconfig.managed`.

The counterpart execution functions `net.confirm_commit`, or `net.cancel_commit`, as well as the State functions `netconfig.commit_cancelled`, or `netconfig.commit_confirmed` can be used to confirm or cancel a commit.

Please note that the commit confirmed and commit cancelled functionalities are available for any platform whether the network devices supports the features natively or not. However, be cautious and make sure you read and understand the caveats before using them in production.

New features in Salt release Fluorine (2019.2.0)

JUNOS

The features from the existing `junos` Execution Module are available via the following functions:

- `napalm.junos_cli`: Execute a CLI command and return the output as text or Python dictionary.
- `napalm.junos_rpc`: Execute an RPC request on the remote Junos device, and return the result as a Python dictionary, easy to digest and manipulate.
- `napalm.junos_install_os`: Install the given image on the device.
- `napalm.junos_facts`: The complete list of Junos facts collected by the `junos-eznc` underlying library.

Note

To be able to use these features, you muse ensure that you meet the requirements for the `junos` module. As `junos-eznc` is already a dependency of NAPALM, you will only have to install `jxmlease`.

Usage examples:

BASH

```
salt '*' napalm.junos_cli 'show arp' format=xml
salt '*' napalm.junos_rpc get-interface-information
```

New features in Salt release Fluorine (2019.2.0)

NETMIKO

The features from the newly added `netmiko` Execution Module are available as:

- `napalm.netmiko_commands`: Execute one or more commands to be execute on the remote device, via Netmiko, and return the output as a text.
- `napalm.netmiko_config`: Load a list of configuration command on the remote device, via Netmiko. The commands can equally be loaded from a local or remote path, and passed through Salt's template rendering pipeline (by default using `Jinja` as the template rendering engine).

Usage examples:

BASH

```bash
salt '*' napalm.netmiko_commands 'show version' 'show interfaces'
salt '*' napalm.netmiko_config config_file=https://bit.ly/2sgljCB
```

New features in Salt release Fluorine (2019.2.0)

ARISTA PYEAPI

For various operations and various extension modules, the following features have been added to gate functionality from the `pyeapi` module:

- `napalm.pyeapi_run_commands`: Execute a list of commands on the Arista switch, via the `pyeapi` library.
- `napalm.pyeapi_config`: Configure the Arista switch with the specified commands, via the `pyeapi` Python library. Similarly to `napalm.netmiko_config`, you can use both local and remote files, with or without templating.

Usage examples:

BASH

```bash
salt '*' napalm.pyeapi_run_commands 'show version' 'show interfaces'
salt '*' napalm.pyeapi_config config_file=salt://path/to/template.jinja
```

New features in Salt release Fluorine (2019.2.0)

CISCO NX-API

In the exact same way as above, the user has absolute control by using the following primitives to manage Cisco Nexus switches via the NX-API:

- `napalm.nxos_api_show`: Execute one or more show (non-configuration) commands, and return the output as plain text or Python dictionary.
- `napalm.nxos_api_rpc`: Execute arbitrary RPC requests via the Nexus API.
- `napalm.nxos_api_config`: Configures the Nexus switch with the specified commands, via the NX-API. The commands can be loaded from the command line, or a local or remote file, eventually rendered using the templating engine of choice (default: `jinja`).

Usage examples:

BASH

```bash
salt '*' napalm.nxos_api_show 'show bgp sessions' 'show processes' raw_text=False
```

New features in Salt release Fluorine (2019.2.0)

CISCOCONFPARSE

The following list of function may be handy when manipulating Cisco IOS or Junos style configurations:

- `napalm.config_filter_lines`: Return a list of detailed matches, for the configuration blocks (parent-child relationship) whose parent and children respect the regular expressions provided.
- `napalm.config_find_lines`: Return the configuration lines that match the regular expression provided.
- `napalm.config_lines_w_child`: Return the configuration lines that match a regular expression, having child lines matching the child regular expression.
- `napalm.config_lines_wo_child`: Return the configuration lines that match a regular expression, that don't have child lines matching the child regular expression.

> **Note**
>
> These functions require the `ciscoconfparse` Python library to be installed.

Usage example (find interfaces that are administratively shut down):

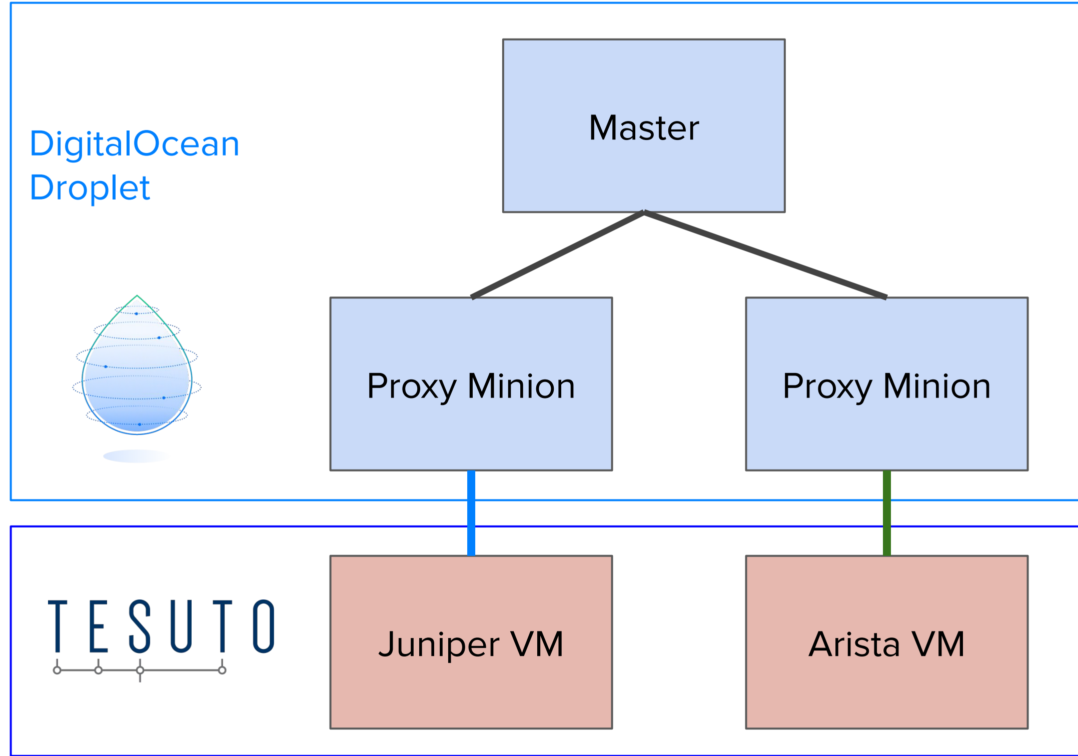
```bash
salt '*' napalm.config_lines_w_child 'interface' 'shutdown'
```

New features in Salt release Fluorine (2019.2.0)

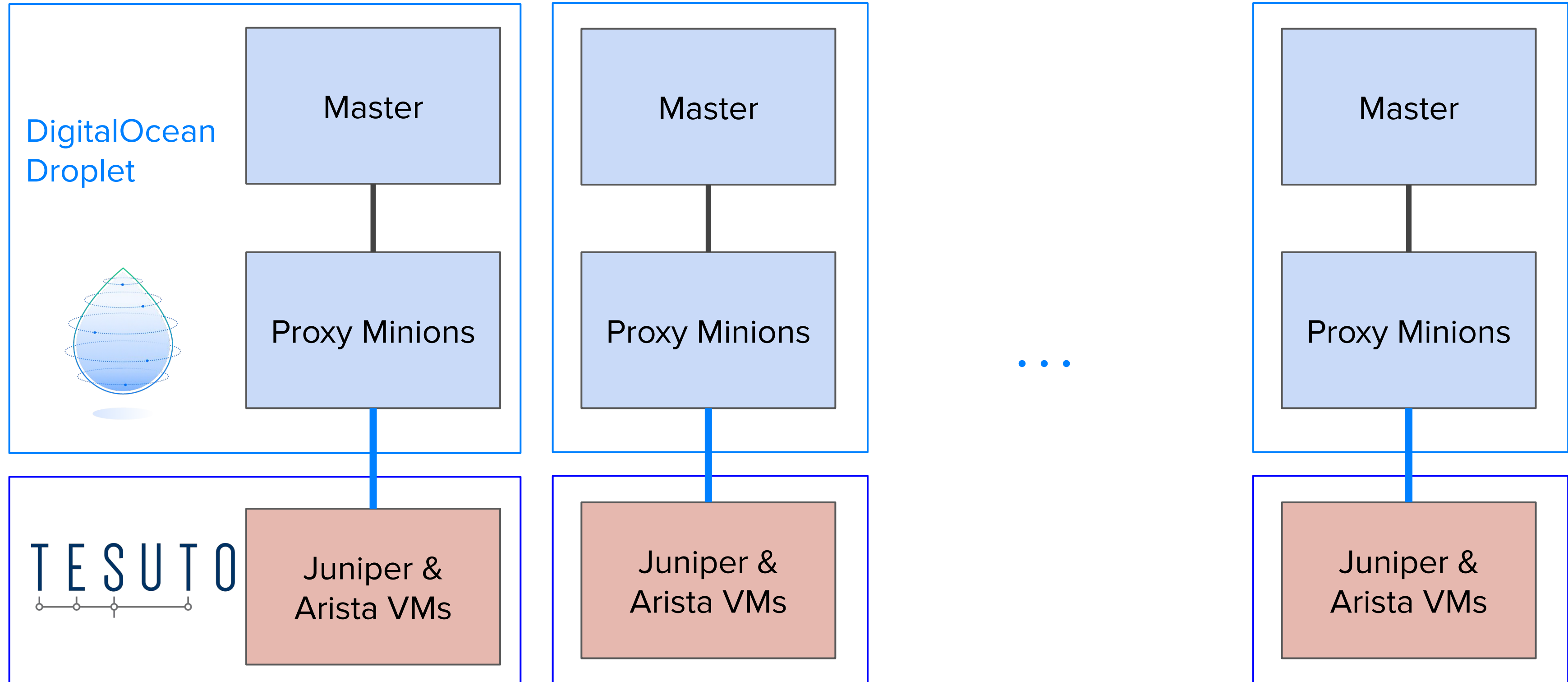
All these new features are important as they allow us to extend Salt's capabilities in our own environment.

Let's see how easily we can write new modules and features.

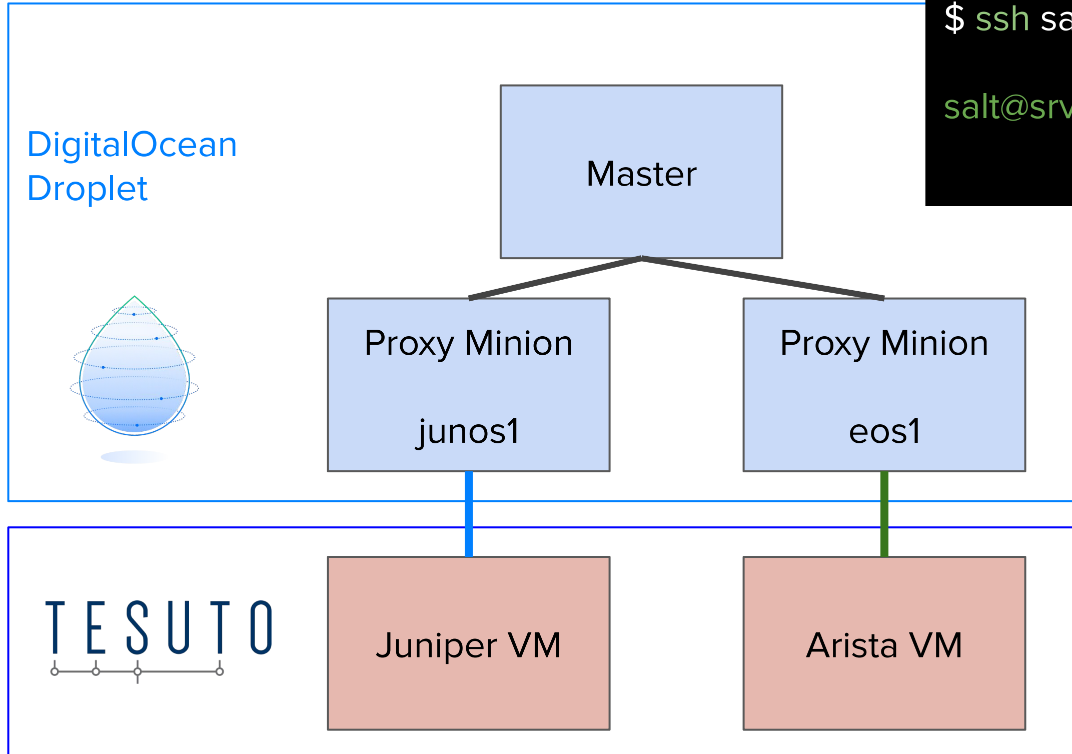
Tutorial / live demo setup (1): my setup



Tutorial / live demo setup (2): your setup



Tutorial / live demo setup (3): your setup




```
$ ssh salt@srv1.automatethe.net

salt@srv1:~$
```

Password: <to be provided live>

Tutorial / live demo setup (4): your setup

```
$ ssh salt@srv1.automatethe.net

salt@srv1:~$ sudo salt -L eos1,junos1 test.ping
eos1:
        True
junos1:
        True
salt@srv1:~$
```

Tutorial / live demo setup (5): your setup

```
$ ssh salt@srv2.automatethe.net

salt@srv2:~$ sudo salt -L eos2,junos2 test.ping
eos2:
    True
junos2:
    True
salt@srv2:~$
```

… and so on to *srv10*.

Tutorial / live demo setup (5)

All the files presented in this tutorial are available on GitHub:
https://github.com/mirceaulinic/nanog76-tutorial

The file paths in the next slides are relative to the repository root, e.g.,
`extmods/_modules/example.py` is this file:
https://github.com/mirceaulinic/nanog76-tutorial/blob/master/extmods/_modules/example.py

Tutorial / live demo setup (6)

The https://github.com/mirceaulinic/nanog76-tutorial git repository is cloned on the *srv<ID>.automatethe.net* Droplets.

/srv/salt is a symlink to the clone path */home/salt/nanog76-tutorial*

```
salt@srv1:~$ ls -la /srv/salt
lrwxrwxrwx 1 root root 27 Jun  6 09:56 /srv/salt -> /home/salt/nanog76-tutorial
salt@srv1:~$ ls -l /home/salt/nanog76-tutorial/
total 32
-rw-r--r-- 1 salt salt  567 Jun  6 11:22 docker-compose.yml
drwxr-xr-x 3 salt salt 4096 Jun  6 11:22 extmods
-rw-r--r-- 1 salt salt 1521 Jun  6 11:22 LICENSE
-rw-r--r-- 1 salt salt   52 Jun  6 11:22 Makefile
-rw-r--r-- 1 salt salt  123 Jun  6 11:22 master
drwxr-xr-x 2 salt salt 4096 Jun  6 13:32 pillar
-rw-r--r-- 1 salt salt  118 Jun  6 11:22 proxy
-rw-r--r-- 1 salt salt  206 Jun  6 11:22 README.rst
```

Tutorial / live demo setup (7)

`/etc/salt/master`

```
open_mode: true

pillar_roots:
  base:
    - /srv/salt/pillar

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

Accept any Minion (not recommended in production).

Equivalent of *`/home/salt/nanog76-tutorial/pillar`* on the *`srv<ID>.automatethe.net`* Droplets.

Equivalent of *`/home/salt/nanog76-tutorial/extmods`* on the *`srv<ID>.automatethe.net`* Droplets.

This is the directory with the extension modules we're going to write shortly.

Tutorial / live demo setup (8): using Docker on your machine

Follow the notes from the README:

https://github.com/mirceaulinic/nanog76-tutorial

```
mircea@master-roshi:~/nanog76-tutorial$ make up
docker-compose up -d
Creating salt-master      ... done
Creating salt-proxy-dummy ... done
```

Tutorial / live demo setup (9): using Docker on your machine

Edit the following files to be able to connect to one of the VMs available:

nanog76-tutorial/pillar/junos_pillar.sls

```
proxy:
  proxytype: napalm
  driver: junos
  host:
junos<ID>.nanog76-demo.digitalocean.cloud.tesuto.com
  username: salt
  password: <to be provided live>
```

Where ID is 1...10 as you may prefer.

Note: the VMs will be available only during the live demo.

Tutorial / live demo setup (9): using Docker on your machine

Edit the following files to be able to connect to one of the VMs available:

nanog76-tutorial/pillar/eos_pillar.sls

```
proxy:
  proxytype: napalm
  driver: eos
  host: eos<ID>.nanog76-demo.digitalocean.cloud.tesuto.com
  username: salt
  password: <to be provided live>
```

Where ID is 1...10 as you may prefer.
Note: the VMs will be available only during the live demo.

Tutorial / live demo setup (9): using Docker on your machine

```
mircea@master-roshi:~/nanog76-tutorial$ make up PROXYID=arista-switch
docker-compose up -d
Creating salt-master                 ... done
Creating salt-proxy-arista-switch ... done
```

```
mircea@master-roshi:~/nanog76-tutorial$ make up PROXYID=juniper-router
docker-compose up -d
salt-master is up-to-date
Recreating salt-proxy-arista-switch ... done
```

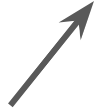
Tutorial / live demo setup (9): using Docker on your machine

```
mircea@master-roshi:~/nanog76-tutorial$ docker exec -ti salt-master bash

root@salt-master:/# salt-key -L
Accepted Keys:
arista-switch
dummy
juniper-router
Denied Keys:
Unaccepted Keys:
Rejected Keys:


root@salt-master:/# salt juniper-router test.ping
juniper-router:
    True
```

Your first extension module

extmods/_modules/example.py

```
def first():
    return True
```



```
$ salt ‘junos-router’
saltutil.sync_modules
junos-router:
    - example
```

```
$ salt ‘junos-router’ example.first
junos-router:
    True
```

Cross-calling Salt functions (1)

extmods/_modules/example.py

```
def second():
    return __salt__['test.false']()

def third():
    return __salt__['example.first']()
```

test.false is a Salt native function that always returns boolean value *False*.

example.first is the previously defined function.

__salt__ is a globally available variable with the mapping to *all* the functions Salt is aware of.

Cross-calling Salt functions (2)

```
$ salt 'junos-router'
saltutil.sync_modules
junos-router:
     - example
```

```
$ salt 'junos-router' example.second
junos-router:
     False
$ salt 'junos-router' example.third
junos-router:
     True
```

Salt functions for low-level API calls

- Junos
  - napalm.junos_rpc
  - napalm.junos_cli
  - napalm.junos_commit
- Arista
  - napalm.pyeapi_run_commands
  - napalm.pyeapi_config
- Cisco Nexus
  - napalm.nxos_api_show
  - napalm.nxos_api_config
- Any platform supported by Netmiko (including the above)
  - napalm.netmiko_commands
  - napalm.netmiko_config

Functions for low-level API calls:

napalm.junos_cli (1)

```
$ salt 'juniper-router' napalm.junos_cli 'show validation statistics'
juniper-router:
    ----------
    message:

        Total RV records: 78242
        Total Replication RV records: 78242
          Prefix entries: 73193
          Origin-AS entries: 78242
        Memory utilization: 15058371 bytes
        Policy origin-validation requests: 0
          Valid: 0
          Invalid: 0
          Unknown: 0
        BGP import policy reevaluation notifications: 1925885
          inet.0, 1763165
          inet6.0, 162720
    out:
        True
```

Functions for low-level API calls:

napalm.junos_cli (2)

```
$ salt 'juniper-router' napalm.junos_cli 'show validation statistics' \
format=xml
juniper-router:
    ----------
    message:
        ----------
        rv-statistics-information:
            ----------
            rv-statistics:
                ----------
                rv-bgp-import-policy-reevaluations:
                    1925885
                rv-bgp-import-policy-rib-name:
                    - inet.0
                    - inet6.0
                rv-bgp-import-policy-rib-reevaluations:
                    - 1763165
                    - 162720
                rv-memory-utilization:
```

Functions for low-level API calls:

`napalm.junos_cli (3)`

```
$ salt 'juniper-router' napalm.junos_cli 'show validation statistics' \
format=xml --out=raw

{'juniper-router': {'message': {'rv-statistics-information':
{'rv-statistics': {'rv-bgp-import-policy-rib-reevaluations': ['1763165',
'162720'], 'rv-policy-origin-validation-requests': '0',
'rv-policy-origin-validation-results-valid': '0', 'rv-origin-as-count':
'78244', 'rv-memory-utilization': '15058761', 'rv-prefix-count': '73195',
'rv-record-count': '78244', 'rv-policy-origin-validation-results-unknown':
'0', 'rv-bgp-import-policy-rib-name': ['inet.0', 'inet6.0'],
'rv-replication-record-count': '78244',
'rv-bgp-import-policy-reevaluations': '1925885',
'rv-policy-origin-validation-results-invalid': '0'}}}, 'out': True}}
```

Functions for low-level API calls: `napalm.junos_rpc (1)`

```
mircea@juniper-router> show validation statistics | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R3/junos">
    <rpc>
        <get-validation-statistics-information >
        </get-validation-statistics-information >
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Functions for low-level API calls: `napalm.junos_rpc` (2)

```
$ salt 'juniper-router' napalm.junos_rpc
get-validation-statistics-information
juniper-router:
    ----------
    comment:
    out:
        ----------
        rv-statistics-information:
            ----------
            rv-statistics:
                ----------
                rv-bgp-import-policy-reevaluations:
                    1925885
                rv-bgp-import-policy-rib-name:
                    - inet.0
                    - inet6.0
                rv-bgp-import-policy-rib-reevaluations:
                    - 1763165
                    - 162720
```

Functions for low-level API calls: `napalm.pyeapi_run_commands`

```
$ salt 'arista-switch' napalm.pyeapi_run_commands 'show version'
arista-switch:
    |_
      ----------
      architecture:
          i386
      bootupTimestamp:
          1534844216.0
      hardwareRevision:
          11.03
      internalVersion:
          4.20.8M-9384033.4208M
      isIntlVersion:
          False
      uptime:
          18767827.61
      version:
          4.20.8M
```

Functions for low-level API calls: napalm.netmiko_commands (1)

```
$ salt 'arista-switch' napalm.netmiko_commands 'show version'
arista-switch:
    - Hardware version:    11.03

      Software image version: 4.20.8M
      Architecture:           i386
      Internal build version: 4.20.8M-9384033.4208M
      Internal build ID:      5c08e74b-ab2b-49fa-bde3-ef7238e2e1ca

      Uptime:                 31 weeks, 0 days, 5 hours and 28 minutes
```

Functions for low-level API calls: `napalm.netmiko_commands` (2)

```
$ salt 'juniper-router' napalm.netmiko_commands 'traceroute monitor
1.1.1.1 summary'
juniper-router:
    -
      HOST: juniper-router                          Loss%    Snt    Last    Avg   Best
Wrst StDev
        1. one.one.one.one                           0.0%     10     0.6    1.0    0.5
4.2   1.1
```

Functions for low-level API calls: napalm.netmiko_config (1)

```
$ sudo salt 'arista-swtich' napalm.netmiko_config 'ntp server 10.10.10.1'
arista-swtich:
    config term
    arista-swtich(config)#ntp server 10.10.10.1
    arista-swtich(config)#end
    arista-swtich#
```

Functions for low-level API calls: `napalm.netmiko_config` (2)

```
$ salt 'juniper-router' napalm.netmiko_config 'set system ntp server
10.10.10.1' commit=True

juniper-router:
    configure
    Entering configuration mode
    The configuration has been changed but not committed

    [edit]
    salt@juniper-router# set system ntp server 10.10.10.1

    [edit]
    salt@juniper-router# commit
    commit complete

    [edit]
    salt@juniper-router#
```

Writing custom modules for network automation (1)

```
mircea@juniper-router> show version | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.3R3/junos">
    <software-information>
        <host-name>router</host-name>
        <product-model>mx480</product-model>
        <product-name>mx480</product-name>
        <junos-version>17.3R3.9</junos-version>
    </software-information>
</rpc-reply>
```

Writing custom modules for network automation (2)

```
$ salt ‘juniper-router’ napalm.junos_cli 'show version' format=xml
--out=raw
{u'juniper-router': {u'message': {u'software-information':
u'host-name': u'juniper-router', u'product-model': u'mx480',
u'product-name': u'mx480', u'junos-version': u'17.3R3.9'}}, u'out':
True}}
```

Writing custom modules for network automation (3)

extmods/_modules/example.py

```
def junos_version():
    ret = __salt__['napalm.junos_cli']('show version', format='xml')
    return ret['message']['software-information']['junos-version']
```

Writing custom modules for network automation (4)

```
$ salt ‘juniper-router’ example.junos_version
juniper-router:
    17.3R3.9
```

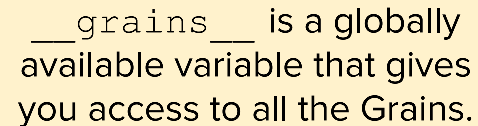
Writing custom modules for network automation: Cross-vendor features (1)

But first, let’s take a look at how we can access Grains data from the execution modules

Remember “Grains represent data collected by Salt from the network device”. One of these is the operating system name.

extmods/_modules/example.py

```
def os_grains():
    return __grains__['os']
```



`__grains__` is a globally available variable that gives you access to all the Grains.

Writing custom modules for network automation: Cross-vendor features (2)

```
$ salt ‘juniper-router’ example.os_grains
juniper-router:
    junos
```

Writing custom modules for network automation: Cross-vendor features (3)

Also remember that “Grains can be used to build automation logic”. For example, invoke the appropriate Salt function depending on the platform.

extmods/_modules/example.py

```
def version():
    if __grains__['os'] == 'junos':
        ret = __salt__['napalm.junos_cli']('show version',
format='xml')
        return
ret['message']['software-information']['junos-version']
    elif __grains__['os'] == 'eos':
        ret = __salt__['napalm.pyeapi_run_commands']('show version')
        return ret[0]['version']
    elif __grains__['os'] == 'nxos':
        ret = __salt__['napalm.nxos_api_rpc']('show version')
        return ret[0]['result']['body']['sys_ver_str']
    raise Exception('Not supported on this platform')
```

Writing custom modules for network automation: Cross-vendor features (3)

Execute this code when running the module against a Juniper device.

extmods/_modules/example.py

```
def version():
    if __grains__['os'] == 'junos':
        ret = __salt__['napalm.junos_cli']('show version',
format='xml')
        return
ret['message']['software-information']['junos-version']
    elif __grains__['os'] == 'eos':
        ret = __salt__['napalm.pyeapi_run_commands']('show version')
        return ret[0]['version']
    elif __grains__['os'] == 'nxos':
        ret = __salt__['napalm.nxos_api_rpc']('show version')
        return ret[0]['result']['body']['sys_ver_str']
    raise Exception('Not supported on this platform')
```

Writing custom modules for network automation: Cross-vendor features (3)

This code when running on an Arista switch. Notice that now it's invoking the *napalm.pyeapi_run_commands* function instead.

extmods/_modules/example.py

```
def version():
    if __grains__['os'] == 'junos':
        ret = __salt__['napalm.junos_cli']('show version',
format='xml')
        return
ret['message']['software-information']['junos-version']
    elif __grains__['os'] == 'eos':
        ret = __salt__['napalm.pyeapi_run_commands']('show version')
        return ret[0]['version']
    elif __grains__['os'] == 'nxos':
        ret = __salt__['napalm.nxos_api_rpc']('show version')
        return ret[0]['result']['body']['sys_ver_str']
    raise Exception('Not supported on this platform')
```

Writing custom modules for network automation: Cross-vendor features (3)

And this code when running on a Cisco Nexus switch.

extmods/_modules/example.py

```
def version():
    if __grains__['os'] == 'junos':
        ret = __salt__['napalm.junos_cli']('show version',
format='xml')
        return
ret['message']['software-information']['junos-version']
    elif __grains__['os'] == 'eos':
        ret = __salt__['napalm.pyeapi_run_commands']('show version')
        return ret[0]['version']
    elif __grains__['os'] == 'nxos':
        ret = __salt__['napalm.nxos_api_rpc']('show version')
        return ret[0]['result']['body']['sys_ver_str']
    raise Exception('Not supported on this platform')
```

Writing custom modules for network automation: Cross-vendor features (3)

And, finally, bail out when running against a different platform that is not Junos, Arista EOS, or Cisco NX-OS.

extmods/_modules/example.py

```
def version():
    if __grains__['os'] == 'junos':
        ret = __salt__['napalm.junos_cli']('show version',
format='xml')
        return
ret['message']['software-information']['junos-version']
    elif __grains__['os'] == 'eos':
        ret = __salt__['napalm.pyeapi_run_commands']('show version')
        return ret[0]['version']
    elif __grains__['os'] == 'nxos':
        ret = __salt__['napalm.nxos_api_rpc']('show version')
        return ret[0]['result']['body']['sys_ver_str']
    raise Exception('Not supported on this platform')
```

Writing custom modules for network automation: Cross-vendor features (4)

```
$ salt ‘juniper-router’ example.version
juniper-router:
    17.3R3.9
$ salt ‘arista-switch’ example.version
arista-switch:
    4.20.8M
$ salt ‘nexus-switch’ example.version
nexus-switch:
    7.0(3)I4(8b)
$ salt ‘hpe-switch’ example.version
hpe-switch:
    The minion function caused an exception: Traceback (most recent call last):
    File "/var/cache/salt/proxy/extmods/hpe-switch/modules/example.py", line 67, in version
    raise Exception(message)
    Exception: Not supported on this platform
```

Writing custom modules for network automation: Separate modules per platform (1)

An alternative to the previous approach is having separate modules (physical files) per platform, yet identified under the same Salt module from the CLI and other Salt subsystems.

Within the automation framework space, Salt is unique in having the possibility to define virtual modules: you can define code in one or more physical files and load them depending on various conditions at runtime. This allows you to have the code physically separated, while preserving the same CLI syntax across multiple platforms. Read more about virtual modules.

This allows us to divide the previous function into simpler ones, in different files, one per platform, e.g.,

- `extmods/_modules/platform_junos.py` for Juniper
- `extmods/_modules/platform_arista.py` for Arista
- `extmods/_modules/platform_nexus.py` for Cisco Nexus

All of them being loaded, and available on the CLI under the same name: `platform`!

Writing custom modules for network automation: Separate modules per platform (2)

extmods/_modules/platform_junos.py

```
def __virtual__():
    if __grains__['os'] == 'junos':
        return 'platform'
    else:
        return (False, 'Not loading this module, as this is not a
Junos device')


def version():
    ret = __salt__['napalm.junos_cli']('show version', format='xml')
    return ret['message']['software-information']['junos-version']
```

Writing custom modules for network automation: Separate modules per platform (2)

extmods/_modules/platform_junos.py

```
def __virtual__():
    if __grains__['os'] == 'junos':
        return 'platform'
    else:
        return (False, 'Not loading this module, as this is not a
Junos device')
```

When the Proxy Minion manages a Juniper device, register the code from this module under the `platform` name. This is the *virtual name*. This is what we're going to use from the CLI and other Salt subsystems.

When the Proxy Minion manages a device that is not Juniper, it won't register this code, making room for another module to be loaded (see next slide).

```
unos_cli']('show version', format='xml')
oftware-information']['junos-version']
```

Writing custom modules for network automation: Separate modules per platform (2)

extmods/_modules/platform_junos.py

```
def __virtual__():
    if __grains__['os'] == '
        return 'platform'
    else:
        return (False, 'Not                                    s not a
Junos device')


def version():
    ret = __salt__['napalm.junos_cli']('show version', format='xml')
    return ret['message']['software-information']['junos-version']
```

You should notice here that this code is the exact one from the previous function `example.junos_version`, as now, having this coded loaded only on Junos devices it ensures that it runs only when needed.

Writing custom modules for network automation: Separate modules per platform (3)

extmods/_modules/platform_arista.py

```
def __virtual__():
    if __grains__['os'] == 'eos':
        return 'platform'
    else:
        return (False, 'Not loading this module, as this is not an
Arista switch’)


def version():
    ret = __salt__['napalm.pyeapi_run_commands'](‘show version')
    return ret[0]['version']
```

Writing custom modules for network automation: Separate modules per platform (3)

extmods/_modules/platform_arista.py

```
def __virtual__():
    if __grains__['os'] == 'eos':
        return 'platform'
    else:
        return (False, 'Not loading this module, as this is not an
Arista switch')


def version():
    ret = __salt__['napalm.pyeapi_run_commands']('show version')
    return ret[0]['version']
```

Similar logic as previously: load this module only when the Proxy Minion manages an Arista switch, and register the code under the `platform` Salt module (virtual name).

Writing custom modules for network automation: Separate modules per platform (4)

extmods/_modules/platform_nexus.py

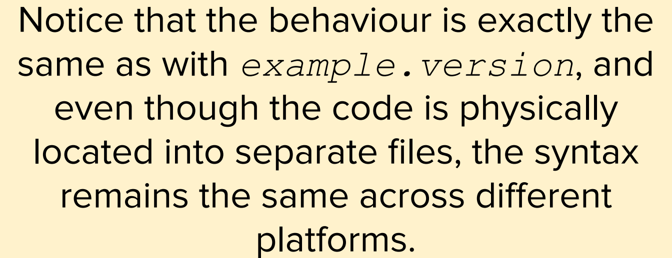
```
def __virtual__():
    if __grains__['os'] == 'nxos':
        return 'platform'
    else:
        return (False, 'Not loading this module, as this is not Cisco
Nexus switch')


def version():
    ret = __salt__['napalm.nxos_api_rpc']('show version')
    return ret[0]['result']['body']['sys_ver_str']
```

Writing custom modules for network automation: Separate modules per platform (5)

```
$ salt ‘juniper-router’ platform.version
juniper-router:
    17.3R3.9
$ salt ‘arista-switch’ platform.version
arista-switch:
    4.20.8M
$ salt ‘nexus-switch’ platform.version
nexus-switch:
    7.0(3)I4(8b)
```

Writing custom modules for network automation: Separate modules per platform (5)

```
$ salt ‘juniper-router’ platform.version
juniper-router:
    17.3R3.9
$ salt ‘arista-switch’ platform.version
arista-switch:
    4.20.8M
$ salt ‘nexus-switch’ platform.version
nexus-switch:
    7.0(3)I4(8b)
```



Using the extension modules for event-driven orchestration

Extension modules defined in your own environment can then be used in the same way as native modules on the CLI, as well as other Salt subsystems, including: the template rendering pipeline, Salt system, Reactor system etc.

Using the extension modules for event-driven orchestration: Jinja templates (1)

templates/example.jinja

```
{%- set os_version = salt.example.version () %}

{%- if grains.os == 'junos' %}
set system host-name junos-{{ os_version }}
{%- elif grains.os == 'eos' %}
hostname eos-{{ os_version }}
{%- endif %}
```

Using the extension modules for event-driven orchestration: Jinja templates (2)

```
$ salt ‘juniper-router’ net.load_template salt://template/example.jinja
juniper-router:
    ----------
    already_configured:
        False
    comment:
        Configuration discarded.
    diff:
        [edit system]
        -  host-name juniper;
        +  host-name junos-18.4R1.8;
    loaded_config:
    result:
        True
```

Using the extension modules for event-driven orchestration: Jinja templates (2)

```
$ salt ‘arista-switch’ net.load_template salt://template/example.jinja
arista-switch:
    ----------
    already_configured:
        False
    comment:
        Configuration discarded.
    diff:
        @@ -4,7 +4,7 @@
         transceiver qsfp default-mode 4x10G
         !
        -hostname vEOS1
        +hostname eos-4.21.1F
         !
         spanning-tree mode mstp
    loaded_config:
    result:
        True
```

Using the extension modules for event-driven orchestration: Custom business logic in response to network events (1)

For example, say that you want to automatically increase the BGP prefix limits when one or more sessions have triggered the threshold (i.e., the neighbour(s) is/are announcing more prefixes than configured).

If you're using napalm-logs and importing network events into the Salt bus (using the napalm_syslog Salt Engine), it's easy to define actions in response to these events, e.g., BGP_PREFIX_THRESH_EXCEEDED.

See, for instance, this blog post, or this presentation for further details and examples.

Using the extension modules for event-driven orchestration: Custom business logic in response to network events (2)

/etc/salt/master

```
reactor:
  - 'napalm/syslog/*/BGP_PREFIX_THRESH_EXCEEDED/*':
    - salt://reactor/update_prefix_limit.sls
```

reactor/update_prefix_limit.sls

```
Update BGP prefix limits:
  local.syslog.update_prefix_limit:
    - tgt: {{ data.host }}
    - arg: {{ data.yang_message }}
```

Invokes a custom execution function named *syslog.update_prefix_limit*, defined by the user in their own environment, with the business logic implemented as required.

Using the extension modules for event-driven orchestration: Custom business logic in response to network events (3)

In words: when a [BGP_PREFIX_THRESH_EXCEEDED]() notification is seen on the event bus, Salt is going to invoke the *`salt://reactor/update_prefix_limit.sls`* Reactor SLS file, which executes, e.g., the Salt function *`syslog.update_prefix_limit`*, defined by the user in their own environment, implementing the business logic required, and other safeguards as the needed.

Thanks to Tesuto for the Juniper and Arista VMs

https://www.tesuto.com/

Thank you!

mu@do.co