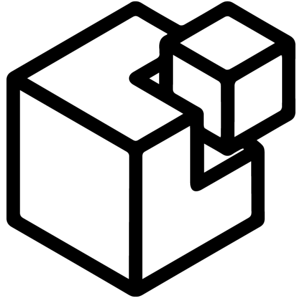


SALTSTACK

Salt Documentation

Release 2018.3.2

SaltStack, Inc.

Oct 22, 2018

Contents

CHAPTER 1

Introduction to Salt

We're not just talking about NaCl.

1.1 The 30 second summary

Salt is:

- a configuration management system, capable of maintaining remote nodes in defined states (for example, ensuring that specific packages are installed and specific services are running)
- a distributed remote execution system used to execute commands and query data on remote nodes, either individually or by arbitrary selection criteria

It was developed in order to bring the best solutions found in the world of remote execution together and make them better, faster, and more malleable. Salt accomplishes this through its ability to handle large loads of information, and not just dozens but hundreds and even thousands of individual servers quickly through a simple and manageable interface.

1.2 Simplicity

Providing versatility between massive scale deployments and smaller systems may seem daunting, but Salt is very simple to set up and maintain, regardless of the size of the project. The architecture of Salt is designed to work with any number of servers, from a handful of local network systems to international deployments across different data centers. The topology is a simple server/client model with the needed functionality built into a single set of daemons. While the default configuration will work with little to no modification, Salt can be fine tuned to meet specific needs.

1.3 Parallel execution

The core functions of Salt:

- enable commands to remote systems to be called in parallel rather than serially
- use a secure and encrypted protocol
- use the smallest and fastest network payloads possible
- provide a simple programming interface

Salt also introduces more granular controls to the realm of remote execution, allowing systems to be targeted not just by hostname, but also by system properties.

1.4 Builds on proven technology

Salt takes advantage of a number of technologies and techniques. The networking layer is built with the excellent ZeroMQ networking library, so the Salt daemon includes a viable and transparent AMQ broker. Salt uses public keys for authentication with the master daemon, then uses faster AES encryption for payload communication; authentication and encryption are integral to Salt. Salt takes advantage of communication via msgpack, enabling fast and light network traffic.

1.5 Python client interface

In order to allow for simple expansion, Salt execution routines can be written as plain Python modules. The data collected from Salt executions can be sent back to the master server, or to any arbitrary program. Salt can be called from a simple Python API, or from the command line, so that Salt can be used to execute one-off commands as well as operate as an integral part of a larger application.

1.6 Fast, flexible, scalable

The result is a system that can execute commands at high speed on target server groups ranging from one to very many servers. Salt is very fast, easy to set up, amazingly malleable and provides a single remote execution architecture that can manage the diverse requirements of any number of servers. The Salt infrastructure brings together the best of the remote execution world, amplifies its capabilities and expands its range, resulting in a system that is as versatile as it is practical, suitable for any network.

1.7 Open

Salt is developed under the Apache 2.0 license, and can be used for open and proprietary projects. Please submit your expansions back to the Salt project so that we can all benefit together as Salt grows. Please feel free to sprinkle Salt around your systems and let the deliciousness come forth.

1.8 Salt Community

Join the Salt!

There are many ways to participate in and communicate with the Salt community.

Salt has an active IRC channel and a mailing list.

1.9 Mailing List

Join the salt-users mailing list. It is the best place to ask questions about Salt and see whats going on with Salt development! The Salt mailing list is hosted by Google Groups. It is open to new members.

1.10 IRC

The #salt IRC channel is hosted on the popular Freenode network. You can use the Freenode webchat client right from your browser.

Logs of the IRC channel activity are being collected courtesy of Moritz Lenz.

If you wish to discuss the development of Salt itself join us in #salt-devel.

1.11 Follow on Github

The Salt code is developed via Github. Follow Salt for constant updates on what is happening in Salt development:

https://github.com/saltstack/salt

1.12 Blogs

SaltStack Inc. keeps a blog with recent news and advancements:

http://www.saltstack.com/blog/

1.13 Example Salt States

The official salt-states repository is: https://github.com/saltstack/salt-states

A few examples of salt states from the community:

- https://github.com/blast-hardcheese/blast-salt-states
- https://github.com/kevingranade/kevingranade-salt-state
- https://github.com/uggedal/states
- https://github.com/mattmcclean/salt-openstack/tree/master/salt
- https://github.com/rentalita/ubuntu-setup/
- https://github.com/brutasse/states
- https://github.com/bclermont/states
- https://github.com/pcrews/salt-data

1.14 Follow on ohloh

https://www.ohloh.net/p/salt

1.15 Other community links

- Salt Stack Inc.
- Subreddit
- Google+
- YouTube
- Facebook
- Twitter
- Wikipedia page

1.16 Hack the Source

If you want to get involved with the development of source code or the documentation efforts, please review the *contributing documentation*!

CHAPTER 2

Installation

This section contains instructions to install Salt. If you are setting up your environment for the first time, you should install a Salt master on a dedicated management server or VM, and then install a Salt minion on each system that you want to manage using Salt. For now you don't need to worry about your *architecture*, you can easily add components and modify your configuration later without needing to reinstall anything.

The general installation process is as follows:

1. Install a Salt master using the instructions for your platform or by running the Salt bootstrap script. If you use the bootstrap script, be sure to include the `-M` option to install the Salt master.
2. Make sure that your Salt minions can *find the Salt master*.
3. Install the Salt minion on each system that you want to manage.
4. Accept the Salt *minion keys* after the Salt minion connects.

After this, you should be able to run a simple command and receive returns from all connected Salt minions.

```
salt '*' test.ping
```

2.1 Quick Install

On most distributions, you can set up a **Salt Minion** with the *Salt bootstrap*.

2.2 Platform-specific Installation Instructions

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

2.2.1 Arch Linux

Installation

Salt (stable) is currently available via the Arch Linux Official repositories. There are currently -git packages available in the Arch User repositories (AUR) as well.

Stable Release

Install Salt stable releases from the Arch Linux Official repositories as follows:

```
pacman -S salt
```

Tracking develop

To install the bleeding edge version of Salt (**may include bugs!**), use the -git package. Installing the -git package as follows:

```
wget https://aur.archlinux.org/packages/sa/salt-git/salt-git.tar.gz
tar xf salt-git.tar.gz
cd salt-git/
makepkg -is
```

Note: yaourt

If a tool such as Yaourt is used, the dependencies will be gathered and built automatically.

The command to install salt using the yaourt tool is:

```
yaourt salt-git
```

Post-installation tasks

systemd

Activate the Salt Master and/or Minion via `systemctl` as follows:

```
systemctl enable salt-master.service
systemctl enable salt-minion.service
```

Start the Master

Once you've completed all of these steps you're ready to start your Salt Master. You should be able to start your Salt Master now using the command seen here:

```
systemctl start salt-master
```

Now go to the *Configuring Salt* page.

2.2.2 Debian GNU/Linux / Raspbian

Debian GNU/Linux distribution and some derivatives such as Raspbian already have included Salt packages to their repositories. However, current stable Debian release contains old outdated Salt releases. It is recommended to use SaltStack repository for Debian as described *below*.

Installation from official Debian and Raspbian repositories is described *here*.

Installation from the Official SaltStack Repository

Packages for Debian 9 (Stretch) and Debian 8 (Jessie) are available in the Official SaltStack repository.

Instructions are at https://repo.saltstack.com/#debian.

Note: Regular security support for Debian 7 ended on April 25th 2016. As a result, 2016.3.1 and 2015.8.10 will be the last Salt releases for which Debian 7 packages are created.

Installation from the Debian / Raspbian Official Repository

The Debian distributions contain mostly old Salt packages built by the Debian Salt Team. You can install Salt components directly from Debian but it is recommended to use the instructions above for the packages from the official Salt repository.

On Jessie there is an option to install Salt minion from Stretch with *python-tornado* dependency from *jessie-backports* repositories.

To install fresh release of Salt minion on Jessie:

1. Add *jessie-backports* and *stretch* repositories:

   Debian:

   ```
   echo 'deb http://httpredir.debian.org/debian jessie-backports main' >> /etc/apt/
   ↪sources.list
   echo 'deb http://httpredir.debian.org/debian stretch main' >> /etc/apt/sources.list
   ```

   Raspbian:

   ```
   echo 'deb http://archive.raspbian.org/raspbian/ stretch main' >> /etc/apt/sources.
   ↪list
   ```

2. Make Jessie a default release:

   ```
   echo 'APT::Default-Release "jessie";' > /etc/apt/apt.conf.d/10apt
   ```

3. Install Salt dependencies:

   Debian:

   ```
   apt-get update
   apt-get install python-zmq python-systemd/jessie-backports python-tornado/jessie-
   ↪backports salt-common/stretch
   ```

   Raspbian:

   ```
   apt-get update
   apt-get install python-zmq python-tornado/stretch salt-common/stretch
   ```

4. Install Salt minion package from Latest Debian Release:

   ```
   apt-get install salt-minion/stretch
   ```

Install Packages

Install the Salt master, minion or other packages from the repository with the *apt-get* command. These examples each install one of Salt components, but more than one package name may be given at a time:

- `apt-get install salt-api`
- `apt-get install salt-cloud`
- `apt-get install salt-master`
- `apt-get install salt-minion`
- `apt-get install salt-ssh`
- `apt-get install salt-syndic`

Post-installation tasks

Now, go to the *Configuring Salt* page.

2.2.3 Arista EOS Salt minion installation guide

The Salt minion for Arista EOS is distributed as a SWIX extension and can be installed directly on the switch. The EOS network operating system is based on old Fedora distributions and the installation of the `salt-minion` requires backports. This SWIX extension contains the necessary backports, together with the Salt basecode.

Note: This SWIX extension has been tested on Arista DCS-7280SE-68-R, running EOS 4.17.5M and vEOS 4.18.3F.

Important Notes

This package is in beta, make sure to test it carefully before running it in production.

If confirmed working correctly, please report and add a note on this page with the platform model and EOS version.

If you want to uninstall this package, please refer to the *uninstalling* section.

Installation from the Official SaltStack Repository

Download the swix package and save it to flash.

```
veos#copy https://salt-eos.netops.life/salt-eos-latest.swix flash:
veos#copy https://salt-eos.netops.life/startup.sh flash:
```

Install the Extension

Copy the Salt package to extension

```
veos#copy flash:salt-eos-latest.swix extension:
```

Install the SWIX

```
veos#extension salt-eos-latest.swix force
```

Verify the installation

```
veos#show extensions | include salt-eos
     salt-eos-2017-07-19.swix      1.0.11/1.fc25        A, F                27
```

Change the Salt master IP address or FQDN, by edit the variable (SALT_MASTER)

```
veos#bash vi /mnt/flash/startup.sh
```

Make sure you enable the eAPI with unix-socket

```
veos(config)#management api http-commands
         protocol unix-socket
         no shutdown
```

Post-installation tasks

Generate Keys and host record and start Salt minion

```
veos#bash
#sudo /mnt/flash/startup.sh
```

`salt-minion` should be running

Copy the installed extensions to boot-extensions

```
veos#copy installed-extensions boot-extensions
```

Apply event-handler to let EOS start salt-minion during boot-up

```
veos(config)#event-handler boot-up-script
   trigger on-boot
   action bash sudo /mnt/flash/startup.sh
```

For more specific installation details of the `salt-minion`, please refer to *Configuring Salt*.

Uninstalling

If you decide to uninstall this package, the following steps are recommended for safety:

1. Remove the extension from boot-extensions

```
veos#bash rm /mnt/flash/boot-extensions
```

2. Remove the extension from extensions folder

```
veos#bash rm /mnt/flash/.extensions/salt-eos-latest.swix
```

2. Remove boot-up script

```
veos(config)#no event-handler boot-up-script
```

Additional Information

This SWIX extension contains the following RPM packages:

```
libsodium-1.0.11-1.fc25.i686.rpm
libstdc++-6.2.1-2.fc25.i686.rpm
openpgm-5.2.122-6.fc24.i686.rpm
python-Jinja2-2.8-0.i686.rpm
python-PyYAML-3.12-0.i686.rpm
python-babel-0.9.6-5.fc18.noarch.rpm
python-backports-1.0-3.fc18.i686.rpm
python-backports-ssl_match_hostname-3.4.0.2-1.fc18.noarch.rpm
python-backports_abc-0.5-0.i686.rpm
python-certifi-2016.9.26-0.i686.rpm
python-chardet-2.0.1-5.fc18.noarch.rpm
python-crypto-1.4.1-1.noarch.rpm
python-crypto-2.6.1-1.fc18.i686.rpm
python-futures-3.1.1-1.noarch.rpm
python-jtextfsm-0.3.1-0.noarch.rpm
python-kitchen-1.1.1-2.fc18.noarch.rpm
python-markupsafe-0.18-1.fc18.i686.rpm
python-msgpack-python-0.4.8-0.i686.rpm
python-napalm-base-0.24.3-1.noarch.rpm
python-napalm-eos-0.6.0-1.noarch.rpm
python-netaddr-0.7.18-0.noarch.rpm
python-pyeapi-0.7.0-0.noarch.rpm
python-salt-2017.7.0_1414_g2fb986f-1.noarch.rpm
python-singledispatch-3.4.0.3-0.i686.rpm
python-six-1.10.0-0.i686.rpm
python-tornado-4.4.2-0.i686.rpm
python-urllib3-1.5-7.fc18.noarch.rpm
python2-zmq-15.3.0-2.fc25.i686.rpm
zeromq-4.1.4-5.fc25.i686.rpm
```

2.2.4 Fedora

Beginning with version 0.9.4, Salt has been available in the primary Fedora repositories and EPEL. It is installable using `yum` or `dnf`, depending on your version of Fedora.

Note: Released versions of Salt starting with `2015.5.2` through `2016.3.2` do not have Fedora packages available though EPEL. To install a version of Salt within this release array, please use SaltStack's Bootstrap Script and use the git method of installing Salt using the version's associated release tag.

Release `2016.3.3` and onward will have packaged versions available via EPEL.

WARNING: Fedora 19 comes with systemd 204. Systemd has known bugs fixed in later revisions that prevent the salt-master from starting reliably or opening the network connections that it needs to. It's not likely that a salt-master will start or run reliably on any distribution that uses systemd version 204 or earlier. Running salt-minions should be OK.

Installation

Salt can be installed using `yum` and is available in the standard Fedora repositories.

Stable Release

Salt is packaged separately for the minion and the master. It is necessary only to install the appropriate package for the role the machine will play. Typically, there will be one master and multiple minions.

```
yum install salt-master
yum install salt-minion
```

Installing from `updates-testing`

When a new Salt release is packaged, it is first admitted into the `updates-testing` repository, before being moved to the stable repo.

To install from `updates-testing`, use the `enablerepo` argument for yum:

```
yum --enablerepo=updates-testing install salt-master
yum --enablerepo=updates-testing install salt-minion
```

Installation Using pip

Since Salt is on PyPI, it can be installed using pip, though most users prefer to install using a package manager.

Installing from pip has a few additional requirements:

- Install the group `Development Tools', `dnf groupinstall 'Development Tools'`
- Install the `zeromq-devel' package if it fails on linking against that afterwards as well.

A pip install does not make the init scripts or the /etc/salt directory, and you will need to provide your own systemd service unit.

Installation from pip:

```
pip install salt
```

> **Warning:** If installing from pip (or from source using `setup.py install`), be advised that the `yum-utils` package is needed for Salt to manage packages. Also, if the Python dependencies are not already installed, then you will need additional libraries/tools installed to build some of them. More information on this can be found *here*.

Post-installation tasks

Master

To have the Master start automatically at boot time:

```
systemctl enable salt-master.service
```

To start the Master:

```
systemctl start salt-master.service
```

Minion

To have the Minion start automatically at boot time:

```
systemctl enable salt-minion.service
```

To start the Minion:

```
systemctl start salt-minion.service
```

Now go to the *Configuring Salt* page.

2.2.5 FreeBSD

Installation

Salt is available in binary package form from both the FreeBSD pkgng repository or directly from SaltStack. The instructions below outline installation via both methods:

FreeBSD repo

The FreeBSD pkgng repository is preconfigured on systems 10.x and above. No configuration is needed to pull from these repositories.

```
pkg install py27-salt
```

These packages are usually available within a few days of upstream release.

SaltStack repo

SaltStack also hosts internal binary builds of the Salt package, available from https://repo.saltstack.com/freebsd/. To make use of this repository, add the following file to your system:

/usr/local/etc/pkg/repos/saltstack.conf:

```
saltstack: {
  url: "https://repo.saltstack.com/freebsd/${ABI}/",
  enabled: yes
}
```

You should now be able to install Salt from this new repository:

```
pkg install py27-salt
```

These packages are usually available earlier than upstream FreeBSD. Also available are release candidates and development releases. Use these pre-release packages with caution.

Post-installation tasks

Master

Copy the sample configuration file:

```
cp /usr/local/etc/salt/master.sample /usr/local/etc/salt/master
```

rc.conf

Activate the Salt Master in `/etc/rc.conf`:

```
sysrc salt_master_enable="YES"
```

Start the Master

Start the Salt Master as follows:

```
service salt_master start
```

Minion

Copy the sample configuration file:

```
cp /usr/local/etc/salt/minion.sample /usr/local/etc/salt/minion
```

rc.conf

Activate the Salt Minion in `/etc/rc.conf`:

```
sysrc salt_minion_enable="YES"
```

Start the Minion

Start the Salt Minion as follows:

```
service salt_minion start
```

Now go to the *Configuring Salt* page.

2.2.6 Gentoo

Salt can be easily installed on Gentoo via Portage:

```
emerge app-admin/salt
```

Post-installation tasks

Now go to the *Configuring Salt* page.

2.2.7 OpenBSD

Salt was added to the OpenBSD ports tree on Aug 10th 2013. It has been tested on OpenBSD 5.5 onwards.

Salt is dependent on the following additional ports. These will be installed as dependencies of the `sysutils/salt` port:

```
devel/py-futures
devel/py-progressbar
net/py-msgpack
net/py-zmq
security/py-crypto
```

```
security/py-M2Crypto
textproc/py-MarkupSafe
textproc/py-yaml
www/py-jinja2
www/py-requests
www/py-tornado
```

Installation

To install Salt from the OpenBSD pkg repo, use the command:

```
pkg_add salt
```

Post-installation tasks

Master

To have the Master start automatically at boot time:

```
rcctl enable salt_master
```

To start the Master:

```
rcctl start salt_master
```

Minion

To have the Minion start automatically at boot time:

```
rcctl enable salt_minion
```

To start the Minion:

```
rcctl start salt_minion
```

Now go to the *Configuring Salt* page.

2.2.8 macOS

Installation from the Official SaltStack Repository

Latest stable build from the selected branch:

The output of `md5 <salt pkg>` should match the contents of the corresponding md5 file.

Earlier builds from supported branches

Archived builds from unsupported branches

Installation from Homebrew

```
brew install saltstack
```

It should be noted that Homebrew explicitly discourages the use of sudo:

> Homebrew is designed to work without using sudo. You can decide to use it but we strongly recommend not to do so. If you have used sudo and run into a bug then it is likely to be the cause. Please don't file a bug report unless you can reproduce it after reinstalling Homebrew from scratch without using sudo

Installation from MacPorts

```
sudo port install salt
```

Installation from Pip

When only using the macOS system's pip, install this way:

```
sudo pip install salt
```

Salt-Master Customizations

Note: Salt master on macOS is not tested or supported by SaltStack. See SaltStack Platform Support for more information.

To run salt-master on macOS, sudo add this configuration option to the /etc/salt/master file:

```
max_open_files: 8192
```

On versions previous to macOS 10.10 (Yosemite), increase the root user maxfiles limit:

```
sudo launchctl limit maxfiles 4096 8192
```

Note: On macOS 10.10 (Yosemite) and higher, maxfiles should not be adjusted. The default limits are sufficient in all but the most extreme scenarios. Overriding these values with the setting below will cause system instability!

Now the salt-master should run without errors:

```
sudo salt-master --log-level=all
```

Post-installation tasks

Now go to the *Configuring Salt* page.

2.2.9 RHEL / CentOS / Scientific Linux / Amazon Linux / Oracle Linux

Salt should work properly with all mainstream derivatives of Red Hat Enterprise Linux, including CentOS, Scientific Linux, Oracle Linux, and Amazon Linux. Report any bugs or issues on the issue tracker.

Installation from the Official SaltStack Repository

Packages for Redhat, CentOS, and Amazon Linux are available in the SaltStack Repository.

- Red Hat / CentOS
- Amazon Linux

Note: As of 2015.8.0, EPEL repository is no longer required for installing on RHEL systems. SaltStack repository provides all needed dependencies.

Warning: If installing on Red Hat Enterprise Linux 7 with disabled (not subscribed on) `RHEL Server Releases' or `RHEL Server Optional Channel' repositories, append CentOS 7 GPG key URL to SaltStack yum repository configuration to install required base packages:

```
[saltstack-repo]
name=SaltStack repo for Red Hat Enterprise Linux $releasever
baseurl=https://repo.saltstack.com/yum/redhat/$releasever/$basearch/latest
enabled=1
gpgcheck=1
gpgkey=https://repo.saltstack.com/yum/redhat/$releasever/$basearch/latest/SALTSTACK-
↪GPG-KEY.pub
       https://repo.saltstack.com/yum/redhat/$releasever/$basearch/latest/base/RPM-
↪GPG-KEY-CentOS-7
```

Note: `systemd` and `systemd-python` are required by Salt, but are not installed by the Red Hat 7 `@base` installation or by the Salt installation. These dependencies might need to be installed before Salt.

Installation from the Community-Maintained Repository

Beginning with version 0.9.4, Salt has been available in EPEL.

Note: Packages in this repository are built by community, and it can take a little while until the latest stable SaltStack release become available.

RHEL/CentOS 6 and 7, Scientific Linux, etc.

Warning: Salt 2015.8 is currently not available in EPEL due to unsatisfied dependencies: `python-crypto` 2.6.1 or higher, and `python-tornado` version 4.2.1 or higher. These packages are not currently available in EPEL for Red Hat Enterprise Linux 6 and 7.

Enabling EPEL

If the EPEL repository is not installed on your system, you can download the RPM for RHEL/CentOS 6 or for RHEL/CentOS 7 and install it using the following command:

```
rpm -Uvh epel-release-X-Y.rpm
```

Replace `epel-release-X-Y.rpm` with the appropriate filename.

Installing Stable Release

Salt is packaged separately for the minion and the master. It is necessary to install only the appropriate package for the role the machine will play. Typically, there will be one master and multiple minions.

- `yum install salt-master`
- `yum install salt-minion`
- `yum install salt-ssh`
- `yum install salt-syndic`
- `yum install salt-cloud`

Installing from `epel-testing`

When a new Salt release is packaged, it is first admitted into the `epel-testing` repository, before being moved to the stable EPEL repository.

To install from `epel-testing`, use the `enablerepo` argument for `yum`:

```
yum --enablerepo=epel-testing install salt-minion
```

Installation Using pip

Since Salt is on PyPI, it can be installed using pip, though most users prefer to install using RPM packages (which can be installed from EPEL).

Installing from pip has a few additional requirements:

- Install the group `Development Tools', `yum groupinstall 'Development Tools'`
- Install the `zeromq-devel' package if it fails on linking against that afterwards as well.

A pip install does not make the init scripts or the /etc/salt directory, and you will need to provide your own systemd service unit.

Installation from pip:

```
pip install salt
```

Warning: If installing from pip (or from source using `setup.py install`), be advised that the `yum-utils` package is needed for Salt to manage packages. Also, if the Python dependencies are not already installed, then you will need additional libraries/tools installed to build some of them. More information on this can be found *here*.

ZeroMQ 4

We recommend using ZeroMQ 4 where available. SaltStack provides ZeroMQ 4.0.5 and `pyzmq` 14.5.0 in the *SaltStack Repository*.

If this repository is added *before* Salt is installed, then installing either `salt-master` or `salt-minion` will automatically pull in ZeroMQ 4.0.5, and additional steps to upgrade ZeroMQ and pyzmq are unnecessary.

Package Management

Salt's interface to *yum* makes heavy use of the **repoquery** utility, from the yum-utils package. This package will be installed as a dependency if salt is installed via EPEL. However, if salt has been installed using pip, or a host is being managed using salt-ssh, then as of version 2014.7.0 yum-utils will be installed automatically to satisfy this dependency.

Post-installation tasks

Master

To have the Master start automatically at boot time:

RHEL/CentOS 5 and 6

```
chkconfig salt-master on
```

RHEL/CentOS 7

```
systemctl enable salt-master.service
```

To start the Master:

RHEL/CentOS 5 and 6

```
service salt-master start
```

RHEL/CentOS 7

```
systemctl start salt-master.service
```

Minion

To have the Minion start automatically at boot time:

RHEL/CentOS 5 and 6

```
chkconfig salt-minion on
```

RHEL/CentOS 7

```
systemctl enable salt-minion.service
```

To start the Minion:

RHEL/CentOS 5 and 6

```
service salt-minion start
```

RHEL/CentOS 7

```
systemctl start salt-minion.service
```

Now go to the *Configuring Salt* page.

2.2.10 Solaris

Salt is known to work on Solaris but community packages are unmaintained.

It is possible to install Salt on Solaris by using *setuptools*.

For example, to install the develop version of salt:

```
git clone https://github.com/saltstack/salt
cd salt
sudo python setup.py install --force
```

Note: SaltStack does offer commercial support for Solaris which includes packages.

2.2.11 Ubuntu

Installation from the Official SaltStack Repository

Packages for Ubuntu 16 (Xenial), Ubuntu 14 (Trusty), and Ubuntu 12 (Precise) are available in the SaltStack repository.

Instructions are at https://repo.saltstack.com/#ubuntu.

Install Packages

Install the Salt master, minion or other packages from the repository with the *apt-get* command. These examples each install one of Salt components, but more than one package name may be given at a time:

- `apt-get install salt-api`
- `apt-get install salt-cloud`
- `apt-get install salt-master`
- `apt-get install salt-minion`
- `apt-get install salt-ssh`
- `apt-get install salt-syndic`

Post-installation tasks

Now go to the *Configuring Salt* page.

2.2.12 Windows

Salt has full support for running the Salt minion on Windows. You must connect Windows Salt minions to a Salt master on a supported operating system to control your Salt Minions.

Many of the standard Salt modules have been ported to work on Windows and many of the Salt States currently work on Windows as well.

Installation from the Official SaltStack Repository

Latest stable build from the selected branch:

The output of `md5sum <salt minion exe>` should match the contents of the corresponding md5 file.

Earlier builds from supported branches

Archived builds from unsupported branches

Note: The installation executable installs dependencies that the Salt minion requires.

The 64bit installer has been tested on Windows 7 64bit and Windows Server 2008R2 64bit. The 32bit installer has been tested on Windows 2008 Server 32bit. Please file a bug report on our GitHub repo if issues for other platforms are found.

There are installers available for Python 2 and Python 3.

The installer will detect previous installations of Salt and ask if you would like to remove them. Clicking OK will remove the Salt binaries and related files but leave any existing config, cache, and PKI information.

Salt Minion Installation

If the system is missing the appropriate version of the Visual C++ Redistributable (vcredist) the user will be prompted to install it. Click `OK` to install the vcredist. Click `Cancel` to abort the installation without making modifications to the system.

If Salt is already installed on the system the user will be prompted to remove the previous installation. Click `OK` to uninstall Salt without removing the configuration, PKI information, or cached files. Click `Cancel` to abort the installation before making any modifications to the system.

After the Welcome and the License Agreement, the installer asks for two bits of information to configure the minion; the master hostname and the minion name. The installer will update the minion config with these options.

If the installer finds an existing minion config file, these fields will be populated with values from the existing config, but they will be grayed out. There will also be a checkbox to use the existing config. If you continue, the existing config will be used. If the checkbox is unchecked, default values are displayed and can be changed. If you continue, the existing config file in `c:\salt\conf` will be removed along with the `c:\salt\conf\minion.d` directory. The values entered will be used with the default config.

The final page allows you to start the minion service and optionally change its startup type. By default, the minion is set to `Automatic`. You can change the minion start type to `Automatic (Delayed Start)` by checking the `Delayed Start' checkbox.

Note: Highstates that require a reboot may fail after reboot because salt continues the highstate before Windows has finished the booting process. This can be fixed by changing the startup type to `Automatic (Delayed Start)'. The drawback is that it may increase the time it takes for the `salt-minion' service to actually start.

The salt-minion service will appear in the Windows Service Manager and can be managed there or from the command line like any other Windows service.

```
sc start salt-minion
net start salt-minion
```

Installation Prerequisites

Most Salt functionality should work just fine right out of the box. A few Salt modules rely on PowerShell. The minimum version of PowerShell required for Salt is version 3. If you intend to work with DSC then Powershell version 5 is the minimum.

Silent Installer Options

The installer can be run silently by providing the /S option at the command line. The installer also accepts the following options for configuring the Salt Minion silently:

Option	Description
/master=	A string value to set the IP address or hostname of the master. Default value is `salt'. You can pass a single master or a comma-separated list of masters. Setting the master will cause the installer to use the default config or a custom config if defined.
/minion-name=	A string value to set the minion name. Default value is `hostname'. Setting the minion name causes the installer to use the default config or a custom config if defined.
/start-minion=	Either a 1 or 0. `1' will start the salt-minion service, `0' will not. Default is to start the service after installation.
/start-minion-delayed	Set the minion start type to Automatic (Delayed Start).
/default-config	Overwrite the existing config if present with the default config for salt. Default is to use the existing config if present. If /master and/or /minion-name is passed, those values will be used to update the new default config.
/custom-config=	A string value specifying the name of a custom config file in the same path as the installer of the full path to a custom config file. If /master and/or /minion-name is passed, those values will be used to update the new custom config.
/S	Runs the installation silently. Uses the above settings or the defaults.
/?	Displays command line help.

Note: /start-service has been deprecated but will continue to function as expected for the time being.

Note: /default-config and /custom-config= will backup an existing config if found. A timestamp and a .bak extension will be added. That includes the minion file and the minion.d directory.

Here are some examples of using the silent installer:

```
# Install the Salt Minion
# Configure the minion and start the service

Salt-Minion-2017.7.1-Py2-AMD64-Setup.exe /S /master=yoursaltmaster /minion-
↪name=yourminionname
```

```
# Install the Salt Minion
# Configure the minion but don't start the minion service

Salt-Minion-2017.7.1-Py3-AMD64-Setup.exe /S /master=yoursaltmaster /minion-
↪name=yourminionname /start-minion=0
```

```
# Install the Salt Minion
# Configure the minion using a custom config and configuring multimaster

Salt-Minion-2017.7.1-Py3-AMD64-Setup.exe /S /custom-config=windows_minion /master=prod_
↪master1,prod_master2
```

Running the Salt Minion on Windows as an Unprivileged User

Notes:

- These instructions were tested with Windows Server 2008 R2
- They are generalizable to any version of Windows that supports a salt-minion

Create the Unprivileged User that the Salt Minion will Run As

1. Click `Start` > `Control Panel` > `User Accounts`.
2. Click `Add or remove user accounts`.
3. Click `Create new account`.
4. Enter `salt-user` (or a name of your preference) in the `New account name` field.
5. Select the `Standard user` radio button.
6. Click the `Create Account` button.
7. Click on the newly created user account.
8. Click the `Create a password` link.
9. In the `New password` and `Confirm new password` fields, provide a password (e.g ``SuperSecretMinionPassword4Me!'').
10. In the `Type a password hint` field, provide appropriate text (e.g. ``My Salt Password'').
11. Click the `Create password` button.
12. Close the `Change an Account` window.

Add the New User to the Access Control List for the Salt Folder

1. In a File Explorer window, browse to the path where Salt is installed (the default path is `C:\Salt`).
2. Right-click on the `Salt` folder and select `Properties`.
3. Click on the `Security` tab.
4. Click the `Edit` button.
5. Click the `Add` button.
6. Type the name of your designated Salt user and click the `OK` button.

7. Check the box to `Allow` the `Modify` permission.
8. Click the `OK` button.
9. Click the `OK` button to close the `Salt Properties` window.

Update the Windows Service User for the `salt-minion` Service

1. Click `Start` > `Administrative Tools` > `Services`.
2. In the Services list, right-click on `salt-minion` and select `Properties`.
3. Click the `Log On` tab.
4. Click the `This account` radio button.
5. Provide the account credentials created in section A.
6. Click the `OK` button.
7. Click the `OK` button to the prompt confirming that the user `has been granted the Log On As A Service right`.
8. Click the `OK` button to the prompt confirming that `The new logon name will not take effect until you stop and restart the service`.
9. Right-Click on `salt-minion` and select `Stop`.
10. Right-Click on `salt-minion` and select `Start`.

Building and Developing on Windows

This document will explain how to set up a development environment for Salt on Windows. The development environment allows you to work with the source code to customize or fix bugs. It will also allow you to build your own installation.

There are several scripts to automate creating a Windows installer as well as setting up an environment that facilitates developing and troubleshooting Salt code. They are located in the `pkg\windows` directory in the Salt repo (here).

Scripts:

Script	Description
`build_env_2.ps1`	A PowerShell script that sets up a Python 2 build environment
`build_env_3.ps1`	A PowerShell script that sets up a Python 3 build environment
`build_pkg.bat`	A batch file that builds a Windows installer based on the contents of the `C:\Python27` directory
`build.bat`	A batch file that fully automates the building of the Windows installer using the above two scripts

Note: The `build.bat` and `build_pkg.bat` scripts both accept a parameter to specify the version of Salt that will be displayed in the Windows installer. If no version is passed, the version will be determined using git.

Both scripts also accept an additional parameter to specify the version of Python to use. The default is 2.

Prerequisite Software

The only prerequisite is Git for Windows.

Create a Build Environment

1. Working Directory

Create a `Salt-Dev` directory on the root of `C:`. This will be our working directory. Navigate to `Salt-Dev` and clone the Salt repo from GitHub.

Open a command line and type:

```
cd \
md Salt-Dev
cd Salt-Dev
git clone https://github.com/saltstack/salt
```

Go into the `salt` directory and checkout the version of salt to work with (2016.3 or higher).

```
cd salt
git checkout 2017.7.2
```

2. Setup the Python Environment

Navigate to the `pkg\windows` directory and execute the **build_env.ps1** PowerShell script.

```
cd pkg\windows
powershell -file build_env_2.ps1
```

Note: You can also do this from Explorer by navigating to the `pkg\windows` directory, right clicking the **build_env_2.ps1** powershell script and selecting **Run with PowerShell**

This will download and install Python 2 with all the dependencies needed to develop and build Salt.

Note: If you get an error or the script fails to run you may need to change the execution policy. Open a powershell window and type the following command:

```
Set-ExecutionPolicy RemoteSigned
```

3. Salt in Editable Mode

Editable mode allows you to more easily modify and test the source code. For more information see the Pip documentation.

Navigate to the root of the `salt` directory and install Salt in editable mode with `pip`

```
cd \Salt-Dev\salt
pip install -e .
```

Note: The `.` is important

Note: If `pip` is not recognized, you may need to restart your shell to get the updated path

Note: If `pip` is still not recognized make sure that the Python Scripts folder is in the System `%PATH%`. (`C:\Python2\Scripts`)

4. Setup Salt Configuration

Salt requires a minion configuration file and a few other directories. The default config file is named `minion` located in `C:\salt\conf`. The easiest way to set this up is to copy the contents of the `salt\pkg\windows\buildenv` directory to `C:\salt`.

```
cd \
md salt
xcopy /s /e \Salt-Dev\salt\pkg\windows\buildenv\* \salt\
```

Now go into the `C:\salt\conf` directory and edit the minion config file named `minion` (no extension). You need to configure the master and id parameters in this file. Edit the following lines:

```
master: <ip or name of your master>
id: <name of your minion>
```

Create a Windows Installer

To create a Windows installer, follow steps 1 and 2 from *Create a Build Environment* above. Then proceed to 3 below:

3. Install Salt

To create the installer for Window we install Salt using Python instead of pip. Navigate to the root `salt` directory and install Salt.

```
cd \Salt-Dev\salt
python setup.py install
```

4. Create the Windows Installer

Navigate to the `pkg\windows` directory and run the `build_pkg.bat` with the build version (2017.7.2) and the Python version as parameters.

```
cd pkg\windows
build_pkg.bat 2017.7.2 2
              ^^^^^^^^ ^
                  |    |
# build version --     |
# python version ------
```

Note: If no version is passed, the `build_pkg.bat` will guess the version number using git. If the python version is not passed, the default is 2.

Creating a Windows Installer: Alternate Method (Easier)

Clone the Salt repo from GitHub into the directory of your choice. We're going to use `Salt-Dev`.

```
cd \
md Salt-Dev
cd Salt-Dev
git clone https://github.com/saltstack/salt
```

Go into the `salt` directory and checkout the version of Salt you want to build.

```
cd salt
git checkout 2017.7.2
```

Then navigate to `pkg\windows` and run the `build.bat` script with the version you're building.

```
cd pkg\windows
build.bat 2017.7.2 3
          ^^^^^^^^ ^
              |    |
# build version    |
# python version --
```

This will install everything needed to build a Windows installer for Salt using Python 3. The binary will be in the `salt\pkg\windows\installer` directory.

Testing the Salt minion

1. Create the directory `C:\salt` (if it doesn't exist already)
2. **Copy the example `conf` and `var` directories from** `pkg\windows\buildenv` into `C:\salt`
3. Edit `C:\salt\conf\minion`

   ```
   master: ipaddress or hostname of your salt-master
   ```

4. Start the salt-minion

   ```
   cd C:\Python27\Scripts
   python salt-minion -l debug
   ```

5. On the salt-master accept the new minion's key

   ```
   sudo salt-key -A
   ```

   This accepts all unaccepted keys. If you're concerned about security just accept the key for this specific minion.

6. Test that your minion is responding

   On the salt-master run:

```
sudo salt '*' test.ping
```

You should get the following response: `{'your minion hostname': True}`

Packages Management Under Windows 2003

Windows Server 2003 and Windows XP have both reached End of Support. Though Salt is not officially supported on operating systems that are EoL, some functionality may continue to work.

On Windows Server 2003, you need to install optional component ``WMI Windows Installer Provider'' to get a full list of installed packages. If you don't have this, salt-minion can't report some installed software.

2.2.13 SUSE

Installation from the Official SaltStack Repository

Packages for SUSE 12 SP1, SUSE 12, SUSE 11, openSUSE 13 and openSUSE Leap 42.1 are available in the SaltStack Repository.

Instructions are at https://repo.saltstack.com/#suse.

Installation from the SUSE Repository

Since openSUSE 13.2, Salt 2014.1.11 is available in the primary repositories. With the release of SUSE manager 3 a new repository setup has been created. The new repo will by systemsmanagement:saltstack, which is the source for newer stable packages. For backward compatibility a linkpackage will be created to the old devel:language:python repo. All development of suse packages will be done in systemsmanagement:saltstack:testing. This will ensure that salt will be in mainline suse repo's, a stable release repo and a testing repo for further enhancements.

Installation

Salt can be installed using `zypper` and is available in the standard openSUSE/SLES repositories.

Stable Release

Salt is packaged separately for the minion and the master. It is necessary only to install the appropriate package for the role the machine will play. Typically, there will be one master and multiple minions.

```
zypper install salt-master
zypper install salt-minion
```

Post-installation tasks openSUSE

Master

To have the Master start automatically at boot time:

```
systemctl enable salt-master.service
```

To start the Master:

```
systemctl start salt-master.service
```

Minion

To have the Minion start automatically at boot time:

```
systemctl enable salt-minion.service
```

To start the Minion:

```
systemctl start salt-minion.service
```

Post-installation tasks SLES

Master

To have the Master start automatically at boot time:

```
chkconfig salt-master on
```

To start the Master:

```
rcsalt-master start
```

Minion

To have the Minion start automatically at boot time:

```
chkconfig salt-minion on
```

To start the Minion:

```
rcsalt-minion start
```

Unstable Release

openSUSE

For openSUSE Tumbleweed run the following as root:

```
zypper addrepo http://download.opensuse.org/repositories/systemsmanagement:/saltstack/
↪openSUSE_Tumbleweed/systemsmanagement:saltstack.repo
zypper refresh
zypper install salt salt-minion salt-master
```

For openSUSE 42.1 Leap run the following as root:

```
zypper addrepo http://download.opensuse.org/repositories/systemsmanagement:/saltstack/
↪openSUSE_Leap_42.1/systemsmanagement:saltstack.repo
zypper refresh
zypper install salt salt-minion salt-master
```

For openSUSE 13.2 run the following as root:

```
zypper addrepo http://download.opensuse.org/repositories/systemsmanagement:/saltstack/
↪openSUSE_13.2/systemsmanagement:saltstack.repo
zypper refresh
zypper install salt salt-minion salt-master
```

SUSE Linux Enterprise

For SLE 12 run the following as root:

```
zypper addrepo http://download.opensuse.org/repositories/systemsmanagement:/saltstack/
↪SLE_12/systemsmanagement:saltstack.repo
zypper refresh
zypper install salt salt-minion salt-master
```

For SLE 11 SP4 run the following as root:

```
zypper addrepo http://download.opensuse.org/repositories/systemsmanagement:/saltstack/
↪SLE_11_SP4/systemsmanagement:saltstack.repo
zypper refresh
zypper install salt salt-minion salt-master
```

Now go to the *Configuring Salt* page.

2.3 Initial Configuration

2.3.1 Configuring Salt

Salt configuration is very simple. The default configuration for the *master* will work for most installations and the only requirement for setting up a *minion* is to set the location of the master in the minion configuration file.

The configuration files will be installed to `/etc/salt` and are named after the respective components, `/etc/salt/master`, and `/etc/salt/minion`.

Master Configuration

By default the Salt master listens on ports 4505 and 4506 on all interfaces (0.0.0.0). To bind Salt to a specific IP, redefine the ``interface'' directive in the master configuration file, typically `/etc/salt/master`, as follows:

```
- #interface: 0.0.0.0
+ interface: 10.0.0.1
```

After updating the configuration file, restart the Salt master. See the *master configuration reference* for more details about other configurable options.

Minion Configuration

Although there are many Salt Minion configuration options, configuring a Salt Minion is very simple. By default a Salt Minion will try to connect to the DNS name ``salt''; if the Minion is able to resolve that name correctly, no configuration is needed.

If the DNS name ``salt'' does not resolve to point to the correct location of the Master, redefine the ``master'' directive in the minion configuration file, typically `/etc/salt/minion`, as follows:

```
- #master: salt
+ master: 10.0.0.1
```

After updating the configuration file, restart the Salt minion. See the *minion configuration reference* for more details about other configurable options.

Proxy Minion Configuration

A proxy minion emulates the behaviour of a regular minion and inherits their options.

Similarly, the configuration file is `/etc/salt/proxy` and the proxy tries to connect to the DNS name ``salt''.

In addition to the regular minion options, there are several proxy-specific - see the *proxy minion configuration reference*.

Running Salt

1. Start the master in the foreground (to daemonize the process, pass the *-d flag*):

```
salt-master
```

2. Start the minion in the foreground (to daemonize the process, pass the *-d flag*):

```
salt-minion
```

Having trouble?

The simplest way to troubleshoot Salt is to run the master and minion in the foreground with *log level* set to `debug`:

```
salt-master --log-level=debug
```

For information on salt's logging system please see the *logging document*.

Run as an unprivileged (non-root) user

To run Salt as another user, set the *user* parameter in the master config file.

Additionally, ownership, and permissions need to be set such that the desired user can read from and write to the following directories (and their subdirectories, where applicable):

- /etc/salt
- /var/cache/salt
- /var/log/salt
- /var/run/salt

More information about running salt as a non-privileged user can be found *here*.

There is also a full *troubleshooting guide* available.

Key Identity

Salt provides commands to validate the identity of your Salt master and Salt minions before the initial key exchange. Validating key identity helps avoid inadvertently connecting to the wrong Salt master, and helps prevent a potential MiTM attack when establishing the initial connection.

Master Key Fingerprint

Print the master key fingerprint by running the following command on the Salt master:

```
salt-key -F master
```

Copy the `master.pub` fingerprint from the *Local Keys* section, and then set this value as the `master_finger` in the minion configuration file. Save the configuration file and then restart the Salt minion.

Minion Key Fingerprint

Run the following command on each Salt minion to view the minion key fingerprint:

```
salt-call --local key.finger
```

Compare this value to the value that is displayed when you run the `salt-key --finger <MINION_ID>` command on the Salt master.

Key Management

Salt uses AES encryption for all communication between the Master and the Minion. This ensures that the commands sent to the Minions cannot be tampered with, and that communication between Master and Minion is authenticated through trusted, accepted keys.

Before commands can be sent to a Minion, its key must be accepted on the Master. Run the `salt-key` command to list the keys known to the Salt Master:

```
[root@master ~]# salt-key -L
Unaccepted Keys:
alpha
bravo
charlie
delta
Accepted Keys:
```

This example shows that the Salt Master is aware of four Minions, but none of the keys has been accepted. To accept the keys and allow the Minions to be controlled by the Master, again use the `salt-key` command:

```
[root@master ~]# salt-key -A
[root@master ~]# salt-key -L
Unaccepted Keys:
Accepted Keys:
alpha
bravo
charlie
delta
```

The `salt-key` command allows for signing keys individually or in bulk. The example above, using `-A` bulk-accepts all pending keys. To accept keys individually use the lowercase of the same option, `-a keyname`.

See also:

salt-key manpage

Sending Commands

Communication between the Master and a Minion may be verified by running the `test.ping` command:

```
[root@master ~]# salt alpha test.ping
alpha:
    True
```

Communication between the Master and all Minions may be tested in a similar way:

```
[root@master ~]# salt '*' test.ping
alpha:
    True
bravo:
    True
charlie:
    True
delta:
    True
```

Each of the Minions should send a `True` response as shown above.

What's Next?

Understanding *targeting* is important. From there, depending on the way you wish to use Salt, you should also proceed to learn about *Remote Execution* and *Configuration Management*.

2.4 Additional Installation Guides

2.4.1 Salt Bootstrap

The Salt Bootstrap script allows for a user to install the Salt Minion or Master on a variety of system distributions and versions. This shell script known as `bootstrap-salt.sh` runs through a series of checks to determine the operating system type and version. It then installs the Salt binaries using the appropriate methods. The Salt Bootstrap script installs the minimum number of packages required to run Salt. This means that in the event you run the bootstrap to install via package, Git will not be installed. Installing the minimum number of packages helps ensure the script stays as lightweight as possible, assuming the user will install any other required packages after the Salt binaries are present on the system. The script source is available on GitHub: https://github.com/saltstack/salt-bootstrap

Supported Operating Systems

Note: In the event you do not see your distribution or version available please review the develop branch on GitHub as it may contain updates that are not present in the stable release: https://github.com/saltstack/salt-bootstrap/tree/develop

Debian and derivatives

- Debian GNU/Linux 7/8
- Linux Mint Debian Edition 1 (based on Debian 8)
- Kali Linux 1.0 (based on Debian 7)

Red Hat family

- Amazon Linux 2012.09/2013.03/2013.09/2014.03/2014.09
- CentOS 5/6/7
- Fedora 17/18/20/21/22
- Oracle Linux 5/6/7
- Red Hat Enterprise Linux 5/6/7
- Scientific Linux 5/6/7

SUSE family

- openSUSE 12/13
- openSUSE Leap 42
- openSUSE Tumbleweed 2015
- SUSE Linux Enterprise Server 11 SP1/11 SP2/11 SP3/12

Ubuntu and derivatives

- Elementary OS 0.2 (based on Ubuntu 12.04)
- Linaro 12.04
- Linux Mint 13/14/16/17
- Trisquel GNU/Linux 6 (based on Ubuntu 12.04)
- Ubuntu 10.x/11.x/12.x/13.x/14.x/15.x/16.x

Other Linux distro

- Arch Linux
- Gentoo

UNIX systems

BSD:

- OpenBSD
- FreeBSD 9/10/11

SunOS:

- SmartOS

Example Usage

If you're looking for the *one-liner* to install Salt, please scroll to the bottom and use the instructions for *Installing via an Insecure One-Liner*

Note: In every two-step example, you would be well-served to examine the downloaded file and examine it to ensure that it does what you expect.

The Salt Bootstrap script has a wide variety of options that can be passed as well as several ways of obtaining the bootstrap script itself.

Note: These examples below show how to bootstrap Salt directly from GitHub or other Git repository. Run the script without any parameters to get latest stable Salt packages for your system from SaltStack corporate repository. See first example in the *Install using wget* section.

Install using curl

Using `curl` to install latest development version from GitHub:

```
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh git develop
```

If you want to install a specific release version (based on the Git tags):

```
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh git v2015.8.8
```

To install a specific branch from a Git fork:

```
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh -g https://github.com/myuser/salt.git git mybranch
```

If all you want is to install a `salt-master` using latest Git:

```
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh -M -N git develop
```

If your host has Internet access only via HTTP proxy:

```
PROXY='http://user:password@myproxy.example.com:3128'
curl -o bootstrap-salt.sh -L -x "$PROXY" https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh -G -H "$PROXY" git
```

Install using wget

Using `wget` to install your distribution's stable packages:

```
wget -O bootstrap-salt.sh https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh
```

Downloading the script from develop branch:

```
wget -O bootstrap-salt.sh https://bootstrap.saltstack.com/develop
sudo sh bootstrap-salt.sh
```

Installing a specific version from git using `wget`:

```
wget -O bootstrap-salt.sh https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh -P git v2015.8.8
```

Note: On the above example we added *-P* which will allow PIP packages to be installed if required but it's not a necessary flag for Git based bootstraps.

Install using Python

If you already have Python installed, `python 2.6`, then it's as easy as:

```
python -m urllib "https://bootstrap.saltstack.com" > bootstrap-salt.sh
sudo sh bootstrap-salt.sh git develop
```

All Python versions should support the following in-line code:

```
python -c 'import urllib; print urllib.urlopen("https://bootstrap.saltstack.com").
↪read()' > bootstrap-salt.sh
sudo sh bootstrap-salt.sh git develop
```

Install using fetch

On a FreeBSD base system you usually don't have either of the above binaries available. You **do** have `fetch` available though:

```
fetch -o bootstrap-salt.sh https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh
```

If you have any SSL issues install `ca_root_nssp`:

```
pkg install ca_root_nssp
```

And either copy the certificates to the place where fetch can find them:

```
cp /usr/local/share/certs/ca-root-nss.crt /etc/ssl/cert.pem
```

Or link them to the right place:

```
ln -s /usr/local/share/certs/ca-root-nss.crt /etc/ssl/cert.pem
```

Installing via an Insecure One-Liner

The following examples illustrate how to install Salt via a one-liner.

Note: Warning! These methods do not involve a verification step and assume that the delivered file is trustworthy.

Any of the example above which use two-lines can be made to run in a single-line configuration with minor modifications.

For example, using `curl` to install your distribution's stable packages:

```
curl -L https://bootstrap.saltstack.com | sudo sh
```

Using `wget` to install your distribution's stable packages:

```
wget -O - https://bootstrap.saltstack.com | sudo sh
```

Installing the latest develop branch of Salt:

```
curl -L https://bootstrap.saltstack.com | sudo sh -s -- git develop
```

Command Line Options

Here's a summary of the command line options:

```
$ sh bootstrap-salt.sh -h

  Installation types:
    - stable              Install latest stable release. This is the default
                          install type
    - stable [branch]     Install latest version on a branch. Only supported
                          for packages available at repo.saltstack.com
    - stable [version]    Install a specific version. Only supported for
                          packages available at repo.saltstack.com
    - daily               Ubuntu specific: configure SaltStack Daily PPA
    - testing             RHEL-family specific: configure EPEL testing repo
    - git                 Install from the head of the develop branch
    - git [ref]           Install from any git ref (such as a branch, tag, or
                          commit)

  Examples:
    - bootstrap-salt.sh
    - bootstrap-salt.sh stable
    - bootstrap-salt.sh stable 2017.7
    - bootstrap-salt.sh stable 2017.7.2
    - bootstrap-salt.sh daily
    - bootstrap-salt.sh testing
    - bootstrap-salt.sh git
```

```
    - bootstrap-salt.sh git 2017.7
    - bootstrap-salt.sh git v2017.7.2
    - bootstrap-salt.sh git 06f249901a2e2f1ed310d58ea3921a129f214358

  Options:
    -h  Display this message
    -v  Display script version
    -n  No colours
    -D  Show debug output
    -c  Temporary configuration directory
    -g  Salt Git repository URL. Default: https://github.com/saltstack/salt.git
    -w  Install packages from downstream package repository rather than
        upstream, saltstack package repository. This is currently only
        implemented for SUSE.
    -k  Temporary directory holding the minion keys which will pre-seed
        the master.
    -s  Sleep time used when waiting for daemons to start, restart and when
        checking for the services running. Default: 3
    -L  Also install salt-cloud and required python-libcloud package
    -M  Also install salt-master
    -S  Also install salt-syndic
    -N  Do not install salt-minion
    -X  Do not start daemons after installation
    -d  Disables checking if Salt services are enabled to start on system boot.
        You can also do this by touching /tmp/disable_salt_checks on the target
        host. Default: ${BS_FALSE}
    -P  Allow pip based installations. On some distributions the required salt
        packages or its dependencies are not available as a package for that
        distribution. Using this flag allows the script to use pip as a last
        resort method. NOTE: This only works for functions which actually
        implement pip based installations.
    -U  If set, fully upgrade the system prior to bootstrapping Salt
    -I  If set, allow insecure connections while downloading any files. For
        example, pass '--no-check-certificate' to 'wget' or '--insecure' to
        'curl'. On Debian and Ubuntu, using this option with -U allows one to obtain
        GnuPG archive keys insecurely if distro has changed release signatures.
    -F  Allow copied files to overwrite existing (config, init.d, etc)
    -K  If set, keep the temporary files in the temporary directories specified
        with -c and -k
    -C  Only run the configuration function. Implies -F (forced overwrite).
        To overwrite Master or Syndic configs, -M or -S, respectively, must
        also be specified. Salt installation will be omitted, but some of the
        dependencies could be installed to write configuration with -j or -J.
    -A  Pass the salt-master DNS name or IP. This will be stored under
        ${BS_SALT_ETC_DIR}/minion.d/99-master-address.conf
    -i  Pass the salt-minion id. This will be stored under
        ${BS_SALT_ETC_DIR}/minion_id
    -p  Extra-package to install while installing Salt dependencies. One package
        per -p flag. You're responsible for providing the proper package name.
    -H  Use the specified HTTP proxy for all download URLs (including https://).
        For example: http://myproxy.example.com:3128
    -Z  Enable additional package repository for newer ZeroMQ
        (only available for RHEL/CentOS/Fedora/Ubuntu based distributions)
    -b  Assume that dependencies are already installed and software sources are
        set up. If git is selected, git tree is still checked out as dependency
        step.
    -f  Force shallow cloning for git installations.
        This may result in an "n/a" in the version number.
```

```
  -l  Disable ssl checks. When passed, switches "https" calls to "http" where
      possible.
  -V  Install Salt into virtualenv
      (only available for Ubuntu based distributions)
  -a  Pip install all Python pkg dependencies for Salt. Requires -V to install
      all pip pkgs into the virtualenv.
      (Only available for Ubuntu based distributions)
  -r  Disable all repository configuration performed by this script. This
      option assumes all necessary repository configuration is already present
      on the system.
  -R  Specify a custom repository URL. Assumes the custom repository URL
      points to a repository that mirrors Salt packages located at
      repo.saltstack.com. The option passed with -R replaces the
      "repo.saltstack.com". If -R is passed, -r is also set. Currently only
      works on CentOS/RHEL and Debian based distributions.
  -J  Replace the Master config file with data passed in as a JSON string. If
      a Master config file is found, a reasonable effort will be made to save
      the file with a ".bak" extension. If used in conjunction with -C or -F,
      no ".bak" file will be created as either of those options will force
      a complete overwrite of the file.
  -j  Replace the Minion config file with data passed in as a JSON string. If
      a Minion config file is found, a reasonable effort will be made to save
      the file with a ".bak" extension. If used in conjunction with -C or -F,
      no ".bak" file will be created as either of those options will force
      a complete overwrite of the file.
  -q  Quiet salt installation from git (setup.py install -q)
  -x  Changes the python version used to install a git version of salt. Currently
      this is considered experimental and has only been tested on Centos 6. This
      only works for git installations.
  -y  Installs a different python version on host. Currently this has only been
      tested with Centos 6 and is considered experimental. This will install the
      ius repo on the box if disable repo is false. This must be used in conjunction
      with -x <pythonversion>.  For example:
          sh bootstrap.sh -P -y -x python2.7 git v2016.11.3
      The above will install python27 and install the git version of salt using the
      python2.7 executable. This only works for git and pip installations.
```

2.4.2 Opening the Firewall up for Salt

The Salt master communicates with the minions using an AES-encrypted ZeroMQ connection. These communications are done over TCP ports **4505** and **4506**, which need to be accessible on the master only. This document outlines suggested firewall rules for allowing these incoming connections to the master.

Note: No firewall configuration needs to be done on Salt minions. These changes refer to the master only.

Fedora 18 and beyond / RHEL 7 / CentOS 7

Starting with Fedora 18 FirewallD is the tool that is used to dynamically manage the firewall rules on a host. It has support for IPv4/6 settings and the separation of runtime and permanent configurations. To interact with FirewallD use the command line client `firewall-cmd`.

firewall-cmd example:

```
firewall-cmd --permanent --zone=<zone> --add-port=4505-4506/tcp
```

Please choose the desired zone according to your setup. Don't forget to reload after you made your changes.

```
firewall-cmd --reload
```

RHEL 6 / CentOS 6

The `lokkit` command packaged with some Linux distributions makes opening iptables firewall ports very simple via the command line. Just be careful to not lock out access to the server by neglecting to open the ssh port.

lokkit example:

```
lokkit -p 22:tcp -p 4505:tcp -p 4506:tcp
```

The `system-config-firewall-tui` command provides a text-based interface to modifying the firewall.

system-config-firewall-tui:

```
system-config-firewall-tui
```

openSUSE

Salt installs firewall rules in /etc/sysconfig/SuSEfirewall2.d/services/salt. Enable with:

```
SuSEfirewall2 open
SuSEfirewall2 start
```

If you have an older package of Salt where the above configuration file is not included, the `SuSEfirewall2` command makes opening iptables firewall ports very simple via the command line.

SuSEfirewall example:

```
SuSEfirewall2 open EXT TCP 4505
SuSEfirewall2 open EXT TCP 4506
```

The firewall module in YaST2 provides a text-based interface to modifying the firewall.

YaST2:

```
yast2 firewall
```

Windows

Windows Firewall is the default component of Microsoft Windows that provides firewalling and packet filtering. There are many 3rd party firewalls available for Windows, some of which use rules from the Windows Firewall. If you are experiencing problems see the vendor's specific documentation for opening the required ports.

The Windows Firewall can be configured using the Windows Interface or from the command line.

Windows Firewall (interface):

1. Open the Windows Firewall Interface by typing `wf.msc` at the command prompt or in a run dialog (*Windows Key + R*)
2. Navigate to **Inbound Rules** in the console tree

3. Add a new rule by clicking **New Rule…** in the Actions area
4. Change the Rule Type to **Port**. Click **Next**
5. Set the Protocol to **TCP** and specify local ports **4505-4506**. Click **Next**
6. Set the Action to **Allow the connection**. Click **Next**
7. Apply the rule to **Domain**, **Private**, and **Public**. Click **Next**
8. Give the new rule a Name, ie: **Salt**. You may also add a description. Click **Finish**

Windows Firewall (command line):

The Windows Firewall rule can be created by issuing a single command. Run the following command from the command line or a run prompt:

```
netsh advfirewall firewall add rule name="Salt" dir=in action=allow protocol=TCP
↪localport=4505-4506
```

iptables

Different Linux distributions store their *iptables* (also known as netfilter) rules in different places, which makes it difficult to standardize firewall documentation. Included are some of the more common locations, but your mileage may vary.

Fedora / RHEL / CentOS:

```
/etc/sysconfig/iptables
```

Arch Linux:

```
/etc/iptables/iptables.rules
```

Debian

Follow these instructions: https://wiki.debian.org/iptables

Once you've found your firewall rules, you'll need to add the two lines below to allow traffic on `tcp/4505` and `tcp/4506`:

```
-A INPUT -m state --state new -m tcp -p tcp --dport 4505 -j ACCEPT
-A INPUT -m state --state new -m tcp -p tcp --dport 4506 -j ACCEPT
```

Ubuntu

Salt installs firewall rules in /etc/ufw/applications.d/salt.ufw. Enable with:

```
ufw allow salt
```

pf.conf

The BSD-family of operating systems uses packet filter (pf). The following example describes the additions to `pf.conf` needed to access the Salt master.

```
pass in on $int_if proto tcp from any to $int_if port 4505
pass in on $int_if proto tcp from any to $int_if port 4506
```

Once these additions have been made to the pf.conf the rules will need to be reloaded. This can be done using the pfctl command.

```
pfctl -vf /etc/pf.conf
```

2.4.3 Whitelist communication to Master

There are situations where you want to selectively allow Minion traffic from specific hosts or networks into your Salt Master. The first scenario which comes to mind is to prevent unwanted traffic to your Master out of security concerns, but another scenario is to handle Minion upgrades when there are backwards incompatible changes between the installed Salt versions in your environment.

Here is an example *Linux iptables* ruleset to be set on the Master:

```
# Allow Minions from these networks
-I INPUT -s 10.1.2.0/24 -p tcp -m multiport --dports 4505,4506 -j ACCEPT
-I INPUT -s 10.1.3.0/24 -p tcp -m multiport --dports 4505,4506 -j ACCEPT
# Allow Salt to communicate with Master on the loopback interface
-A INPUT -i lo -p tcp -m multiport --dports 4505,4506 -j ACCEPT
# Reject everything else
-A INPUT -p tcp -m multiport --dports 4505,4506 -j REJECT
```

Note: The important thing to note here is that the salt command needs to communicate with the listening network socket of salt-master on the *loopback* interface. Without this you will see no outgoing Salt traffic from the master, even for a simple salt '*' test.ping, because the salt client never reached the salt-master to tell it to carry out the execution.

2.4.4 Preseed Minion with Accepted Key

In some situations, it is not convenient to wait for a minion to start before accepting its key on the master. For instance, you may want the minion to bootstrap itself as soon as it comes online. You may also want to let your developers provision new development machines on the fly.

See also:

Many ways to preseed minion keys

Salt has other ways to generate and pre-accept minion keys in addition to the manual steps outlined below.

salt-cloud performs these same steps automatically when new cloud VMs are created (unless instructed not to).

salt-api exposes an HTTP call to Salt's REST API to *generate and download the new minion keys as a tarball*.

There is a general four step process to do this:

1. Generate the keys on the master:

```
root@saltmaster# salt-key --gen-keys=[key_name]
```

Pick a name for the key, such as the minion's id.

2. Add the public key to the accepted minion folder:

```
root@saltmaster# cp key_name.pub /etc/salt/pki/master/minions/[minion_id]
```

It is necessary that the public key file has the same name as your minion id. This is how Salt matches minions with their keys. Also note that the pki folder could be in a different location, depending on your OS or if specified in the master config file.

3. Distribute the minion keys.

There is no single method to get the keypair to your minion. The difficulty is finding a distribution method which is secure. For Amazon EC2 only, an AWS best practice is to use IAM Roles to pass credentials. (See blog post, http://blogs.aws.amazon.com/security/post/Tx610S2MLVZWEA/Using-IAM-roles-to-distribute-non-AWS-credentials-to-your-EC2-instances)

Security Warning

Since the minion key is already accepted on the master, distributing the private key poses a potential security risk. A malicious party will have access to your entire state tree and other sensitive data if they gain access to a preseeded minion key.

---

4. Preseed the Minion with the keys

You will want to place the minion keys before starting the salt-minion daemon:

```
/etc/salt/pki/minion/minion.pem
/etc/salt/pki/minion/minion.pub
```

Once in place, you should be able to start salt-minion and run `salt-call state.apply` or any other salt commands that require master authentication.

2.4.5 The macOS (Maverick) Developer Step By Step Guide To Salt Installation

This document provides a step-by-step guide to installing a Salt cluster consisting of one master, and one minion running on a local VM hosted on macOS.

Note: This guide is aimed at developers who wish to run Salt in a virtual machine. The official (Linux) walkthrough can be found here.

The 5 Cent Salt Intro

Since you're here you've probably already heard about Salt, so you already know Salt lets you configure and run commands on hordes of servers easily. Here's a brief overview of a Salt cluster:

- Salt works by having a ``master'' server sending commands to one or multiple ``minion'' servers. The master server is the ``command center''. It is going to be the place where you store your configuration files, aka: ``which server is the db, which is the web server, and what libraries and software they should have installed''. The minions receive orders from the master. Minions are the servers actually performing work for your business.
- Salt has two types of configuration files:

  1. the ``salt communication channels'' or ``meta'' or ``config'' configuration files (not official names): one for the master (usually is /etc/salt/master , **on the master server**), and one for minions (default is /etc/salt/minion or /etc/salt/minion.conf, **on the minion servers**). Those files are used to determine things like the Salt Master IP, port, Salt folder locations, etc.. If these are configured incorrectly, your minions will probably be unable to receive orders from the master, or the master will not know which software a given minion should install.

2. the ``business'' or ``service'' configuration files (once again, not an official name): these are configuration files, ending with ".sls" extension, that describe which software should run on which server, along with particular configuration properties for the software that is being installed. These files should be created in the /srv/salt folder by default, but their location can be changed using … /etc/salt/master configuration file!

Note: This tutorial contains a third important configuration file, not to be confused with the previous two: the virtual machine provisioning configuration file. This in itself is not specifically tied to Salt, but it also contains some Salt configuration. More on that in step 3. Also note that all configuration files are YAML files. So indentation matters.

Note: Salt also works with ``masterless'' configuration where a minion is autonomous (in which case salt can be seen as a local configuration tool), or in ``multiple master'' configuration. See the documentation for more on that.

Before Digging In, The Architecture Of The Salt Cluster

Salt Master

The ``Salt master'' server is going to be the Mac OS machine, directly. Commands will be run from a terminal app, so Salt will need to be installed on the Mac. This is going to be more convenient for toying around with configuration files.

Salt Minion

We'll only have one ``Salt minion'' server. It is going to be running on a Virtual Machine running on the Mac, using VirtualBox. It will run an Ubuntu distribution.

Step 1 - Configuring The Salt Master On Your Mac

Official Documentation

Because Salt has a lot of dependencies that are not built in macOS, we will use Homebrew to install Salt. Homebrew is a package manager for Mac, it's great, use it (for this tutorial at least!). Some people spend a lot of time installing libs by hand to better understand dependencies, and then realize how useful a package manager is once they're configuring a brand new machine and have to do it all over again. It also lets you *uninstall* things easily.

Note: Brew is a Ruby program (Ruby is installed by default with your Mac). Brew downloads, compiles, and links software. The linking phase is when compiled software is deployed on your machine. It may conflict with manually installed software, especially in the /usr/local directory. It's ok, remove the manually installed version then refresh the link by typing `brew link 'packageName'`. Brew has a `brew doctor` command that can help you troubleshoot. It's a great command, use it often. Brew requires xcode command line tools. When you run brew the first time it asks you to install them if they're not already on your system. Brew installs software in /usr/local/bin (system bins are in /usr/bin). In order to use those bins you need your $PATH to search there first. Brew tells you if your $PATH needs to be fixed.

Tip: Use the keyboard shortcut `cmd + shift + period` in the ``open'' macOS dialog box to display hidden files and folders, such as .profile.

Install Homebrew

Install Homebrew here http://brew.sh/

Or just type

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

Now type the following commands in your terminal (you may want to type `brew doctor` after each to make sure everything's fine):

```
brew install python
brew install swig
brew install zmq
```

Note: zmq is ZeroMQ. It's a fantastic library used for server to server network communication and is at the core of Salt efficiency.

Install Salt

You should now have everything ready to launch this command:

```
pip install salt
```

Note: There should be no need for `sudo pip install salt`. Brew installed Python for your user, so you should have all the access. In case you would like to check, type `which python` to ensure that it's /usr/local/bin/python, and `which pip` which should be /usr/local/bin/pip.

Now type `python` in a terminal then, `import salt`. There should be no errors. Now exit the Python terminal using `exit()`.

Create The Master Configuration

If the default /etc/salt/master configuration file was not created, copy-paste it from here: http://docs.saltstack.com/ref/configuration/examples.html#configuration-examples-master

Note: `/etc/salt/master` is a file, not a folder.

Salt Master configuration changes. The Salt master needs a few customization to be able to run on macOS:

```
sudo launchctl limit maxfiles 4096 8192
```

In the /etc/salt/master file, change max_open_files to 8192 (or just add the line: `max_open_files: 8192` (no quote) if it doesn't already exists).

You should now be able to launch the Salt master:

```
sudo salt-master --log-level=all
```

There should be no errors when running the above command.

Note: This command is supposed to be a daemon, but for toying around, we'll keep it running on a terminal to monitor the activity.

Now that the master is set, let's configure a minion on a VM.

Step 2 - Configuring The Minion VM

The Salt minion is going to run on a Virtual Machine. There are a lot of software options that let you run virtual machines on a mac, But for this tutorial we're going to use VirtualBox. In addition to virtualBox, we will use Vagrant, which allows you to create the base VM configuration.

Vagrant lets you build ready to use VM images, starting from an OS image and customizing it using ``provisioners''. In our case, we'll use it to:

- Download the base Ubuntu image
- Install salt on that Ubuntu image (Salt is going to be the ``provisioner'' for the VM).
- Launch the VM
- SSH into the VM to debug
- Stop the VM once you're done.

Install VirtualBox

Go get it here: https://www.virtualBox.org/wiki/Downloads (click on VirtualBox for macOS hosts => x86/amd64)

Install Vagrant

Go get it here: http://downloads.vagrantup.com/ and choose the latest version (1.3.5 at time of writing), then the .dmg file. Double-click to install it. Make sure the `vagrant` command is found when run in the terminal. Type `vagrant`. It should display a list of commands.

Create The Minion VM Folder

Create a folder in which you will store your minion's VM. In this tutorial, it's going to be a minion folder in the $home directory.

```
cd $home
mkdir minion
```

Initialize Vagrant

From the minion folder, type

```
vagrant init
```

This command creates a default Vagrantfile configuration file. This configuration file will be used to pass configuration parameters to the Salt provisioner in Step 3.

Import Precise64 Ubuntu Box

```
vagrant box add precise64 http://files.vagrantup.com/precise64.box
```

Note: This box is added at the global Vagrant level. You only need to do it once as each VM will use this same file.

Modify the Vagrantfile

Modify ./minion/Vagrantfile to use th precise64 box. Change the `config.vm.box` line to:

```
config.vm.box = "precise64"
```

Uncomment the line creating a host-only IP. This is the ip of your minion (you can change it to something else if that IP is already in use):

```
config.vm.network :private_network, ip: "192.168.33.10"
```

At this point you should have a VM that can run, although there won't be much in it. Let's check that.

Checking The VM

From the $home/minion folder type:

```
vagrant up
```

A log showing the VM booting should be present. Once it's done you'll be back to the terminal:

```
ping 192.168.33.10
```

The VM should respond to your ping request.

Now log into the VM in ssh using Vagrant again:

```
vagrant ssh
```

You should see the shell prompt change to something similar to `vagrant@precise64:~$` meaning you're inside the VM. From there, enter the following:

```
ping 10.0.2.2
```

Note: That ip is the ip of your VM host (the macOS host). The number is a VirtualBox default and is displayed in the log after the Vagrant ssh command. We'll use that IP to tell the minion where the Salt master is. Once you're done, end the ssh session by typing `exit`.

It's now time to connect the VM to the salt master

Step 3 - Connecting Master and Minion

Creating The Minion Configuration File

Create the /etc/salt/minion file. In that file, put the following lines, giving the ID for this minion, and the IP of the master:

```
master: 10.0.2.2
id: 'minion1'
file_client: remote
```

Minions authenticate with the master using keys. Keys are generated automatically if you don't provide one and can accept them later on. However, this requires accepting the minion key every time the minion is destroyed or created (which could be quite often). A better way is to create those keys in advance, feed them to the minion, and authorize them once.

Preseed minion keys

From the minion folder on your Mac run:

```
sudo salt-key --gen-keys=minion1
```

This should create two files: minion1.pem, and minion1.pub. Since those files have been created using sudo, but will be used by vagrant, you need to change ownership:

```
sudo chown youruser:yourgroup minion1.pem
sudo chown youruser:yourgroup minion1.pub
```

Then copy the .pub file into the list of accepted minions:

```
sudo cp minion1.pub /etc/salt/pki/master/minions/minion1
```

Modify Vagrantfile to Use Salt Provisioner

Let's now modify the Vagrantfile used to provision the Salt VM. Add the following section in the Vagrantfile (note: it should be at the same indentation level as the other properties):

```
# salt-vagrant config
config.vm.provision :salt do |salt|
    salt.run_highstate = true
    salt.minion_config = "/etc/salt/minion"
    salt.minion_key = "./minion1.pem"
    salt.minion_pub = "./minion1.pub"
end
```

Now destroy the vm and recreate it from the /minion folder:

```
vagrant destroy
vagrant up
```

If everything is fine you should see the following message:

```
"Bootstrapping Salt... (this may take a while)
Salt successfully configured and installed!"
```

Checking Master-Minion Communication

To make sure the master and minion are talking to each other, enter the following:

```
sudo salt '*' test.ping
```

You should see your minion answering the ping. It's now time to do some configuration.

Step 4 - Configure Services to Install On the Minion

In this step we'll use the Salt master to instruct our minion to install Nginx.

Checking the system's original state

First, make sure that an HTTP server is not installed on our minion. When opening a browser directed at `http://192.168.33.10/` You should get an error saying the site cannot be reached.

Initialize the top.sls file

System configuration is done in `/srv/salt/top.sls` (and subfiles/folders), and then applied by running the `state.apply` function to have the Salt master order its minions to update their instructions and run the associated commands.

First Create an empty file on your Salt master (macOS machine):

```
touch /srv/salt/top.sls
```

When the file is empty, or if no configuration is found for our minion an error is reported:

```
sudo salt 'minion1' state.apply
```

This should return an error stating: **No Top file or external nodes data matches found**.

Create The Nginx Configuration

Now is finally the time to enter the real meat of our server's configuration. For this tutorial our minion will be treated as a web server that needs to have Nginx installed.

Insert the following lines into `/srv/salt/top.sls` (which should current be empty).

```
base:
  'minion1':
    - bin.nginx
```

Now create `/srv/salt/bin/nginx.sls` containing the following:

```
nginx:
  pkg.installed:
    - name: nginx
  service.running:
    - enable: True
    - reload: True
```

Check Minion State

Finally, run the `state.apply` function again:

```
sudo salt 'minion1' state.apply
```

You should see a log showing that the Nginx package has been installed and the service configured. To prove it, open your browser and navigate to http://192.168.33.10/, you should see the standard Nginx welcome page.

Congratulations!

Where To Go From Here

A full description of configuration management within Salt (sls files among other things) is available here: http://docs.saltstack.com/en/latest/index.html#configuration-management

2.4.6 running salt as normal user tutorial

Before continuing make sure you have a working Salt installation by following the *Installation* and the *configuration* instructions.

Stuck?

There are many ways to *get help from the Salt community* including our mailing list and our IRC channel #salt.

Running Salt functions as non root user

If you don't want to run salt cloud as root or even install it you can configure it to have a virtual root in your working directory.

The salt system uses the `salt.syspath` module to find the variables

If you run the salt-build, it will generated in:

```
./build/lib.linux-x86_64-2.7/salt/_syspaths.py
```

To generate it, run the command:

```
python setup.py build
```

Copy the generated module into your salt directory

```
cp ./build/lib.linux-x86_64-2.7/salt/_syspaths.py salt/_syspaths.py
```

Edit it to include needed variables and your new paths

```
# you need to edit this
ROOT_DIR = *your current dir* + '/salt/root'

# you need to edit this
INSTALL_DIR = *location of source code*

CONFIG_DIR =  ROOT_DIR + '/etc/salt'
CACHE_DIR = ROOT_DIR + '/var/cache/salt'
```

```
SOCK_DIR = ROOT_DIR + '/var/run/salt'
SRV_ROOT_DIR= ROOT_DIR + '/srv'
BASE_FILE_ROOTS_DIR = ROOT_DIR + '/srv/salt'
BASE_PILLAR_ROOTS_DIR = ROOT_DIR + '/srv/pillar'
BASE_MASTER_ROOTS_DIR = ROOT_DIR + '/srv/salt-master'
LOGS_DIR = ROOT_DIR + '/var/log/salt'
PIDFILE_DIR = ROOT_DIR + '/var/run'
CLOUD_DIR = INSTALL_DIR + '/cloud'
BOOTSTRAP = CLOUD_DIR + '/deploy/bootstrap-salt.sh'
```

Create the directory structure

```
mkdir -p root/etc/salt root/var/cache/run root/run/salt root/srv
root/srv/salt root/srv/pillar root/srv/salt-master root/var/log/salt root/var/run
```

Populate the configuration files:

```
cp -r conf/* root/etc/salt/
```

Edit your `root/etc/salt/master` configuration that is used by salt-cloud:

```
user: *your user name*
```

Run like this:

```
PYTHONPATH=`pwd` scripts/salt-cloud
```

2.4.7 Standalone Minion

Since the Salt minion contains such extensive functionality it can be useful to run it standalone. A standalone minion can be used to do a number of things:

- Use salt-call commands on a system without connectivity to a master
- Masterless States, run states entirely from files local to the minion

Note: When running Salt in masterless mode, do not run the salt-minion daemon. Otherwise, it will attempt to connect to a master and fail. The salt-call command stands on its own and does not need the salt-minion daemon.

Minion Configuration

Throughout this document there are several references to setting different options to configure a masterless Minion. Salt Minions are easy to configure via a configuration file that is located, by default, in `/etc/salt/minion`. Note, however, that on FreeBSD systems, the minion configuration file is located in `/usr/local/etc/salt/minion`.

You can learn more about minion configuration options in the *Configuring the Salt Minion* docs.

Telling Salt Call to Run Masterless

The salt-call command is used to run module functions locally on a minion instead of executing them from the master. Normally the salt-call command checks into the master to retrieve file server and pillar data, but when running standalone salt-call needs to be instructed to not check the master for this data. To instruct the minion to

not look for a master when running salt-call the *file_client* configuration option needs to be set. By default the *file_client* is set to `remote` so that the minion knows that file server and pillar data are to be gathered from the master. When setting the *file_client* option to `local` the minion is configured to not gather this data from the master.

```
file_client: local
```

Now the salt-call command will not look for a master and will assume that the local system has all of the file and pillar resources.

Running States Masterless

The state system can be easily run without a Salt master, with all needed files local to the minion. To do this the minion configuration file needs to be set up to know how to return file_roots information like the master. The file_roots setting defaults to /srv/salt for the base environment just like on the master:

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

Now set up the Salt State Tree, top file, and SLS modules in the same way that they would be set up on a master. Now, with the *file_client* option set to `local` and an available state tree then calls to functions in the state module will use the information in the file_roots on the minion instead of checking in with the master.

Remember that when creating a state tree on a minion there are no syntax or path changes needed, SLS modules written to be used from a master do not need to be modified in any way to work with a minion.

This makes it easy to ``script'' deployments with Salt states without having to set up a master, and allows for these SLS modules to be easily moved into a Salt master as the deployment grows.

The declared state can now be executed with:

```
salt-call state.apply
```

Or the salt-call command can be executed with the `--local` flag, this makes it unnecessary to change the configuration file:

```
salt-call state.apply --local
```

External Pillars

External pillars are supported when running in masterless mode.

2.4.8 Salt Masterless Quickstart

Running a masterless salt-minion lets you use Salt's configuration management for a single machine without calling out to a Salt master on another machine.

Since the Salt minion contains such extensive functionality it can be useful to run it standalone. A standalone minion can be used to do a number of things:

- Stand up a master server via States (Salting a Salt Master)
- Use salt-call commands on a system without connectivity to a master

- Masterless States, run states entirely from files local to the minion

It is also useful for testing out state trees before deploying to a production setup.

Bootstrap Salt Minion

The salt-bootstrap script makes bootstrapping a server with Salt simple for any OS with a Bourne shell:

```
curl -L https://bootstrap.saltstack.com -o bootstrap_salt.sh
sudo sh bootstrap_salt.sh
```

See the salt-bootstrap documentation for other one liners. When using Vagrant to test out salt, the Vagrant salt provisioner will provision the VM for you.

Telling Salt to Run Masterless

To instruct the minion to not look for a master, the `file_client` configuration option needs to be set in the minion configuration file. By default the `file_client` is set to `remote` so that the minion gathers file server and pillar data from the salt master. When setting the `file_client` option to `local` the minion is configured to not gather this data from the master.

```
file_client: local
```

Now the salt minion will not look for a master and will assume that the local system has all of the file and pillar resources.

Configuration which resided in the *master configuration* (e.g. `/etc/salt/master`) should be moved to the *minion configuration* since the minion does not read the master configuration.

Note: When running Salt in masterless mode, do not run the salt-minion daemon. Otherwise, it will attempt to connect to a master and fail. The salt-call command stands on its own and does not need the salt-minion daemon.

Create State Tree

Following the successful installation of a salt-minion, the next step is to create a state tree, which is where the SLS files that comprise the possible states of the minion are stored.

The following example walks through the steps necessary to create a state tree that ensures that the server has the Apache webserver installed.

Note: For a complete explanation on Salt States, see the tutorial.

1. Create the `top.sls` file:

`/srv/salt/top.sls:`

```
base:
  '*':
    - webserver
```

2. Create the webserver state tree:

`/srv/salt/webserver.sls:`

```
apache:                 # ID declaration
  pkg:                  # state declaration
    - installed         # function declaration
```

Note: The apache package has different names on different platforms, for instance on Debian/Ubuntu it is apache2, on Fedora/RHEL it is httpd and on Arch it is apache

The only thing left is to provision our minion using `salt-call`.

Salt-call

The salt-call command is used to run remote execution functions locally on a minion instead of executing them from the master. Normally the salt-call command checks into the master to retrieve file server and pillar data, but when running standalone salt-call needs to be instructed to not check the master for this data:

```
salt-call --local state.apply
```

The `--local` flag tells the salt-minion to look for the state tree in the local file system and not to contact a Salt Master for instructions.

To provide verbose output, use `-l debug`:

```
salt-call --local state.apply -l debug
```

The minion first examines the `top.sls` file and determines that it is a part of the group matched by `*` glob and that the `webserver` SLS should be applied.

It then examines the `webserver.sls` file and finds the `apache` state, which installs the Apache package.

The minion should now have Apache installed, and the next step is to begin learning how to write *more complex states*.

2.5 Dependencies

Salt should run on any Unix-like platform so long as the dependencies are met.

- Python 2.7 >= 2.7 <3.0
- msgpack-python - High-performance message interchange format
- YAML - Python YAML bindings
- Jinja2 - parsing Salt States (configurable in the master settings)
- MarkupSafe - Implements a XML/HTML/XHTML Markup safe string for Python
- apache-libcloud - Python lib for interacting with many of the popular cloud service providers using a unified API
- Requests - HTTP library
- Tornado - Web framework and asynchronous networking library
- futures - Backport of the concurrent.futures package from Python 3.2

Depending on the chosen Salt transport, ZeroMQ or RAET, dependencies vary:

- ZeroMQ:
  - ZeroMQ >= 3.2.0
  - pyzmq >= 2.2.0 - ZeroMQ Python bindings
  - PyCrypto - The Python cryptography toolkit
- RAET:
  - libnacl - Python bindings to libsodium
  - ioflo - The flo programming interface raet and salt-raet is built on
  - RAET - The worlds most awesome UDP protocol

Salt defaults to the ZeroMQ transport, and the choice can be made at install time, for example:

```
python setup.py --salt-transport=raet install
```

This way, only the required dependencies are pulled by the setup script if need be.

If installing using pip, the `--salt-transport` install option can be provided like:

```
pip install --install-option="--salt-transport=raet" salt
```

Note: Salt does not bundle dependencies that are typically distributed as part of the base OS. If you have unmet dependencies and are using a custom or minimal installation, you might need to install some additional packages from your OS vendor.

2.6 Optional Dependencies

- mako - an optional parser for Salt States (configurable in the master settings)
- gcc - dynamic Cython module compiling

2.7 Upgrading Salt

When upgrading Salt, the master(s) should always be upgraded first. Backward compatibility for minions running newer versions of salt than their masters is not guaranteed.

Whenever possible, backward compatibility between new masters and old minions will be preserved. Generally, the only exception to this policy is in case of a security vulnerability.

See also:

Installing Salt for development and contributing to the project.

2.8 Building Packages using Salt Pack

Salt-pack is an open-source package builder for most commonly used Linux platforms, for example: Redhat/CentOS and Debian/Ubuntu families, utilizing SaltStack states and execution modules to build Salt and a specified set of dependencies, from which a platform specific repository can be built.

https://github.com/saltstack/salt-pack

CHAPTER 3

Configuring Salt

This section explains how to configure user access, view and store job results, secure and troubleshoot, and how to perform many other administrative tasks.

3.1 Configuring the Salt Master

The Salt system is amazingly simple and easy to configure, the two components of the Salt system each have a respective configuration file. The **`salt-master`** is configured via the master configuration file, and the **`salt-minion`** is configured via the minion configuration file.

See also:

Example master configuration file.

The configuration file for the salt-master is located at `/etc/salt/master` by default. A notable exception is FreeBSD, where the configuration file is located at `/usr/local/etc/salt`. The available options are as follows:

3.1.1 Primary Master Configuration

`interface`

Default: `0.0.0.0` (all interfaces)

The local interface to bind to, must be an IP address.

```
interface: 192.168.0.1
```

`ipv6`

Default: `False`

Whether the master should listen for IPv6 connections. If this is set to True, the interface option must be adjusted too (for example: `interface: '::'`)

```
ipv6: True
```

publish_port

Default: 4505

The network port to set up the publication interface.

```
publish_port: 4505
```

master_id

Default: None

The id to be passed in the publish job to minions. This is used for MultiSyndics to return the job to the requesting master.

Note: This must be the same string as the syndic is configured with.

```
master_id: MasterOfMaster
```

user

Default: root

The user to run the Salt processes

```
user: root
```

enable_ssh_minions

Default: False

Tell the master to also use salt-ssh when running commands against minions.

```
enable_ssh_minions: True
```

Note: Cross-minion communication is still not possible. The Salt mine and publish.publish do not work between minion types.

ret_port

Default: 4506

The port used by the return server, this is the server used by Salt to receive execution returns and command executions.

```
ret_port: 4506
```

pidfile

Default: `/var/run/salt-master.pid`

Specify the location of the master pidfile.

```
pidfile: /var/run/salt-master.pid
```

root_dir

Default: `/`

The system root directory to operate from, change this to make Salt run from an alternative root.

```
root_dir: /
```

Note: This directory is prepended to the following options: *pki_dir*, *cachedir*, *sock_dir*, *log_file*, *autosign_file*, *autoreject_file*, *pidfile*, *autosign_grains_dir*.

conf_file

Default: `/etc/salt/master`

The path to the master's configuration file.

```
conf_file: /etc/salt/master
```

pki_dir

Default: `/etc/salt/pki/master`

The directory to store the pki authentication keys.

```
pki_dir: /etc/salt/pki/master
```

extension_modules

Changed in version 2016.3.0: The default location for this directory has been moved. Prior to this version, the location was a directory named `extmods` in the Salt cachedir (on most platforms, `/var/cache/salt/extmods`). It has been moved into the master cachedir (on most platforms, `/var/cache/salt/master/extmods`).

Directory for custom modules. This directory can contain subdirectories for each of Salt's module types such as `runners`, `output`, `wheel`, `modules`, `states`, `returners`, `engines`, `utils`, etc. This path is appended to *root_dir*.

```
extension_modules: /root/salt_extmods
```

extmod_whitelist/extmod_blacklist

New in version 2017.7.0.

By using this dictionary, the modules that are synced to the master's extmod cache using *saltutil.sync_** can be limited. If nothing is set to a specific type, then all modules are accepted. To block all modules of a specific type, whitelist an empty list.

```
extmod_whitelist:
  modules:
    - custom_module
  engines:
    - custom_engine
  pillars: []

extmod_blacklist:
  modules:
    - specific_module
```

Valid options:

- modules
- states
- grains
- renderers
- returners
- output
- proxy
- runners
- wheel
- engines
- queues
- pillar
- utils
- sdb
- cache
- clouds
- tops
- roster
- tokens

module_dirs

Default: `[]`

Like `extension_modules`, but a list of extra directories to search for Salt modules.

```
module_dirs:
  - /var/cache/salt/minion/extmods
```

cachedir

Default: `/var/cache/salt/master`

The location used to store cache information, particularly the job information for executed salt commands.

This directory may contain sensitive data and should be protected accordingly.

```
cachedir: /var/cache/salt/master
```

verify_env

Default: `True`

Verify and set permissions on configuration directories at startup.

```
verify_env: True
```

keep_jobs

Default: `24`

Set the number of hours to keep old job information. Note that setting this option to `0` disables the cache cleaner.

```
keep_jobs: 24
```

gather_job_timeout

New in version 2014.7.0.

Default: `10`

The number of seconds to wait when the client is requesting information about running jobs.

```
gather_job_timeout: 10
```

timeout

Default: `5`

Set the default timeout for the salt command and api.

loop_interval

Default: `60`

The loop_interval option controls the seconds for the master's maintenance process check cycle. This process updates file server backends, cleans the job cache and executes the scheduler.

output

Default: nested

Set the default outputter used by the salt command.

outputter_dirs

Default: []

A list of additional directories to search for salt outputters in.

```
outputter_dirs: []
```

output_file

Default: None

Set the default output file used by the salt command. Default is to output to the CLI and not to a file. Functions the same way as the ``--out-file'' CLI option, only sets this to a single file for all salt commands.

```
output_file: /path/output/file
```

show_timeout

Default: True

Tell the client to show minions that have timed out.

```
show_timeout: True
```

show_jid

Default: False

Tell the client to display the jid when a job is published.

```
show_jid: False
```

color

Default: True

By default output is colored, to disable colored output set the color value to False.

```
color: False
```

color_theme

Default: ""

Specifies a path to the color theme to use for colored command line output.

```
color_theme: /etc/salt/color_theme
```

cli_summary

Default: False

When set to True, displays a summary of the number of minions targeted, the number of minions returned, and the number of minions that did not return.

```
cli_summary: False
```

sock_dir

Default: /var/run/salt/master

Set the location to use for creating Unix sockets for master process communication.

```
sock_dir: /var/run/salt/master
```

enable_gpu_grains

Default: False

Enable GPU hardware data for your master. Be aware that the master can take a while to start up when lspci and/or dmidecode is used to populate the grains for the master.

```
enable_gpu_grains: True
```

job_cache

Default: True

The master maintains a temporary job cache. While this is a great addition, it can be a burden on the master for larger deployments (over 5000 minions). Disabling the job cache will make previously executed jobs unavailable to the jobs system and is not generally recommended. Normally it is wise to make sure the master has access to a faster IO system or a tmpfs is mounted to the jobs dir.

```
job_cache: True
```

Note: Setting the job_cache to False will not cache minion returns, but the JID directory for each job is still created. The creation of the JID directories is necessary because Salt uses those directories to check for JID collisions. By setting this option to False, the job cache directory, which is /var/cache/salt/master/jobs/ by default, will be smaller, but the JID directories will still be present.

Note that the *keep_jobs* option can be set to a lower value, such as 1, to limit the number of hours jobs are stored in the job cache. (The default is 24 hours.)

Please see the *Managing the Job Cache* documentation for more information.

minion_data_cache

Default: `True`

The minion data cache is a cache of information about the minions stored on the master, this information is primarily the pillar, grains and mine data. The data is cached via the cache subsystem in the Master cachedir under the name of the minion or in a supported database. The data is used to predetermine what minions are expected to reply from executions.

```
minion_data_cache: True
```

cache

Default: `localfs`

Cache subsystem module to use for minion data cache.

```
cache: consul
```

memcache_expire_seconds

Default: `0`

Memcache is an additional cache layer that keeps a limited amount of data fetched from the minion data cache for a limited period of time in memory that makes cache operations faster. It doesn't make much sense for the `localfs` cache driver but helps for more complex drivers like `consul`.

This option sets the memcache items expiration time. By default is set to `0` that disables the memcache.

```
memcache_expire_seconds: 30
```

memcache_max_items

Default: `1024`

Set memcache limit in items that are bank-key pairs. I.e the list of minion_0/data, minion_0/mine, minion_1/data contains 3 items. This value depends on the count of minions usually targeted in your environment. The best one could be found by analyzing the cache log with `memcache_debug` enabled.

```
memcache_max_items: 1024
```

memcache_full_cleanup

Default: `False`

If cache storage got full, i.e. the items count exceeds the `memcache_max_items` value, memcache cleans up it's storage. If this option set to `False` memcache removes the only one oldest value from it's storage. If this set set to `True` memcache removes all the expired items and also removes the oldest one if there are no expired items.

```
memcache_full_cleanup: True
```

memcache_debug

Default: `False`

Enable collecting the memcache stats and log it on *debug* log level. If enabled memcache collect information about how many `fetch` calls has been done and how many of them has been hit by memcache. Also it outputs the rate value that is the result of division of the first two values. This should help to choose right values for the expiration time and the cache size.

```
memcache_debug: True
```

ext_job_cache

Default: `''`

Used to specify a default returner for all minions. When this option is set, the specified returner needs to be properly configured and the minions will always default to sending returns to this returner. This will also disable the local job cache on the master.

```
ext_job_cache: redis
```

event_return

New in version 2015.5.0.

Default: `''`

Specify the returner(s) to use to log events. Each returner may have installation and configuration requirements. Read the returner's documentation.

Note: Not all returners support event returns. Verify that a returner has an `event_return()` function before configuring this option with a returner.

```
event_return:
  - syslog
  - splunk
```

event_return_queue

New in version 2015.5.0.

Default: `0`

On busy systems, enabling event_returns can cause a considerable load on the storage system for returners. Events can be queued on the master and stored in a batched fashion using a single transaction for multiple events. By default, events are not queued.

```
event_return_queue: 0
```

event_return_whitelist

New in version 2015.5.0.

Default: `[]`

Only return events matching tags in a whitelist.

Changed in version 2016.11.0: Supports glob matching patterns.

```
event_return_whitelist:
  - salt/master/a_tag
  - salt/run/*/ret
```

event_return_blacklist

New in version 2015.5.0.

Default: `[]`

Store all event returns _except_ the tags in a blacklist.

Changed in version 2016.11.0: Supports glob matching patterns.

```
event_return_blacklist:
  - salt/master/not_this_tag
  - salt/wheel/*/ret
```

max_event_size

New in version 2014.7.0.

Default: `1048576`

Passing very large events can cause the minion to consume large amounts of memory. This value tunes the maximum size of a message allowed onto the master event bus. The value is expressed in bytes.

```
max_event_size: 1048576
```

master_job_cache

New in version 2014.7.0.

Default: `local_cache`

Specify the returner to use for the job cache. The job cache will only be interacted with from the salt master and therefore does not need to be accessible from the minions.

```
master_job_cache: redis
```

enforce_mine_cache

Default: False

By-default when disabling the minion_data_cache mine will stop working since it is based on cached data, by enabling this option we explicitly enabling only the cache for the mine system.

```
enforce_mine_cache: False
```

max_minions

Default: 0

The maximum number of minion connections allowed by the master. Use this to accommodate the number of minions per master if you have different types of hardware serving your minions. The default of `0` means unlimited connections. Please note that this can slow down the authentication process a bit in large setups.

```
max_minions: 100
```

con_cache

Default: False

If max_minions is used in large installations, the master might experience high-load situations because of having to check the number of connected minions for every authentication. This cache provides the minion-ids of all connected minions to all MWorker-processes and greatly improves the performance of max_minions.

```
con_cache: True
```

presence_events

Default: False

Causes the master to periodically look for actively connected minions. *Presence events* are fired on the event bus on a regular interval with a list of connected minions, as well as events with lists of newly connected or disconnected minions. This is a master-only operation that does not send executions to minions. Note, this does not detect minions that connect to a master via localhost.

```
presence_events: False
```

ping_on_rotate

New in version 2014.7.0.

Default: `False`

By default, the master AES key rotates every 24 hours. The next command following a key rotation will trigger a key refresh from the minion which may result in minions which do not respond to the first command after a key refresh.

To tell the master to ping all minions immediately after an AES key refresh, set `ping_on_rotate` to `True`. This should mitigate the issue where a minion does not appear to initially respond after a key is rotated.

Note that enabling this may cause high load on the master immediately after the key rotation event as minions reconnect. Consider this carefully if this salt master is managing a large number of minions.

If disabled, it is recommended to handle this event by listening for the `aes_key_rotate` event with the `key` tag and acting appropriately.

```
ping_on_rotate: False
```

transport

Default: zeromq

Changes the underlying transport layer. ZeroMQ is the recommended transport while additional transport layers are under development. Supported values are zeromq, raet (experimental), and tcp (experimental). This setting has a significant impact on performance and should not be changed unless you know what you are doing!

```
transport: zeromq
```

transport_opts

Default: {}

(experimental) Starts multiple transports and overrides options for each transport with the provided dictionary This setting has a significant impact on performance and should not be changed unless you know what you are doing! The following example shows how to start a TCP transport alongside a ZMQ transport.

```
transport_opts:
  tcp:
    publish_port: 4605
    ret_port: 4606
  zeromq: []
```

master_stats

Default: False

Turning on the master stats enables runtime throughput and statistics events to be fired from the master event bus. These events will report on what functions have been run on the master and how long these runs have, on average, taken over a given period of time.

master_stats_event_iter

Default: 60

The time in seconds to fire master_stats events. This will only fire in conjunction with receiving a request to the master, idle masters will not fire these events.

sock_pool_size

Default: 1

To avoid blocking waiting while writing a data to a socket, we support socket pool for Salt applications. For example, a job with a large number of target host list can cause long period blocking waiting. The option is used by ZMQ and TCP transports, and the other transport methods don't need the socket pool by definition. Most of Salt tools, including CLI, are enough to use a single bucket of socket pool. On the other hands, it is highly recommended to set the size of socket pool larger than 1 for other Salt applications, especially Salt API, which must write data to socket concurrently.

```
sock_pool_size: 15
```

ipc_mode

Default: ipc

The ipc strategy. (i.e., sockets versus tcp, etc.) Windows platforms lack POSIX IPC and must rely on TCP based inter-process communications. ipc_mode is set to tcp by default on Windows.

```
ipc_mode: ipc
```

tcp_master_pub_port

Default: 4512

The TCP port on which events for the master should be published if ipc_mode is TCP.

```
tcp_master_pub_port: 4512
```

tcp_master_pull_port

Default: 4513

The TCP port on which events for the master should be pulled if ipc_mode is TCP.

```
tcp_master_pull_port: 4513
```

tcp_master_publish_pull

Default: 4514

The TCP port on which events for the master should be pulled fom and then republished onto the event bus on the master.

```
tcp_master_publish_pull: 4514
```

tcp_master_workers

Default: 4515

The TCP port for mworkers to connect to on the master.

```
tcp_master_workers: 4515
```

auth_events

New in version 2017.7.3.

Default: True

Determines whether the master will fire authentication events. *Authentication events* are fired when a minion performs an authentication check with the master.

```
auth_events: True
```

minion_data_cache_events

New in version 2017.7.3.

Default: True

Determines whether the master will fire minion data cache events. Minion data cache events are fired when a minion requests a minion data cache refresh.

```
minion_data_cache_events: True
```

3.1.2 Salt-SSH Configuration

roster

Default: flat

Define the default salt-ssh roster module to use

```
roster: cache
```

roster_defaults

New in version 2017.7.0.

Default settings which will be inherited by all rosters.

```
roster_defaults:
  user: daniel
  sudo: True
  priv: /root/.ssh/id_rsa
  tty: True
```

roster_file

Default: /etc/salt/roster

Pass in an alternative location for the salt-ssh *flat* roster file.

```
roster_file: /root/roster
```

rosters

Default: None

Define locations for *flat* roster files so they can be chosen when using Salt API. An administrator can place roster files into these locations. Then, when calling Salt API, the *roster_file* parameter should contain a relative path to these locations. That is, roster_file=/foo/roster will be resolved as /etc/salt/roster.d/foo/roster etc. This feature prevents passing insecure custom rosters through the Salt API.

```
rosters:
 - /etc/salt/roster.d
 - /opt/salt/some/more/rosters
```

ssh_passwd

Default: `''`

The ssh password to log in with.

```
ssh_passwd: ''
```

ssh_port

Default: `22`

The target system's ssh port number.

```
ssh_port: 22
```

ssh_scan_ports

Default: `22`

Comma-separated list of ports to scan.

```
ssh_scan_ports: 22
```

ssh_scan_timeout

Default: `0.01`

Scanning socket timeout for salt-ssh.

```
ssh_scan_timeout: 0.01
```

ssh_sudo

Default: `False`

Boolean to run command via sudo.

```
ssh_sudo: False
```

ssh_timeout

Default: `60`

Number of seconds to wait for a response when establishing an SSH connection.

```
ssh_timeout: 60
```

ssh_user

Default: `root`

The user to log in as.

```
ssh_user: root
```

ssh_log_file

New in version 2016.3.5.

Default: `/var/log/salt/ssh`

Specify the log file of the `salt-ssh` command.

```
ssh_log_file: /var/log/salt/ssh
```

ssh_minion_opts

Default: None

Pass in minion option overrides that will be inserted into the SHIM for salt-ssh calls. The local minion config is not used for salt-ssh. Can be overridden on a per-minion basis in the roster (`minion_opts`)

```
ssh_minion_opts:
  gpg_keydir: /root/gpg
```

ssh_use_home_key

Default: False

Set this to True to default to using `~/.ssh/id_rsa` for salt-ssh authentication with minions

```
ssh_use_home_key: False
```

ssh_identities_only

Default: `False`

Set this to `True` to default salt-ssh to run with `-o IdentitiesOnly=yes`. This option is intended for situations where the ssh-agent offers many different identities and allows ssh to ignore those identities and use the only one specified in options.

```
ssh_identities_only: False
```

ssh_list_nodegroups

Default: `{}`

List-only nodegroups for salt-ssh. Each group must be formed as either a comma-separated list, or a YAML list. This option is useful to group minions into easy-to-target groups when using salt-ssh. These groups can then be targeted with the normal -N argument to salt-ssh.

```
ssh_list_nodegroups:
  groupA: minion1,minion2
  groupB: minion1,minion3
```

thin_extra_mods

Default: None

List of additional modules, needed to be included into the Salt Thin. Pass a list of importable Python modules that are typically located in the *site-packages* Python directory so they will be also always included into the Salt Thin, once generated.

min_extra_mods

Default: None

Identical as *thin_extra_mods*, only applied to the Salt Minimal.

3.1.3 Master Security Settings

open_mode

Default: `False`

Open mode is a dangerous security feature. One problem encountered with pki authentication systems is that keys can become ``mixed up'' and authentication begins to fail. Open mode turns off authentication and tells the master to accept all authentication. This will clean up the pki keys received from the minions. Open mode should not be turned on for general use. Open mode should only be used for a short period of time to clean up pki keys. To turn on open mode set this value to `True`.

```
open_mode: False
```

auto_accept

Default: `False`

Enable auto_accept. This setting will automatically accept all incoming public keys from minions.

```
auto_accept: False
```

keysize

Default: `2048`

The size of key that should be generated when creating new keys.

```
keysize: 2048
```

autosign_timeout

New in version 2014.7.0.

Default: `120`

Time in minutes that a incoming public key with a matching name found in pki_dir/minion_autosign/keyid is automatically accepted. Expired autosign keys are removed when the master checks the minion_autosign directory. This method to auto accept minions can be safer than an autosign_file because the keyid record can expire and is limited to being an exact name match. This should still be considered a less than secure option, due to the fact that trust is based on just the requesting minion id.

autosign_file

Default: `not defined`

If the `autosign_file` is specified incoming keys specified in the autosign_file will be automatically accepted. Matches will be searched for first by string comparison, then by globbing, then by full-string regex matching. This should still be considered a less than secure option, due to the fact that trust is based on just the requesting minion id.

Changed in version 2018.3.0: For security reasons the file must be readonly except for it's owner. If *permissive_pki_access* is `True` the owning group can also have write access, but if Salt is running as `root` it must be a member of that group. A less strict requirement also existed in previous version.

autoreject_file

New in version 2014.1.0.

Default: `not defined`

Works like *autosign_file*, but instead allows you to specify minion IDs for which keys will automatically be rejected. Will override both membership in the *autosign_file* and the *auto_accept* setting.

autosign_grains_dir

New in version 2018.3.0.

Default: `not defined`

If the `autosign_grains_dir` is specified, incoming keys from minions with grain values that match those defined in files in the autosign_grains_dir will be accepted automatically. Grain values that should be accepted automatically can be defined by creating a file named like the corresponding grain in the autosign_grains_dir and writing the values into that file, one value per line. Lines starting with a # will be ignored. Minion must be configured to send the corresponding grains on authentication. This should still be considered a less than secure option, due to the fact that trust is based on just the requesting minion.

Please see the *Autoaccept Minions from Grains* documentation for more information.

```
autosign_grains_dir: /etc/salt/autosign_grains
```

permissive_pki_access

Default: `False`

Enable permissive access to the salt keys. This allows you to run the master or minion as root, but have a non-root group be given access to your pki_dir. To make the access explicit, root must belong to the group you've given access to. This is potentially quite insecure. If an autosign_file is specified, enabling permissive_pki_access will allow group access to that specific file.

```
permissive_pki_access: False
```

publisher_acl

Default: `{}`

Enable user accounts on the master to execute specific modules. These modules can be expressed as regular expressions.

```
publisher_acl:
  fred:
    - test.ping
    - pkg.*
```

publisher_acl_blacklist

Default: `{}`

Blacklist users or modules

This example would blacklist all non sudo users, including root from running any commands. It would also blacklist any use of the ``cmd'' module.

This is completely disabled by default.

```
publisher_acl_blacklist:
  users:
    - root
    - '^(?!sudo_).*$'   #  all non sudo users
  modules:
    - cmd.*
    - test.echo
```

sudo_acl

Default: `False`

Enforce `publisher_acl` and `publisher_acl_blacklist` when users have sudo access to the salt command.

```
sudo_acl: False
```

external_auth

Default: `{}`

The external auth system uses the Salt auth modules to authenticate and validate users to access areas of the Salt system.

```
external_auth:
  pam:
    fred:
      - test.*
```

token_expire

Default: 43200

Time (in seconds) for a newly generated token to live.

Default: 12 hours

```
token_expire: 43200
```

token_expire_user_override

Default: False

Allow eauth users to specify the expiry time of the tokens they generate.

A boolean applies to all users or a dictionary of whitelisted eauth backends and usernames may be given:

```
token_expire_user_override:
  pam:
    - fred
    - tom
  ldap:
    - gary
```

keep_acl_in_token

Default: False

Set to True to enable keeping the calculated user's auth list in the token file. This is disabled by default and the auth list is calculated or requested from the eauth driver each time.

```
keep_acl_in_token: False
```

eauth_acl_module

Default: ''

Auth subsystem module to use to get authorized access list for a user. By default it's the same module used for external authentication.

```
eauth_acl_module: django
```

file_recv

Default: False

Allow minions to push files to the master. This is disabled by default, for security purposes.

```
file_recv: False
```

file_recv_max_size

New in version 2014.7.0.

Default: `100`

Set a hard-limit on the size of the files that can be pushed to the master. It will be interpreted as megabytes.

```
file_recv_max_size: 100
```

master_sign_pubkey

Default: `False`

Sign the master auth-replies with a cryptographic signature of the master's public key. Please see the tutorial how to use these settings in the Multimaster-PKI with Failover Tutorial

```
master_sign_pubkey: True
```

master_sign_key_name

Default: `master_sign`

The customizable name of the signing-key-pair without suffix.

```
master_sign_key_name: <filename_without_suffix>
```

master_pubkey_signature

Default: `master_pubkey_signature`

The name of the file in the master's pki-directory that holds the pre-calculated signature of the master's public-key.

```
master_pubkey_signature: <filename>
```

master_use_pubkey_signature

Default: `False`

Instead of computing the signature for each auth-reply, use a pre-calculated signature. The *master_pubkey_signature* must also be set for this.

```
master_use_pubkey_signature: True
```

rotate_aes_key

Default: True

Rotate the salt-masters AES-key when a minion-public is deleted with salt-key. This is a very important security-setting. Disabling it will enable deleted minions to still listen in on the messages published by the salt-master. Do not disable this unless it is absolutely clear what this does.

```
rotate_aes_key: True
```

publish_session

Default: 86400

The number of seconds between AES key rotations on the master.

```
publish_session: Default: 86400
```

ssl

New in version 2016.11.0.

Default: None

TLS/SSL connection options. This could be set to a dictionary containing arguments corresponding to python ssl.wrap_socket method. For details see Tornado and Python documentation.

Note: to set enum arguments values like cert_reqs and ssl_version use constant names without ssl module prefix: CERT_REQUIRED or PROTOCOL_SSLv23.

```
ssl:
    keyfile: <path_to_keyfile>
    certfile: <path_to_certfile>
    ssl_version: PROTOCOL_TLSv1_2
```

preserve_minion_cache

Default: False

By default, the master deletes its cache of minion data when the key for that minion is removed. To preserve the cache after key deletion, set preserve_minion_cache to True.

WARNING: This may have security implications if compromised minions auth with a previous deleted minion ID.

```
preserve_minion_cache: False
```

allow_minion_key_revoke

Default: True

Controls whether a minion can request its own key revocation. When True the master will honor the minion's request and revoke its key. When False, the master will drop the request and the minion's key will remain accepted.

```
allow_minion_key_revoke: False
```

`optimization_order`

Default: `[0,1,2]`

In cases where Salt is distributed without .py files, this option determines the priority of optimization level(s) Salt's module loader should prefer.

Note: This option is only supported on Python 3.5+.

```
optimization_order:
  - 2
  - 0
  - 1
```

3.1.4 Master Large Scale Tuning Settings

`max_open_files`

Default: `100000`

Each minion connecting to the master uses AT LEAST one file descriptor, the master subscription connection. If enough minions connect you might start seeing on the console(and then salt-master crashes):

```
Too many open files (tcp_listener.cpp:335)
Aborted (core dumped)
```

```
max_open_files: 100000
```

By default this value will be the one of *ulimit -Hn*, i.e., the hard limit for max open files.

To set a different value than the default one, uncomment, and configure this setting. Remember that this value CANNOT be higher than the hard limit. Raising the hard limit depends on the OS and/or distribution, a good way to find the limit is to search the internet for something like this:

```
raise max open files hard limit debian
```

`worker_threads`

Default: `5`

The number of threads to start for receiving commands and replies from minions. If minions are stalling on replies because you have many minions, raise the worker_threads value.

Worker threads should not be put below 3 when using the peer system, but can drop down to 1 worker otherwise.

Note: When the master daemon starts, it is expected behaviour to see multiple salt-master processes, even if `worker_threads' is set to `1'. At a minimum, a controlling process will start along with a Publisher, an EventPublisher, and a number of MWorker processes will be started. The number of MWorker processes is tuneable by the `worker_threads' configuration value while the others are not.

```
worker_threads: 5
```

pub_hwm

Default: 1000

The zeromq high water mark on the publisher interface.

```
pub_hwm: 1000
```

zmq_backlog

Default: 1000

The listen queue size of the ZeroMQ backlog.

```
zmq_backlog: 1000
```

3.1.5 Master Module Management

runner_dirs

Default: []

Set additional directories to search for runner modules.

```
runner_dirs:
  - /var/lib/salt/runners
```

utils_dirs

New in version 2018.3.0.

Default: []

Set additional directories to search for util modules.

```
utils_dirs:
  - /var/lib/salt/utils
```

cython_enable

Default: False

Set to true to enable Cython modules (.pyx files) to be compiled on the fly on the Salt master.

```
cython_enable: False
```

3.1.6 Master State System Settings

state_top

Default: top.sls

The state system uses a ``top'' file to tell the minions what environment to use and what modules to use. The state_top file is defined relative to the root of the base environment. The value of ``state_top'' is also used for the pillar top file

```
state_top: top.sls
```

state_top_saltenv

This option has no default value. Set it to an environment name to ensure that *only* the top file from that environment is considered during a *highstate*.

Note: Using this value does not change the merging strategy. For instance, if `top_file_merging_strategy` is set to `merge`, and `state_top_saltenv` is set to `foo`, then any sections for environments other than `foo` in the top file for the `foo` environment will be ignored. With `state_top_saltenv` set to `base`, all states from all environments in the `base` top file will be applied, while all other top files are ignored. The only way to set `state_top_saltenv` to something other than `base` and not have the other environments in the targeted top file ignored, would be to set `top_file_merging_strategy` to `merge_all`.

```
state_top_saltenv: dev
```

top_file_merging_strategy

Changed in version 2016.11.0: A `merge_all` strategy has been added.

Default: `merge`

When no specific fileserver environment (a.k.a. `saltenv`) has been specified for a *highstate*, all environments' top files are inspected. This config option determines how the SLS targets in those top files are handled.

When set to `merge`, the `base` environment's top file is evaluated first, followed by the other environments' top files. The first target expression (e.g. `'*'`) for a given environment is kept, and when the same target expression is used in a different top file evaluated later, it is ignored. Because `base` is evaluated first, it is authoritative. For example, if there is a target for `'*'` for the `foo` environment in both the `base` and `foo` environment's top files, the one in the `foo` environment would be ignored. The environments will be evaluated in no specific order (aside from `base` coming first). For greater control over the order in which the environments are evaluated, use `env_order`. Note that, aside from the `base` environment's top file, any sections in top files that do not match that top file's environment will be ignored. So, for example, a section for the `qa` environment would be ignored if it appears in the `dev` environment's top file. To keep use cases like this from being ignored, use the `merge_all` strategy.

When set to `same`, then for each environment, only that environment's top file is processed, with the others being ignored. For example, only the `dev` environment's top file will be processed for the `dev` environment, and any SLS targets defined for `dev` in the `base` environment's (or any other environment's) top file will be ignored. If an environment does not have a top file, then the top file from the `default_top` config parameter will be used as a fallback.

When set to `merge_all`, then all states in all environments in all top files will be applied. The order in which individual SLS files will be executed will depend on the order in which the top files were evaluated, and the environments will be evaluated in no specific order. For greater control over the order in which the environments are evaluated, use `env_order`.

```
top_file_merging_strategy: same
```

env_order

Default: []

When *top_file_merging_strategy* is set to merge, and no environment is specified for a *highstate*, this config option allows for the order in which top files are evaluated to be explicitly defined.

```
env_order:
  - base
  - dev
  - qa
```

master_tops

Default: {}

The master_tops option replaces the external_nodes option by creating a pluggable system for the generation of external top data. The external_nodes option is deprecated by the master_tops option. To gain the capabilities of the classic external_nodes system, use the following configuration:

```
master_tops:
  ext_nodes: <Shell command which returns yaml>
```

renderer

Default: yaml_jinja

The renderer to use on the minions to render the state data.

```
renderer: yaml_jinja
```

userdata_template

New in version 2016.11.4.

Default: None

The renderer to use for templating userdata files in salt-cloud, if the userdata_template is not set in the cloud profile. If no value is set in the cloud profile or master config file, no templating will be performed.

```
userdata_template: jinja
```

jinja_env

New in version 2018.3.0.

Default: {}

jinja_env overrides the default Jinja environment options for **all templates except sls templates**. To set the options for sls templates use *jinja_sls_env*.

Note: The Jinja2 Environment documentation is the official source for the default values. Not all the options listed in the jinja documentation can be overridden using *jinja_env* or *jinja_sls_env*.

The default options are:

```
jinja_env:
  block_start_string: '{%'
  block_end_string: '%}'
  variable_start_string: '{{'
  variable_end_string: '}}'
  comment_start_string: '{#'
  comment_end_string: '#}'
  line_statement_prefix:
  line_comment_prefix:
  trim_blocks: False
  lstrip_blocks: False
  newline_sequence: '\n'
  keep_trailing_newline: False
```

jinja_sls_env

New in version 2018.3.0.

Default: {}

jinja_sls_env sets the Jinja environment options for **sls templates**. The defaults and accepted options are exactly the same as they are for *jinja_env*.

The default options are:

```
jinja_sls_env:
  block_start_string: '{%'
  block_end_string: '%}'
  variable_start_string: '{{'
  variable_end_string: '}}'
  comment_start_string: '{#'
  comment_end_string: '#}'
  line_statement_prefix:
  line_comment_prefix:
  trim_blocks: False
  lstrip_blocks: False
  newline_sequence: '\n'
  keep_trailing_newline: False
```

Example using line statements and line comments to increase ease of use:

If your configuration options are

```
jinja_sls_env:
  line_statement_prefix: '%'
  line_comment_prefix: '##'
```

With these options jinja will interpret anything after a % at the start of a line (ignoreing whitespace) as a jinja statement and will interpret anything after a ## as a comment.

This allows the following more convenient syntax to be used:

```
## (this comment will not stay once rendered)
# (this comment remains in the rendered template)
## ensure all the formula services are running
% for service in formula_services:
enable_service_{{ service }}:
```

```
  service.running:
    name: {{ service }}
% endfor
```

The following less convenient but equivalent syntax would have to be used if you had not set the line_statement and line_comment options:

```
{# (this comment will not stay once rendered) #}
# (this comment remains in the rendered template)
{# ensure all the formula services are running #}
{% for service in formula_services %}
enable_service_{{ service }}:
  service.running:
    name: {{ service }}
{% endfor %}
```

`jinja_trim_blocks`

Deprecated since version 2018.3.0: Replaced by *jinja_env* and *jinja_sls_env*

New in version 2014.1.0.

Default: `False`

If this is set to `True`, the first newline after a Jinja block is removed (block, not variable tag!). Defaults to `False` and corresponds to the Jinja environment init variable `trim_blocks`.

```
jinja_trim_blocks: False
```

`jinja_lstrip_blocks`

Deprecated since version 2018.3.0: Replaced by *jinja_env* and *jinja_sls_env*

New in version 2014.1.0.

Default: `False`

If this is set to `True`, leading spaces and tabs are stripped from the start of a line to a block. Defaults to `False` and corresponds to the Jinja environment init variable `lstrip_blocks`.

```
jinja_lstrip_blocks: False
```

`failhard`

Default: `False`

Set the global failhard flag. This informs all states to stop running states at the moment a single state fails.

```
failhard: False
```

`state_verbose`

Default: `True`

Controls the verbosity of state runs. By default, the results of all states are returned, but setting this value to False will cause salt to only display output for states that failed or states that have changes.

```
state_verbose: False
```

state_output

Default: full

The state_output setting controls which results will be output full multi line:

- full, terse - each state will be full/terse
- mixed - only states with errors will be full
- changes - states with changes and errors will be full

full_id, mixed_id, changes_id and terse_id are also allowed; when set, the state ID will be used as name in the output.

```
state_output: full
```

state_output_diff

Default: False

The state_output_diff setting changes whether or not the output from successful states is returned. Useful when even the terse output of these states is cluttering the logs. Set it to True to ignore them.

```
state_output_diff: False
```

state_aggregate

Default: False

Automatically aggregate all states that have support for mod_aggregate by setting to True. Or pass a list of state module names to automatically aggregate just those types.

```
state_aggregate:
  - pkg
```

```
state_aggregate: True
```

state_events

Default: False

Send progress events as each function in a state run completes execution by setting to True. Progress events are in the format salt/job/<JID>/prog/<MID>/<RUN NUM>.

```
state_events: True
```

yaml_utf8

Default: False

Enable extra routines for YAML renderer used states containing UTF characters.

```
yaml_utf8: False
```

runner_returns

Default: False

If set to True, runner jobs will be saved to job cache (defined by *master_job_cache*).

```
runner_returns: True
```

3.1.7 Master File Server Settings

fileserver_backend

Default: ['roots']

Salt supports a modular fileserver backend system, this system allows the salt master to link directly to third party systems to gather and manage the files available to minions. Multiple backends can be configured and will be searched for the requested file in the order in which they are defined here. The default setting only enables the standard backend roots, which is configured using the *file_roots* option.

Example:

```
fileserver_backend:
  - roots
  - gitfs
```

Note: For masterless Salt, this parameter must be specified in the minion config file.

fileserver_followsymlinks

New in version 2014.1.0.

Default: True

By default, the file_server follows symlinks when walking the filesystem tree. Currently this only applies to the default roots fileserver_backend.

```
fileserver_followsymlinks: True
```

fileserver_ignoresymlinks

New in version 2014.1.0.

Default: False

If you do not want symlinks to be treated as the files they are pointing to, set `fileserver_ignoresymlinks` to `True`. By default this is set to False. When set to `True`, any detected symlink while listing files on the Master will not be returned to the Minion.

```
fileserver_ignoresymlinks: False
```

`fileserver_limit_traversal`

New in version 2014.1.0.

Deprecated since version 2018.3.4: This option is now ignored. Firstly, it only traversed *`file_roots`*, which means it did not work for the other fileserver backends. Secondly, since this option was added we have added caching to the code that traverses the file_roots (and gitfs, etc.), which greatly reduces the amount of traversal that is done.

Default: `False`

By default, the Salt fileserver recurses fully into all defined environments to attempt to find files. To limit this behavior so that the fileserver only traverses directories with SLS files and special Salt directories like _modules, set `fileserver_limit_traversal` to `True`. This might be useful for installations where a file root has a very large number of files and performance is impacted.

```
fileserver_limit_traversal: False
```

`fileserver_list_cache_time`

New in version 2014.1.0.

Changed in version 2016.11.0: The default was changed from `30` seconds to `20`.

Default: `20`

Salt caches the list of files/symlinks/directories for each fileserver backend and environment as they are requested, to guard against a performance bottleneck at scale when many minions all ask the fileserver which files are available simultaneously. This configuration parameter allows for the max age of that cache to be altered.

Set this value to `0` to disable use of this cache altogether, but keep in mind that this may increase the CPU load on the master when running a highstate on a large number of minions.

Note: Rather than altering this configuration parameter, it may be advisable to use the `fileserver.clear_list_cache` runner to clear these caches.

```
fileserver_list_cache_time: 5
```

`fileserver_verify_config`

New in version 2017.7.0.

Default: `True`

By default, as the master starts it performs some sanity checks on the configured fileserver backends. If any of these sanity checks fail (such as when an invalid configuration is used), the master daemon will abort.

To skip these sanity checks, set this option to `False`.

```
fileserver_verify_config: False
```

hash_type

Default: sha256

The hash_type is the hash to use when discovering the hash of a file on the master server. The default is sha256, but md5, sha1, sha224, sha384, and sha512 are also supported.

```
hash_type: sha256
```

file_buffer_size

Default: 1048576

The buffer size in the file server in bytes.

```
file_buffer_size: 1048576
```

file_ignore_regex

Default: ''

A regular expression (or a list of expressions) that will be matched against the file path before syncing the modules and states to the minions. This includes files affected by the file.recurse state. For example, if you manage your custom modules and states in subversion and don't want all the `.svn' folders and content synced to your minions, you could set this to `/.svn($|/)'. By default nothing is ignored.

```
file_ignore_regex:
  - '/\.svn($|/)'
  - '/\.git($|/)'
```

file_ignore_glob

Default ''

A file glob (or list of file globs) that will be matched against the file path before syncing the modules and states to the minions. This is similar to file_ignore_regex above, but works on globs instead of regex. By default nothing is ignored.

```
file_ignore_glob:
  - '\*.pyc'
  - '\*/somefolder/\*.bak'
  - '\*.swp'
```

Note: Vim's .swp files are a common cause of Unicode errors in *file.recurse* states which use templating. Unless there is a good reason to distribute them via the fileserver, it is good practice to include '*.swp' in the *file_ignore_glob*.

master_roots

Default: `/srv/salt-master`

A master-only copy of the *file_roots* dictionary, used by the state compiler.

```
master_roots: /srv/salt-master
```

roots: Master's Local File Server

file_roots

Default:

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

Salt runs a lightweight file server written in ZeroMQ to deliver files to minions. This file server is built into the master daemon and does not require a dedicated port.

The file server works on environments passed to the master. Each environment can have multiple root directories. The subdirectories in the multiple file roots cannot match, otherwise the downloaded files will not be able to be reliably ensured. A base environment is required to house the top file.

Example:

```
file_roots:
  base:
    - /srv/salt
  dev:
    - /srv/salt/dev/services
    - /srv/salt/dev/states
  prod:
    - /srv/salt/prod/services
    - /srv/salt/prod/states
```

Note: For masterless Salt, this parameter must be specified in the minion config file.

roots_update_interval

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for *file_roots*.

Note: Since `file_roots` consists of files local to the minion, the update process for this fileserver backend just reaps the cache for this backend.

```
roots_update_interval: 120
```

gitfs: Git Remote File Server Backend

gitfs_remotes

Default: []

When using the git fileserver backend at least one git remote needs to be defined. The user running the salt master will need read access to the repo.

The repos will be searched in order to find the file requested by a client and the first repo to have the file will return it. Branches and tags are translated into salt environments.

```
gitfs_remotes:
  - git://github.com/saltstack/salt-states.git
  - file:///var/git/saltmaster
```

Note: file:// repos will be treated as a remote and copied into the master's gitfs cache, so only the *local* refs for those repos will be exposed as fileserver environments.

As of 2014.7.0, it is possible to have per-repo versions of several of the gitfs configuration parameters. For more information, see the *GitFS Walkthrough*.

gitfs_provider

New in version 2014.7.0.

Optional parameter used to specify the provider to be used for gitfs. More information can be found in the *GitFS Walkthrough*.

Must be either pygit2 or gitpython. If unset, then each will be tried in that same order, and the first one with a compatible version installed will be the provider that is used.

```
gitfs_provider: gitpython
```

gitfs_ssl_verify

Default: True

Specifies whether or not to ignore SSL certificate errors when fetching from the repositories configured in *gitfs_remotes*. The False setting is useful if you're using a git repo that uses a self-signed certificate. However, keep in mind that setting this to anything other True is a considered insecure, and using an SSH-based transport (if available) may be a better option.

```
gitfs_ssl_verify: False
```

Note: pygit2 only supports disabling SSL verification in versions 0.23.2 and newer.

Changed in version 2015.8.0: This option can now be configured on individual repositories as well. See *here* for more info.

Changed in version 2016.11.0: The default config value changed from False to True.

gitfs_mountpoint

New in version 2014.7.0.

Default: `''`

Specifies a path on the salt fileserver which will be prepended to all files served by gitfs. This option can be used in conjunction with *gitfs_root*. It can also be configured for an individual repository, see *here* for more info.

```
gitfs_mountpoint: salt://foo/bar
```

Note: The `salt://` protocol designation can be left off (in other words, `foo/bar` and `salt://foo/bar` are equivalent). Assuming a file `baz.sh` in the root of a gitfs remote, and the above example mountpoint, this file would be served up via `salt://foo/bar/baz.sh`.

gitfs_root

Default: `''`

Relative path to a subdirectory within the repository from which Salt should begin to serve files. This is useful when there are files in the repository that should not be available to the Salt fileserver. Can be used in conjunction with *gitfs_mountpoint*. If used, then from Salt's perspective the directories above the one specified will be ignored and the relative path will (for the purposes of gitfs) be considered as the root of the repo.

```
gitfs_root: somefolder/otherfolder
```

Changed in version 2014.7.0: This option can now be configured on individual repositories as well. See *here* for more info.

gitfs_base

Default: `master`

Defines which branch/tag should be used as the `base` environment.

```
gitfs_base: salt
```

Changed in version 2014.7.0: This option can now be configured on individual repositories as well. See *here* for more info.

gitfs_saltenv

New in version 2016.11.0.

Default: `[]`

Global settings for *per-saltenv configuration parameters*. Though per-saltenv configuration parameters are typically one-off changes specific to a single gitfs remote, and thus more often configured on a per-remote basis, this parameter can be used to specify per-saltenv changes which should apply to all remotes. For example, the below configuration will map the `develop` branch to the `dev` saltenv for all gitfs remotes.

```
gitfs_saltenv:
  - dev:
    - ref: develop
```

gitfs_disable_saltenv_mapping

New in version 2018.3.0.

Default: `False`

When set to `True`, all saltenv mapping logic is disregarded (aside from which branch/tag is mapped to the `base` saltenv). To use any other environments, they must then be defined using *per-saltenv configuration parameters*.

```
gitfs_disable_saltenv_mapping: True
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_ref_types

New in version 2018.3.0.

Default: `['branch','tag','sha']`

This option defines what types of refs are mapped to fileserver environments (i.e. saltenvs). It also sets the order of preference when there are ambiguously-named refs (i.e. when a branch and tag both have the same name). The below example disables mapping of both tags and SHAs, so that only branches are mapped as saltenvs:

```
gitfs_ref_types:
  - branch
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

Note: `sha` is special in that it will not show up when listing saltenvs (e.g. with the *fileserver.envs* runner), but works within states and with *cp.cache_file* to retrieve a file from a specific git SHA.

gitfs_saltenv_whitelist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from `gitfs_env_whitelist` to `gitfs_saltenv_whitelist`

Default: `[]`

Used to restrict which environments are made available. Can speed up state runs if the repos in *gitfs_remotes* contain many branches/tags. More information can be found in the *GitFS Walkthrough*.

```
gitfs_saltenv_whitelist:
  - base
```

```
  - v1.*
  - 'mybranch\d+'
```

gitfs_saltenv_blacklist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from `gitfs_env_blacklist` to `gitfs_saltenv_blacklist`

Default: `[]`

Used to restrict which environments are made available. Can speed up state runs if the repos in *gitfs_remotes* contain many branches/tags. More information can be found in the *GitFS Walkthrough*.

```
gitfs_saltenv_blacklist:
  - base
  - v1.*
  - 'mybranch\d+'
```

gitfs_global_lock

New in version 2015.8.9.

Default: `True`

When set to `False`, if there is an update lock for a gitfs remote and the pid written to it is not running on the master, the lock file will be automatically cleared and a new lock will be obtained. When set to `True`, Salt will simply log a warning when there is an update lock present.

On single-master deployments, disabling this option can help automatically deal with instances where the master was shutdown/restarted during the middle of a gitfs update, leaving a update lock in place.

However, on multi-master deployments with the gitfs cachedir shared via GlusterFS, nfs, or another network filesystem, it is strongly recommended not to disable this option as doing so will cause lock files to be removed if they were created by a different master.

```
# Disable global lock
gitfs_global_lock: False
```

gitfs_update_interval

New in version 2018.3.0.

Default: `60`

This option defines the default update interval (in seconds) for gitfs remotes. The update interval can also be set for a single repository via a *per-remote config option*

```
gitfs_update_interval: 120
```

GitFS Authentication Options

These parameters only currently apply to the pygit2 gitfs provider. Examples of how to use these can be found in the *GitFS Walkthrough*.

gitfs_user

New in version 2014.7.0.

Default: `''`

Along with *gitfs_password*, is used to authenticate to HTTPS remotes.

```
gitfs_user: git
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_password

New in version 2014.7.0.

Default: `''`

Along with *gitfs_user*, is used to authenticate to HTTPS remotes. This parameter is not required if the repository does not use authentication.

```
gitfs_password: mypassword
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_insecure_auth

New in version 2014.7.0.

Default: `False`

By default, Salt will not authenticate to an HTTP (non-HTTPS) remote. This parameter enables authentication over HTTP. **Enable this at your own risk.**

```
gitfs_insecure_auth: True
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_pubkey

New in version 2014.7.0.

Default: `''`

Along with *gitfs_privkey* (and optionally *gitfs_passphrase*), is used to authenticate to SSH remotes. Required for SSH remotes.

```
gitfs_pubkey: /path/to/key.pub
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_privkey

New in version 2014.7.0.

Default: `''`

Along with *gitfs_pubkey* (and optionally *gitfs_passphrase*), is used to authenticate to SSH remotes. Required for SSH remotes.

```
gitfs_privkey: /path/to/key
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_passphrase

New in version 2014.7.0.

Default: `''`

This parameter is optional, required only when the SSH key being used to authenticate is protected by a passphrase.

```
gitfs_passphrase: mypassphrase
```

Note: This is is a global configuration option, see *here* for examples of configuring it for individual repositories.

gitfs_refspecs

New in version 2017.7.0.

Default: `['+refs/heads/*:refs/remotes/origin/*','+refs/tags/*:refs/tags/*']`

When fetching from remote repositories, by default Salt will fetch branches and tags. This parameter can be used to override the default and specify alternate refspecs to be fetched. More information on how this feature works can be found in the *GitFS Walkthrough*.

```
gitfs_refspecs:
  - '+refs/heads/*:refs/remotes/origin/*'
  - '+refs/tags/*:refs/tags/*'
  - '+refs/pull/*/head:refs/remotes/origin/pr/*'
  - '+refs/pull/*/merge:refs/remotes/origin/merge/*'
```

hgfs: Mercurial Remote File Server Backend

hgfs_remotes

New in version 0.17.0.

Default: `[]`

When using the `hg` fileserver backend at least one mercurial remote needs to be defined. The user running the salt master will need read access to the repo.

The repos will be searched in order to find the file requested by a client and the first repo to have the file will return it. Branches and/or bookmarks are translated into salt environments, as defined by the *hgfs_branch_method* parameter.

```
hgfs_remotes:
  - https://username@bitbucket.org/username/reponame
```

Note: As of 2014.7.0, it is possible to have per-repo versions of the *hgfs_root*, *hgfs_mountpoint*, *hgfs_base*, and *hgfs_branch_method* parameters. For example:

```
hgfs_remotes:
  - https://username@bitbucket.org/username/repo1
    - base: saltstates
  - https://username@bitbucket.org/username/repo2:
    - root: salt
    - mountpoint: salt://foo/bar/baz
  - https://username@bitbucket.org/username/repo3:
    - root: salt/states
    - branch_method: mixed
```

hgfs_branch_method

New in version 0.17.0.

Default: `branches`

Defines the objects that will be used as fileserver environments.

- `branches` - Only branches and tags will be used
- `bookmarks` - Only bookmarks and tags will be used
- `mixed` - Branches, bookmarks, and tags will be used

```
hgfs_branch_method: mixed
```

Note: Starting in version 2014.1.0, the value of the *hgfs_base* parameter defines which branch is used as the `base` environment, allowing for a `base` environment to be used with an *hgfs_branch_method* of `bookmarks`.

Prior to this release, the `default` branch will be used as the `base` environment.

hgfs_mountpoint

New in version 2014.7.0.

Default: `''`

Specifies a path on the salt fileserver which will be prepended to all files served by hgfs. This option can be used in conjunction with *hgfs_root*. It can also be configured on a per-remote basis, see *here* for more info.

```
hgfs_mountpoint: salt://foo/bar
```

Note: The salt:// protocol designation can be left off (in other words, foo/bar and salt://foo/bar are equivalent). Assuming a file baz.sh in the root of an hgfs remote, this file would be served up via salt://foo/bar/baz.sh.

hgfs_root

New in version 0.17.0.

Default: ''

Relative path to a subdirectory within the repository from which Salt should begin to serve files. This is useful when there are files in the repository that should not be available to the Salt fileserver. Can be used in conjunction with *hgfs_mountpoint*. If used, then from Salt's perspective the directories above the one specified will be ignored and the relative path will (for the purposes of hgfs) be considered as the root of the repo.

```
hgfs_root: somefolder/otherfolder
```

Changed in version 2014.7.0: Ability to specify hgfs roots on a per-remote basis was added. See *here* for more info.

hgfs_base

New in version 2014.1.0.

Default: default

Defines which branch should be used as the base environment. Change this if *hgfs_branch_method* is set to bookmarks to specify which bookmark should be used as the base environment.

```
hgfs_base: salt
```

hgfs_saltenv_whitelist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from hgfs_env_whitelist to hgfs_saltenv_whitelist

Default: []

Used to restrict which environments are made available. Can speed up state runs if your hgfs remotes contain many branches/bookmarks/tags. Full names, globs, and regular expressions are supported. If using a regular expression, the expression must match the entire minion ID.

If used, only branches/bookmarks/tags which match one of the specified expressions will be exposed as fileserver environments.

If used in conjunction with *hgfs_saltenv_blacklist*, then the subset of branches/bookmarks/tags which match the whitelist but do *not* match the blacklist will be exposed as fileserver environments.

```
hgfs_saltenv_whitelist:
  - base
  - v1.*
  - 'mybranch\d+'
```

hgfs_saltenv_blacklist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from `hgfs_env_blacklist` to `hgfs_saltenv_blacklist`

Default: `[]`

Used to restrict which environments are made available. Can speed up state runs if your hgfs remotes contain many branches/bookmarks/tags. Full names, globs, and regular expressions are supported. If using a regular expression, the expression must match the entire minion ID.

If used, branches/bookmarks/tags which match one of the specified expressions will *not* be exposed as fileserver environments.

If used in conjunction with *hgfs_saltenv_whitelist*, then the subset of branches/bookmarks/tags which match the whitelist but do *not* match the blacklist will be exposed as fileserver environments.

```
hgfs_saltenv_blacklist:
  - base
  - v1.*
  - 'mybranch\d+'
```

hgfs_update_interval

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for *hgfs_remotes*.

```
hgfs_update_interval: 120
```

svnfs: Subversion Remote File Server Backend

svnfs_remotes

New in version 0.17.0.

Default: `[]`

When using the `svn` fileserver backend at least one subversion remote needs to be defined. The user running the salt master will need read access to the repo.

The repos will be searched in order to find the file requested by a client and the first repo to have the file will return it. The trunk, branches, and tags become environments, with the trunk being the `base` environment.

```
svnfs_remotes:
  - svn://foo.com/svn/myproject
```

Note: As of 2014.7.0, it is possible to have per-repo versions of the following configuration parameters:

- *svnfs_root*
- *svnfs_mountpoint*
- *svnfs_trunk*
- *svnfs_branches*
- *svnfs_tags*

For example:

```
svnfs_remotes:
  - svn://foo.com/svn/project1
  - svn://foo.com/svn/project2:
    - root: salt
    - mountpoint: salt://foo/bar/baz
  - svn//foo.com/svn/project3:
    - root: salt/states
    - branches: branch
    - tags: tag
```

svnfs_mountpoint

New in version 2014.7.0.

Default: `''`

Specifies a path on the salt fileserver which will be prepended to all files served by hgfs. This option can be used in conjunction with *svnfs_root*. It can also be configured on a per-remote basis, see *here* for more info.

```
svnfs_mountpoint: salt://foo/bar
```

Note: The `salt://` protocol designation can be left off (in other words, `foo/bar` and `salt://foo/bar` are equivalent). Assuming a file `baz.sh` in the root of an svnfs remote, this file would be served up via `salt://foo/bar/baz.sh`.

svnfs_root

New in version 0.17.0.

Default: `''`

Relative path to a subdirectory within the repository from which Salt should begin to serve files. This is useful when there are files in the repository that should not be available to the Salt fileserver. Can be used in conjunction with *svnfs_mountpoint*. If used, then from Salt's perspective the directories above the one specified will be ignored and the relative path will (for the purposes of svnfs) be considered as the root of the repo.

```
svnfs_root: somefolder/otherfolder
```

Changed in version 2014.7.0: Ability to specify svnfs roots on a per-remote basis was added. See *here* for more info.

svnfs_trunk

New in version 2014.7.0.

Default: `trunk`

Path relative to the root of the repository where the trunk is located. Can also be configured on a per-remote basis, see *here* for more info.

```
svnfs_trunk: trunk
```

svnfs_branches

New in version 2014.7.0.

Default: `branches`

Path relative to the root of the repository where the branches are located. Can also be configured on a per-remote basis, see *here* for more info.

```
svnfs_branches: branches
```

svnfs_tags

New in version 2014.7.0.

Default: `tags`

Path relative to the root of the repository where the tags are located. Can also be configured on a per-remote basis, see *here* for more info.

```
svnfs_tags: tags
```

svnfs_saltenv_whitelist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from `svnfs_env_whitelist` to `svnfs_saltenv_whitelist`

Default: `[]`

Used to restrict which environments are made available. Can speed up state runs if your svnfs remotes contain many branches/tags. Full names, globs, and regular expressions are supported. If using a regular expression, the expression must match the entire minion ID.

If used, only branches/tags which match one of the specified expressions will be exposed as fileserver environments.

If used in conjunction with *svnfs_saltenv_blacklist*, then the subset of branches/tags which match the whitelist but do *not* match the blacklist will be exposed as fileserver environments.

```
svnfs_saltenv_whitelist:
  - base
  - v1.*
  - 'mybranch\d+'
```

svnfs_saltenv_blacklist

New in version 2014.7.0.

Changed in version 2018.3.0: Renamed from `svnfs_env_blacklist` to `svnfs_saltenv_blacklist`

Default: `[]`

Used to restrict which environments are made available. Can speed up state runs if your svnfs remotes contain many branches/tags. Full names, globs, and regular expressions are supported. If using a regular expression, the expression must match the entire minion ID.

If used, branches/tags which match one of the specified expressions will *not* be exposed as fileserver environments.

If used in conjunction with *svnfs_saltenv_whitelist*, then the subset of branches/tags which match the whitelist but do *not* match the blacklist will be exposed as fileserver environments.

```
svnfs_saltenv_blacklist:
  - base
  - v1.*
  - 'mybranch\d+'
```

svnfs_update_interval

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for *svnfs_remotes*.

```
svnfs_update_interval: 120
```

minionfs: MinionFS Remote File Server Backend

minionfs_env

New in version 2014.7.0.

Default: `base`

Environment from which MinionFS files are made available.

```
minionfs_env: minionfs
```

minionfs_mountpoint

New in version 2014.7.0.

Default: `''`

Specifies a path on the salt fileserver from which minionfs files are served.

```
minionfs_mountpoint: salt://foo/bar
```

Note: The `salt://` protocol designation can be left off (in other words, `foo/bar` and `salt://foo/bar` are equivalent).

minionfs_whitelist

New in version 2014.7.0.

Default: `[]`

Used to restrict which minions' pushed files are exposed via minionfs. If using a regular expression, the expression must match the entire minion ID.

If used, only the pushed files from minions which match one of the specified expressions will be exposed.

If used in conjunction with *minionfs_blacklist*, then the subset of hosts which match the whitelist but do *not* match the blacklist will be exposed.

```
minionfs_whitelist:
  - server01
  - dev*
  - 'mail\d+.mydomain.tld'
```

minionfs_blacklist

New in version 2014.7.0.

Default: `[]`

Used to restrict which minions' pushed files are exposed via minionfs. If using a regular expression, the expression must match the entire minion ID.

If used, only the pushed files from minions which match one of the specified expressions will *not* be exposed.

If used in conjunction with *minionfs_whitelist*, then the subset of hosts which match the whitelist but do *not* match the blacklist will be exposed.

```
minionfs_blacklist:
  - server01
  - dev*
  - 'mail\d+.mydomain.tld'
```

minionfs_update_interval

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for *MinionFS*.

Note: Since *MinionFS* consists of files local to the master, the update process for this fileserver backend just reaps the cache for this backend.

```
minionfs_update_interval: 120
```

azurefs: Azure File Server Backend

New in version 2015.8.0.

See the *azurefs documentation* for usage examples.

`azurefs_update_interval`

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for azurefs.

```
azurefs_update_interval: 120
```

s3fs: S3 File Server Backend

New in version 0.16.0.

See the *s3fs documentation* for usage examples.

`s3fs_update_interval`

New in version 2018.3.0.

Default: `60`

This option defines the update interval (in seconds) for s3fs.

```
s3fs_update_interval: 120
```

3.1.8 Pillar Configuration

`pillar_roots`

Default:

```
base:
  - /srv/pillar
```

Set the environments and directories used to hold pillar sls data. This configuration is the same as *file_roots*:

```
pillar_roots:
  base:
    - /srv/pillar
  dev:
    - /srv/pillar/dev
  prod:
    - /srv/pillar/prod
```

on_demand_ext_pillar

New in version 2016.3.6,2016.11.3,2017.7.0.

Default: `['libvirt','virtkey']`

The external pillars permitted to be used on-demand using `pillar.ext`.

```
on_demand_ext_pillar:
  - libvirt
  - virtkey
  - git
```

Warning: This will allow minions to request specific pillar data via `pillar.ext`, and may be considered a security risk. However, pillar data generated in this way will not affect the *in-memory pillar data*, so this risk is limited to instances in which states/modules/etc. (built-in or custom) rely upon pillar data generated by `pillar.ext`.

decrypt_pillar

New in version 2017.7.0.

Default: `[]`

A list of paths to be recursively decrypted during pillar compilation.

```
decrypt_pillar:
  - 'foo:bar': gpg
  - 'lorem:ipsum:dolor'
```

Entries in this list can be formatted either as a simple string, or as a key/value pair, with the key being the pillar location, and the value being the renderer to use for pillar decryption. If the former is used, the renderer specified by `decrypt_pillar_default` will be used.

decrypt_pillar_delimiter

New in version 2017.7.0.

Default: `:`

The delimiter used to distinguish nested data structures in the `decrypt_pillar` option.

```
decrypt_pillar_delimiter: '|'
decrypt_pillar:
  - 'foo|bar': gpg
  - 'lorem|ipsum|dolor'
```

decrypt_pillar_default

New in version 2017.7.0.

Default: `gpg`

The default renderer used for decryption, if one is not specified for a given pillar key in `decrypt_pillar`.

```
decrypt_pillar_default: my_custom_renderer
```

decrypt_pillar_renderers

New in version 2017.7.0.

Default: `['gpg']`

List of renderers which are permitted to be used for pillar decryption.

```
decrypt_pillar_renderers:
  - gpg
  - my_custom_renderer
```

pillar_opts

Default: `False`

The `pillar_opts` option adds the master configuration file data to a dict in the pillar called `master`. This can be used to set simple configurations in the master config file that can then be used on minions.

Note that setting this option to `True` means the master config file will be included in all minion's pillars. While this makes global configuration of services and systems easy, it may not be desired if sensitive data is stored in the master configuration.

```
pillar_opts: False
```

pillar_safe_render_error

Default: `True`

The pillar_safe_render_error option prevents the master from passing pillar render errors to the minion. This is set on by default because the error could contain templating data which would give that minion information it shouldn't have, like a password! When set `True` the error message will only show:

```
Rendering SLS 'my.sls' failed. Please see master log for details.
```

```
pillar_safe_render_error: True
```

ext_pillar

The ext_pillar option allows for any number of external pillar interfaces to be called when populating pillar data. The configuration is based on ext_pillar functions. The available ext_pillar functions can be found herein:

https://github.com/saltstack/salt/blob/develop/salt/pillar

By default, the ext_pillar interface is not configured to run.

Default: `[]`

```
ext_pillar:
  - hiera: /etc/hiera.yaml
  - cmd_yaml: cat /etc/salt/yaml
```

```
  - reclass:
      inventory_base_uri: /etc/reclass
```

There are additional details at *Pillars*

ext_pillar_first

New in version 2015.5.0.

Default: `False`

This option allows for external pillar sources to be evaluated before *pillar_roots*. External pillar data is evaluated separately from *pillar_roots* pillar data, and then both sets of pillar data are merged into a single pillar dictionary, so the value of this config option will have an impact on which key ``wins'' when there is one of the same name in both the external pillar data and *pillar_roots* pillar data. By setting this option to `True`, ext_pillar keys will be overridden by *pillar_roots*, while leaving it as `False` will allow ext_pillar keys to override those from *pillar_roots*.

Note: For a while, this config option did not work as specified above, because of a bug in Pillar compilation. This bug has been resolved in version 2016.3.4 and later.

```
ext_pillar_first: False
```

pillarenv_from_saltenv

Default: `False`

When set to `True`, the `pillarenv` value will assume the value of the effective saltenv when running states. This essentially makes `salt-run pillar.show_pillar saltenv=dev` equivalent to `salt-run pillar.show_pillar saltenv=dev pillarenv=dev`. If `pillarenv` is set on the CLI, it will override this option.

```
pillarenv_from_saltenv: True
```

Note: For salt remote execution commands this option should be set in the Minion configuration instead.

pillar_raise_on_missing

New in version 2015.5.0.

Default: `False`

Set this option to `True` to force a `KeyError` to be raised whenever an attempt to retrieve a named value from pillar fails. When this option is set to `False`, the failed attempt returns an empty string.

Git External Pillar (git_pillar) Configuration Options

git_pillar_provider

New in version 2015.8.0.

Specify the provider to be used for git_pillar. Must be either pygit2 or gitpython. If unset, then both will be tried in that same order, and the first one with a compatible version installed will be the provider that is used.

```
git_pillar_provider: gitpython
```

git_pillar_base

New in version 2015.8.0.

Default: master

If the desired branch matches this value, and the environment is omitted from the git_pillar configuration, then the environment for that git_pillar remote will be base. For example, in the configuration below, the foo branch/tag would be assigned to the base environment, while bar would be mapped to the bar environment.

```
git_pillar_base: foo

ext_pillar:
  - git:
    - foo https://mygitserver/git-pillar.git
    - bar https://mygitserver/git-pillar.git
```

git_pillar_branch

New in version 2015.8.0.

Default: master

If the branch is omitted from a git_pillar remote, then this branch will be used instead. For example, in the configuration below, the first two remotes would use the pillardata branch/tag, while the third would use the foo branch/tag.

```
git_pillar_branch: pillardata

ext_pillar:
  - git:
    - https://mygitserver/pillar1.git
    - https://mygitserver/pillar2.git:
      - root: pillar
    - foo https://mygitserver/pillar3.git
```

git_pillar_env

New in version 2015.8.0.

Default: '' (unset)

Environment to use for git_pillar remotes. This is normally derived from the branch/tag (or from a per-remote env parameter), but if set this will override the process of deriving the env from the branch/tag name. For example, in the configuration below the foo branch would be assigned to the base environment, while the bar branch would need to explicitly have bar configured as it's environment to keep it from also being mapped to the base environment.

```
git_pillar_env: base
```

```
ext_pillar:
  - git:
    - foo https://mygitserver/git-pillar.git
    - bar https://mygitserver/git-pillar.git:
      - env: bar
```

For this reason, this option is recommended to be left unset, unless the use case calls for all (or almost all) of the git_pillar remotes to use the same environment irrespective of the branch/tag being used.

git_pillar_root

New in version 2015.8.0.

Default: `''`

Path relative to the root of the repository where the git_pillar top file and SLS files are located. In the below configuration, the pillar top file and SLS files would be looked for in a subdirectory called `pillar`.

```
git_pillar_root: pillar

ext_pillar:
  - git:
    - master https://mygitserver/pillar1.git
    - master https://mygitserver/pillar2.git
```

Note: This is a global option. If only one or two repos need to have their files sourced from a subdirectory, then *git_pillar_root* can be omitted and the root can be specified on a per-remote basis, like so:

```
ext_pillar:
  - git:
    - master https://mygitserver/pillar1.git
    - master https://mygitserver/pillar2.git:
      - root: pillar
```

In this example, for the first remote the top file and SLS files would be looked for in the root of the repository, while in the second remote the pillar data would be retrieved from the `pillar` subdirectory.

git_pillar_ssl_verify

New in version 2015.8.0.

Changed in version 2016.11.0.

Default: `False`

Specifies whether or not to ignore SSL certificate errors when contacting the remote repository. The `False` setting is useful if you're using a git repo that uses a self-signed certificate. However, keep in mind that setting this to anything other `True` is a considered insecure, and using an SSH-based transport (if available) may be a better option.

In the 2016.11.0 release, the default config value changed from `False` to `True`.

```
git_pillar_ssl_verify: True
```

Note: pygit2 only supports disabling SSL verification in versions 0.23.2 and newer.

git_pillar_global_lock

New in version 2015.8.9.

Default: `True`

When set to `False`, if there is an update/checkout lock for a git_pillar remote and the pid written to it is not running on the master, the lock file will be automatically cleared and a new lock will be obtained. When set to `True`, Salt will simply log a warning when there is an lock present.

On single-master deployments, disabling this option can help automatically deal with instances where the master was shutdown/restarted during the middle of a git_pillar update/checkout, leaving a lock in place.

However, on multi-master deployments with the git_pillar cachedir shared via GlusterFS, nfs, or another network filesystem, it is strongly recommended not to disable this option as doing so will cause lock files to be removed if they were created by a different master.

```
# Disable global lock
git_pillar_global_lock: False
```

git_pillar_includes

New in version 2017.7.0.

Default: `True`

Normally, when processing *git_pillar remotes*, if more than one repo under the same `git` section in the `ext_pillar` configuration refers to the same pillar environment, then each repo in a given environment will have access to the other repos' files to be referenced in their top files. However, it may be desirable to disable this behavior. If so, set this value to `False`.

For a more detailed examination of how includes work, see *this explanation* from the git_pillar documentation.

```
git_pillar_includes: False
```

Git External Pillar Authentication Options

These parameters only currently apply to the `pygit2` *git_pillar_provider*. Authentication works the same as it does in gitfs, as outlined in the *GitFS Walkthrough*, though the global configuration options are named differently to reflect that they are for git_pillar instead of gitfs.

git_pillar_user

New in version 2015.8.0.

Default: `''`

Along with *git_pillar_password*, is used to authenticate to HTTPS remotes.

```
git_pillar_user: git
```

git_pillar_password

New in version 2015.8.0.

Default: `''`

Along with *git_pillar_user*, is used to authenticate to HTTPS remotes. This parameter is not required if the repository does not use authentication.

```
git_pillar_password: mypassword
```

git_pillar_insecure_auth

New in version 2015.8.0.

Default: `False`

By default, Salt will not authenticate to an HTTP (non-HTTPS) remote. This parameter enables authentication over HTTP. **Enable this at your own risk.**

```
git_pillar_insecure_auth: True
```

git_pillar_pubkey

New in version 2015.8.0.

Default: `''`

Along with *git_pillar_privkey* (and optionally *git_pillar_passphrase*), is used to authenticate to SSH remotes.

```
git_pillar_pubkey: /path/to/key.pub
```

git_pillar_privkey

New in version 2015.8.0.

Default: `''`

Along with *git_pillar_pubkey* (and optionally *git_pillar_passphrase*), is used to authenticate to SSH remotes.

```
git_pillar_privkey: /path/to/key
```

git_pillar_passphrase

New in version 2015.8.0.

Default: `''`

This parameter is optional, required only when the SSH key being used to authenticate is protected by a passphrase.

```
git_pillar_passphrase: mypassphrase
```

git_pillar_refspecs

New in version 2017.7.0.

Default: `['+refs/heads/*:refs/remotes/origin/*','+refs/tags/*:refs/tags/*']`

When fetching from remote repositories, by default Salt will fetch branches and tags. This parameter can be used to override the default and specify alternate refspecs to be fetched. This parameter works similarly to its *GitFS counterpart*, in that it can be configured both globally and for individual remotes.

```
git_pillar_refspecs:
  - '+refs/heads/*:refs/remotes/origin/*'
  - '+refs/tags/*:refs/tags/*'
  - '+refs/pull/*/head:refs/remotes/origin/pr/*'
  - '+refs/pull/*/merge:refs/remotes/origin/merge/*'
```

git_pillar_verify_config

New in version 2017.7.0.

Default: `True`

By default, as the master starts it performs some sanity checks on the configured git_pillar repositories. If any of these sanity checks fail (such as when an invalid configuration is used), the master daemon will abort.

To skip these sanity checks, set this option to `False`.

```
git_pillar_verify_config: False
```

Pillar Merging Options

pillar_source_merging_strategy

New in version 2014.7.0.

Default: `smart`

The pillar_source_merging_strategy option allows you to configure merging strategy between different sources. It accepts 5 values:

- `none`:

  It will not do any merging at all and only parse the pillar data from the passed environment and `base' if no environment was specified.

  New in version 2016.3.4.

- `recurse`:

  It will recursively merge data. For example, theses 2 sources:

```
foo: 42
bar:
    element1: True
```

```
bar:
    element2: True
baz: quux
```

will be merged as:

```
foo: 42
bar:
    element1: True
    element2: True
baz: quux
```

- `aggregate`:

  instructs aggregation of elements between sources that use the #!yamlex renderer.

  For example, these two documents:

```
#!yamlex
foo: 42
bar: !aggregate {
  element1: True
}
baz: !aggregate quux
```

```
#!yamlex
bar: !aggregate {
  element2: True
}
baz: !aggregate quux2
```

will be merged as:

```
foo: 42
bar:
  element1: True
  element2: True
baz:
  - quux
  - quux2
```

- `overwrite`:

  Will use the behaviour of the 2014.1 branch and earlier.

  Overwrites elements according the order in which they are processed.

  First pillar processed:

```
A:
  first_key: blah
  second_key: blah
```

Second pillar processed:

```
A:
  third_key: blah
  fourth_key: blah
```

will be merged as:

```
A:
  third_key: blah
  fourth_key: blah
```

- `smart` (default):

  Guesses the best strategy based on the ``renderer'' setting.

pillar_merge_lists

New in version 2015.8.0.

Default: `False`

Recursively merge lists by aggregating them instead of replacing them.

```
pillar_merge_lists: False
```

pillar_includes_override_sls

New in version 2017.7.6,2018.3.1.

Default: `False`

Prior to version 2017.7.3, keys from *pillar includes* would be merged on top of the pillar SLS. Since 2017.7.3, the includes are merged together and then the pillar SLS is merged on top of that.

Set this option to `True` to return to the old behavior.

```
pillar_includes_override_sls: True
```

Pillar Cache Options

pillar_cache

New in version 2015.8.8.

Default: `False`

A master can cache pillars locally to bypass the expense of having to render them for each minion on every request. This feature should only be enabled in cases where pillar rendering time is known to be unsatisfactory and any attendant security concerns about storing pillars in a master cache have been addressed.

When enabling this feature, be certain to read through the additional `pillar_cache_*` configuration options to fully understand the tunable parameters and their implications.

```
pillar_cache: False
```

Note: Setting `pillar_cache: True` has no effect on *targeting minions with pillar*.

pillar_cache_ttl

New in version 2015.8.8.

Default: `3600`

If and only if a master has set `pillar_cache: True`, the cache TTL controls the amount of time, in seconds, before the cache is considered invalid by a master and a fresh pillar is recompiled and stored.

pillar_cache_backend

New in version 2015.8.8.

Default: `disk`

If an only if a master has set `pillar_cache: True`, one of several storage providers can be utilized:

- `disk` (default):

  The default storage backend. This caches rendered pillars to the master cache. Rendered pillars are serialized and deserialized as `msgpack` structures for speed. Note that pillars are stored UNENCRYPTED. Ensure that the master cache has permissions set appropriately (sane defaults are provided).

- `memory` [EXPERIMENTAL]:

  An optional backend for pillar caches which uses a pure-Python in-memory data structure for maximal performance. There are several caveats, however. First, because each master worker contains its own in-memory cache, there is no guarantee of cache consistency between minion requests. This works best in situations where the pillar rarely if ever changes. Secondly, and perhaps more importantly, this means that unencrypted pillars will be accessible to any process which can examine the memory of the `salt-master`! This may represent a substantial security risk.

```
pillar_cache_backend: disk
```

3.1.9 Master Reactor Settings

reactor

Default: `[]`

Defines a salt reactor. See the *Reactor* documentation for more information.

```
reactor:
  - 'salt/minion/*/start':
    - salt://reactor/startup_tasks.sls
```

reactor_refresh_interval

Default: `60`

The TTL for the cache of the reactor configuration.

```
reactor_refresh_interval: 60
```

reactor_worker_threads

Default: 10

The number of workers for the runner/wheel in the reactor.

```
reactor_worker_threads: 10
```

reactor_worker_hwm

Default: 10000

The queue size for workers in the reactor.

```
reactor_worker_hwm: 10000
```

3.1.10 Syndic Server Settings

A Salt syndic is a Salt master used to pass commands from a higher Salt master to minions below the syndic. Using the syndic is simple. If this is a master that will have syndic servers(s) below it, set the `order_masters` setting to `True`.

If this is a master that will be running a syndic daemon for passthrough the `syndic_master` setting needs to be set to the location of the master server.

Do not forget that, in other words, it means that it shares with the local minion its ID and PKI directory.

order_masters

Default: False

Extra data needs to be sent with publications if the master is controlling a lower level master via a syndic minion. If this is the case the order_masters value must be set to True

```
order_masters: False
```

syndic_master

Changed in version 2016.3.5,2016.11.1: Set default higher level master address.

Default: masterofmasters

If this master will be running the `salt-syndic` to connect to a higher level master, specify the higher level master with this configuration value.

```
syndic_master: masterofmasters
```

You can optionally connect a syndic to multiple higher level masters by setting the `syndic_master` value to a list:

```
syndic_master:
  - masterofmasters1
  - masterofmasters2
```

Each higher level master must be set up in a multi-master configuration.

syndic_master_port

Default: 4506

If this master will be running the salt-syndic to connect to a higher level master, specify the higher level master port with this configuration value.

```
syndic_master_port: 4506
```

syndic_pidfile

Default: /var/run/salt-syndic.pid

If this master will be running the salt-syndic to connect to a higher level master, specify the pidfile of the syndic daemon.

```
syndic_pidfile: /var/run/syndic.pid
```

syndic_log_file

Default: /var/log/salt/syndic

If this master will be running the salt-syndic to connect to a higher level master, specify the log file of the syndic daemon.

```
syndic_log_file: /var/log/salt-syndic.log
```

syndic_failover

New in version 2016.3.0.

Default: random

The behaviour of the multi-syndic when connection to a master of masters failed. Can specify random (default) or ordered. If set to random, masters will be iterated in random order. If ordered is specified, the configured order will be used.

```
syndic_failover: random
```

syndic_wait

Default: 5

The number of seconds for the salt client to wait for additional syndics to check in with their lists of expected minions before giving up.

```
syndic_wait: 5
```

syndic_forward_all_events

New in version 2017.7.0.

Default: `False`

Option on multi-syndic or single when connected to multiple masters to be able to send events to all connected masters.

```
syndic_forward_all_events: False
```

3.1.11 Peer Publish Settings

Salt minions can send commands to other minions, but only if the minion is allowed to. By default ``Peer Publication'' is disabled, and when enabled it is enabled for specific minions and specific commands. This allows secure compartmentalization of commands based on individual minions.

peer

Default: {}

The configuration uses regular expressions to match minions and then a list of regular expressions to match functions. The following will allow the minion authenticated as foo.example.com to execute functions from the test and pkg modules.

```
peer:
  foo.example.com:
      - test.*
      - pkg.*
```

This will allow all minions to execute all commands:

```
peer:
  .*:
      - .*
```

This is not recommended, since it would allow anyone who gets root on any single minion to instantly have root on all of the minions!

By adding an additional layer you can limit the target hosts in addition to the accessible commands:

```
peer:
  foo.example.com:
    'db*':
      - test.*
      - pkg.*
```

peer_run

Default: {}

The peer_run option is used to open up runners on the master to access from the minions. The peer_run configuration matches the format of the peer configuration.

The following example would allow foo.example.com to execute the manage.up runner:

```
peer_run:
  foo.example.com:
      - manage.up
```

3.1.12 Master Logging Settings

log_file

Default: `/var/log/salt/master`

The master log can be sent to a regular file, local path name, or network location. See also *log_file*.

Examples:

```
log_file: /var/log/salt/master
```

```
log_file: file:///dev/log
```

```
log_file: udp://loghost:10514
```

log_level

Default: `warning`

The level of messages to send to the console. See also *log_level*.

```
log_level: warning
```

log_level_logfile

Default: `warning`

The level of messages to send to the log file. See also *log_level_logfile*. When it is not set explicitly it will inherit the level set by *log_level* option.

```
log_level_logfile: warning
```

log_datefmt

Default: `%H:%M:%S`

The date and time format used in console log messages. See also *log_datefmt*.

```
log_datefmt: '%H:%M:%S'
```

log_datefmt_logfile

Default: %Y-%m-%d %H:%M:%S

The date and time format used in log file messages. See also *log_datefmt_logfile*.

```
log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'
```

log_fmt_console

Default: [%(levelname)-8s] %(message)s

The format of the console logging messages. See also *log_fmt_console*.

Note: Log colors are enabled in log_fmt_console rather than the *color* config since the logging system is loaded before the master config.

Console log colors are specified by these additional formatters:

%(colorlevel)s %(colorname)s %(colorprocess)s %(colormsg)s

Since it is desirable to include the surrounding brackets, `[' and `]', in the coloring of the messages, these color formatters also include padding as well. Color LogRecord attributes are only available for console logging.

```
log_fmt_console: '%(colorlevel)s %(colormsg)s'
log_fmt_console: '[%(levelname)-8s] %(message)s'
```

log_fmt_logfile

Default: %(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s

The format of the log file logging messages. See also *log_fmt_logfile*.

```
log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'
```

log_granular_levels

Default: {}

This can be used to control logging levels more specifically. See also *log_granular_levels*.

3.1.13 Node Groups

Default: {}

Node groups allow for logical groupings of minion nodes. A group consists of a group name and a compound target.

```
nodegroups:
  group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl*.domain.com'
  group2: 'G@os:Debian and foo.domain.com'
  group3: 'G@os:Debian and N@group1'
  group4:
    - 'G@foo:bar'
```

```
    - 'or'
    - 'G@foo:baz'
```

More information on using nodegroups can be found *here*.

3.1.14 Range Cluster Settings

range_server

Default: `'range:80'`

The range server (and optional port) that serves your cluster information https://github.com/ytoolshed/range/wiki/%22yamlfile%22-module-file-spec

```
range_server: range:80
```

3.1.15 Include Configuration

Configuration can be loaded from multiple files. The order in which this is done is:

1. The master config file itself
2. The files matching the glob in *default_include*
3. The files matching the glob in *include* (if defined)

Each successive step overrides any values defined in the previous steps. Therefore, any config options defined in one of the *default_include* files would override the same value in the master config file, and any options defined in *include* would override both.

default_include

Default: `master.d/*.conf`

The master can include configuration from other files. Per default the master will automatically include all config files from `master.d/*.conf` where `master.d` is relative to the directory of the master configuration file.

Note: Salt creates files in the `master.d` directory for its own use. These files are prefixed with an underscore. A common example of this is the `_schedule.conf` file.

include

Default: `not defined`

The master can include configuration from other files. To enable this, pass a list of paths to this option. The paths can be either relative or absolute; if relative, they are considered to be relative to the directory the main minion configuration file lives in. Paths can make use of shell-style globbing. If no files are matched by a path passed to this option then the master will log a warning message.

```
# Include files from a master.d directory in the same
# directory as the master config file
include: master.d/*

# Include a single extra file into the configuration
include: /etc/roles/webserver

# Include several files and the master.d directory
include:
  - extra_config
  - master.d/*
  - /etc/roles/webserver
```

3.1.16 Keepalive Settings

tcp_keepalive

Default: True

The tcp keepalive interval to set on TCP ports. This setting can be used to tune Salt connectivity issues in messy network environments with misbehaving firewalls.

```
tcp_keepalive: True
```

tcp_keepalive_cnt

Default: -1

Sets the ZeroMQ TCP keepalive count. May be used to tune issues with minion disconnects.

```
tcp_keepalive_cnt: -1
```

tcp_keepalive_idle

Default: 300

Sets ZeroMQ TCP keepalive idle. May be used to tune issues with minion disconnects.

```
tcp_keepalive_idle: 300
```

tcp_keepalive_intvl

Default: -1

Sets ZeroMQ TCP keepalive interval. May be used to tune issues with minion disconnects.

```
tcp_keepalive_intvl': -1
```

3.1.17 Windows Software Repo Settings

winrepo_provider

New in version 2015.8.0.

Specify the provider to be used for winrepo. Must be either `pygit2` or `gitpython`. If unset, then both will be tried in that same order, and the first one with a compatible version installed will be the provider that is used.

```
winrepo_provider: gitpython
```

winrepo_dir

Changed in version 2015.8.0: Renamed from `win_repo` to `winrepo_dir`.

Default: `/srv/salt/win/repo`

Location on the master where the *winrepo_remotes* are checked out for pre-2015.8.0 minions. 2015.8.0 and later minions use *winrepo_remotes_ng* instead.

```
winrepo_dir: /srv/salt/win/repo
```

winrepo_dir_ng

New in version 2015.8.0: A new *ng* repo was added.

Default: `/srv/salt/win/repo-ng`

Location on the master where the *winrepo_remotes_ng* are checked out for 2015.8.0 and later minions.

```
winrepo_dir_ng: /srv/salt/win/repo-ng
```

winrepo_cachefile

Changed in version 2015.8.0: Renamed from `win_repo_mastercachefile` to `winrepo_cachefile`

Note: 2015.8.0 and later minions do not use this setting since the cachefile is now located on the minion.

Default: `winrepo.p`

Path relative to *winrepo_dir* where the winrepo cache should be created.

```
winrepo_cachefile: winrepo.p
```

winrepo_remotes

Changed in version 2015.8.0: Renamed from `win_gitrepos` to `winrepo_remotes`.

Default: `['https://github.com/saltstack/salt-winrepo.git']`

List of git repositories to checkout and include in the winrepo for pre-2015.8.0 minions. 2015.8.0 and later minions use *winrepo_remotes_ng* instead.

```
winrepo_remotes:
  - https://github.com/saltstack/salt-winrepo.git
```

To specify a specific revision of the repository, prepend a commit ID to the URL of the repository:

```
winrepo_remotes:
  - '<commit_id> https://github.com/saltstack/salt-winrepo.git'
```

Replace `<commit_id>` with the SHA1 hash of a commit ID. Specifying a commit ID is useful in that it allows one to revert back to a previous version in the event that an error is introduced in the latest revision of the repo.

winrepo_remotes_ng

New in version 2015.8.0: A new *ng* repo was added.

Default: `['https://github.com/saltstack/salt-winrepo-ng.git']`

List of git repositories to checkout and include in the winrepo for 2015.8.0 and later minions.

```
winrepo_remotes_ng:
  - https://github.com/saltstack/salt-winrepo-ng.git
```

To specify a specific revision of the repository, prepend a commit ID to the URL of the repository:

```
winrepo_remotes_ng:
  - '<commit_id> https://github.com/saltstack/salt-winrepo-ng.git'
```

Replace `<commit_id>` with the SHA1 hash of a commit ID. Specifying a commit ID is useful in that it allows one to revert back to a previous version in the event that an error is introduced in the latest revision of the repo.

winrepo_branch

New in version 2015.8.0.

Default: `master`

If the branch is omitted from a winrepo remote, then this branch will be used instead. For example, in the configuration below, the first two remotes would use the `winrepo` branch/tag, while the third would use the `foo` branch/tag.

```
winrepo_branch: winrepo

winrepo_remotes:
  - https://mygitserver/winrepo1.git
  - https://mygitserver/winrepo2.git:
  - foo https://mygitserver/winrepo3.git
```

winrepo_ssl_verify

New in version 2015.8.0.

Changed in version 2016.11.0.

Default: `False`

Specifies whether or not to ignore SSL certificate errors when contacting the remote repository. The False setting is useful if you're using a git repo that uses a self-signed certificate. However, keep in mind that setting this to anything other True is a considered insecure, and using an SSH-based transport (if available) may be a better option.

In the 2016.11.0 release, the default config value changed from False to True.

```
winrepo_ssl_verify: True
```

Winrepo Authentication Options

These parameters only currently apply to the pygit2 *winrepo_provider*. Authentication works the same as it does in gitfs, as outlined in the *GitFS Walkthrough*, though the global configuration options are named differently to reflect that they are for winrepo instead of gitfs.

winrepo_user

New in version 2015.8.0.

Default: ''

Along with *winrepo_password*, is used to authenticate to HTTPS remotes.

```
winrepo_user: git
```

winrepo_password

New in version 2015.8.0.

Default: ''

Along with *winrepo_user*, is used to authenticate to HTTPS remotes. This parameter is not required if the repository does not use authentication.

```
winrepo_password: mypassword
```

winrepo_insecure_auth

New in version 2015.8.0.

Default: False

By default, Salt will not authenticate to an HTTP (non-HTTPS) remote. This parameter enables authentication over HTTP. **Enable this at your own risk.**

```
winrepo_insecure_auth: True
```

winrepo_pubkey

New in version 2015.8.0.

Default: ''

Along with *winrepo_privkey* (and optionally *winrepo_passphrase*), is used to authenticate to SSH remotes.

```
winrepo_pubkey: /path/to/key.pub
```

winrepo_privkey

New in version 2015.8.0.

Default: `''`

Along with *winrepo_pubkey* (and optionally *winrepo_passphrase*), is used to authenticate to SSH remotes.

```
winrepo_privkey: /path/to/key
```

winrepo_passphrase

New in version 2015.8.0.

Default: `''`

This parameter is optional, required only when the SSH key being used to authenticate is protected by a passphrase.

```
winrepo_passphrase: mypassphrase
```

winrepo_refspecs

New in version 2017.7.0.

Default: `['+refs/heads/*:refs/remotes/origin/*','+refs/tags/*:refs/tags/*']`

When fetching from remote repositories, by default Salt will fetch branches and tags. This parameter can be used to override the default and specify alternate refspecs to be fetched. This parameter works similarly to its *GitFS counterpart*, in that it can be configured both globally and for individual remotes.

```
winrepo_refspecs:
  - '+refs/heads/*:refs/remotes/origin/*'
  - '+refs/tags/*:refs/tags/*'
  - '+refs/pull/*/head:refs/remotes/origin/pr/*'
  - '+refs/pull/*/merge:refs/remotes/origin/merge/*'
```

3.1.18 Configure Master on Windows

The master on Windows requires no additional configuration. You can modify the master configuration by creating/editing the master config file located at `c:\salt\conf\master`. The same configuration options available on Linux are available in Windows, as long as they apply. For example, SSH options wouldn't apply in Windows. The main differences are the file paths. If you are familiar with common salt paths, the following table may be useful:

linux Paths		Windows Paths
`/etc/salt`	`<--->`	`c:\salt\conf`
`/`	`<--->`	`c:\salt`

So, for example, the master config file in Linux is `/etc/salt/master`. In Windows the master config file is `c:\salt\conf\master`. The Linux path `/etc/salt` becomes `c:\salt\conf` in Windows.

Common File Locations

Linux Paths	Windows Paths
conf_file: /etc/salt/master	conf_file: c:\salt\conf\master
log_file: /var/log/salt/master	log_file: c:\salt\var\log\salt\master
pidfile: /var/run/salt-master.pid	pidfile: c:\salt\var\run\salt-master.pid

Common Directories

Linux Paths	Windows Paths
cachedir: /var/cache/salt/master	cachedir: c:\salt\var\cache\salt\master
extension_modules: /var/cache/salt/master/extmods	c:\salt\var\cache\salt\master\extmods
pki_dir: /etc/salt/pki/master	pki_dir: c:\salt\conf\pki\master
root_dir: /	root_dir: c:\salt
sock_dir: /var/run/salt/master	sock_dir: c:\salt\var\run\salt\master

Roots

file_roots

Linux Paths	Windows Paths
/srv/salt	c:\salt\srv\salt
/srv/spm/salt	c:\salt\srv\spm\salt

pillar_roots

Linux Paths	Windows Paths
/srv/pillar	c:\salt\srv\pillar
/srv/spm/pillar	c:\salt\srv\spm\pillar

Win Repo Settings

Linux Paths	Windows Paths
winrepo_dir: /srv/salt/win/repo	winrepo_dir: c:\salt\srv\salt\win\repo
winrepo_dir_ng: /srv/salt/win/repo-ng	winrepo_dir_ng: c:\salt\srv\salt\win\repo-ng

3.2 Configuring the Salt Minion

The Salt system is amazingly simple and easy to configure. The two components of the Salt system each have a respective configuration file. The **salt-master** is configured via the master configuration file, and the **salt-minion** is configured via the minion configuration file.

See also:

example minion configuration file

The Salt Minion configuration is very simple. Typically, the only value that needs to be set is the master value so the minion knows where to locate its master.

By default, the salt-minion configuration will be in /etc/salt/minion. A notable exception is FreeBSD, where the configuration will be in /usr/local/etc/salt/minion.

3.2.1 Minion Primary Configuration

master

Default: salt

The hostname or IP address of the master. See *ipv6* for IPv6 connections to the master.

Default: salt

```
master: salt
```

master:port Syntax

New in version 2015.8.0.

The master config option can also be set to use the master's IP in conjunction with a port number by default.

```
master: localhost:1234
```

For IPv6 formatting with a port, remember to add brackets around the IP address before adding the port and enclose the line in single quotes to make it a string:

```
master: '[2001:db8:85a3:8d3:1319:8a2e:370:7348]:1234'
```

Note: If a port is specified in the master as well as *master_port*, the master_port setting will be overridden by the master configuration.

List of Masters Syntax

The option can also be set to a list of masters, enabling *multi-master* mode.

```
master:
  - address1
  - address2
```

Changed in version 2014.7.0: The master can be dynamically configured. The *master* value can be set to an module function which will be executed and will assume that the returning value is the ip or hostname of the desired master. If a function is being specified, then the *master_type* option must be set to func, to tell the minion that the value is a function to be run and not a fully-qualified domain name.

```
master: module.function
master_type: func
```

In addition, instead of using multi-master mode, the minion can be configured to use the list of master addresses as a failover list, trying the first address, then the second, etc. until the minion successfully connects. To enable this behavior, set *master_type* to failover:

```
master:
  - address1
  - address2
master_type: failover
```

ipv6

Default: None

Whether the master should be connected over IPv6. By default salt minion will try to automatically detect IPv6 connectivity to master.

```
ipv6: True
```

master_uri_format

New in version 2015.8.0.

Specify the format in which the master address will be evaluated. Valid options are `default` or `ip_only`. If `ip_only` is specified, then the master address will not be split into IP and PORT, so be sure that only an IP (or domain name) is set in the *master* configuration setting.

```
master_uri_format: ip_only
```

master_tops_first

New in version 2018.3.0.

Default: False

SLS targets defined using the *Master Tops* system are normally executed *after* any matches defined in the *Top File*. Set this option to `True` to have the minion execute the *Master Tops* states first.

```
master_tops_first: True
```

master_type

New in version 2014.7.0.

Default: str

The type of the *master* variable. Can be `str`, `failover`, `func` or `disable`.

```
master_type: failover
```

If this option is set to `failover`, *master* must be a list of master addresses. The minion will then try each master in the order specified in the list until it successfully connects. *master_alive_interval* must also be set, this determines how often the minion will verify the presence of the master.

```
master_type: func
```

If the master needs to be dynamically assigned by executing a function instead of reading in the static master value, set this to `func`. This can be used to manage the minion's master setting from an execution module. By simply changing the algorithm in the module to return a new master ip/fqdn, restart the minion and it will connect to the new master.

As of version 2016.11.0 this option can be set to `disable` and the minion will never attempt to talk to the master. This is useful for running a masterless minion daemon.

```
master_type: disable
```

max_event_size

New in version 2014.7.0.

Default: `1048576`

Passing very large events can cause the minion to consume large amounts of memory. This value tunes the maximum size of a message allowed onto the minion event bus. The value is expressed in bytes.

```
max_event_size: 1048576
```

master_failback

New in version 2016.3.0.

Default: `False`

If the minion is in multi-master mode and the :conf_minion`master_type` configuration option is set to `failover`, this setting can be set to `True` to force the minion to fail back to the first master in the list if the first master is back online.

```
master_failback: False
```

master_failback_interval

New in version 2016.3.0.

Default: `0`

If the minion is in multi-master mode, the :conf_minion`master_type` configuration is set to `failover`, and the `master_failback` option is enabled, the master failback interval can be set to ping the top master with this interval, in seconds.

```
master_failback_interval: 0
```

master_alive_interval

Default: `0`

Configures how often, in seconds, the minion will verify that the current master is alive and responding. The minion will try to establish a connection to the next master in the list if it finds the existing one is dead.

```
master_alive_interval: 30
```

master_shuffle

New in version 2014.7.0.

Default: `False`

If *master* is a list of addresses and :conf_minion`master_type` is `failover`, shuffle them before trying to connect to distribute the minions over all available masters. This uses Python's `random.shuffle` method.

```
master_shuffle: True
```

random_master

Default: `False`

If *master* is a list of addresses, and :conf_minion`master_type` is set to `failover` shuffle them before trying to connect to distribute the minions over all available masters. This uses Python's `random.shuffle` method.

```
random_master: True
```

retry_dns

Default: `30`

Set the number of seconds to wait before attempting to resolve the master hostname if name resolution fails. Defaults to 30 seconds. Set to zero if the minion should shutdown and not retry.

```
retry_dns: 30
```

retry_dns_count

New in version 2018.3.4.

Default: `None`

Set the number of attempts to perform when resolving the master hostname if name resolution fails. By default the minion will retry indefinitely.

```
retry_dns_count: 3
```

master_port

Default: `4506`

The port of the master ret server, this needs to coincide with the ret_port option on the Salt master.

```
master_port: 4506
```

publish_port

Default: `4505`

The port of the master publish server, this needs to coincide with the publish_port option on the Salt master.

```
publish_port: 4505
```

source_interface_name

New in version 2018.3.0.

The name of the interface to use when establishing the connection to the Master.

Note: If multiple IP addresses are configured on the named interface, the first one will be selected. In that case, for a better selection, consider using the *source_address* option.

Note: To use an IPv6 address from the named interface, make sure the option *ipv6* is enabled, i.e., `ipv6: true`.

Note: If the interface is down, it will avoid using it, and the Minion will bind to `0.0.0.0` (all interfaces).

Warning: This option requires modern version of the underlying libraries used by the selected transport:

- `zeromq` requires `pyzmq` >= 16.0.1 and `libzmq` >= 4.1.6
- `tcp` requires `tornado` >= 4.5

Configuration example:

```
source_interface_name: bond0.1234
```

source_address

New in version 2018.3.0.

The source IP address or the domain name to be used when connecting the Minion to the Master. See *ipv6* for IPv6 connections to the Master.

Warning: This option requires modern version of the underlying libraries used by the selected transport:

- `zeromq` requires `pyzmq` >= 16.0.1 and `libzmq` >= 4.1.6
- `tcp` requires `tornado` >= 4.5

Configuration example:

```
source_address: if-bond0-1234.sjc.us-west.internal
```

source_ret_port

New in version 2018.3.0.

The source port to be used when connecting the Minion to the Master ret server.

Warning: This option requires modern version of the underlying libraries used by the selected transport:

- `zeromq` requires `pyzmq` >= 16.0.1 and `libzmq` >= 4.1.6
- `tcp` requires `tornado` >= 4.5

Configuration example:

```
source_ret_port: 49017
```

`source_publish_port`

New in version 2018.3.0.

The source port to be used when connecting the Minion to the Master publish server.

Warning: This option requires modern version of the underlying libraries used by the selected transport:

- `zeromq` requires `pyzmq` >= 16.0.1 and `libzmq` >= 4.1.6
- `tcp` requires `tornado` >= 4.5

Configuration example:

```
source_publish_port: 49018
```

`user`

Default: `root`

The user to run the Salt processes

```
user: root
```

`sudo_user`

Default: `''`

The user to run salt remote execution commands as via sudo. If this option is enabled then sudo will be used to change the active user executing the remote command. If enabled the user will need to be allowed access via the sudoers file for the user that the salt minion is configured to run as. The most common option would be to use the root user. If this option is set the `user` option should also be set to a non-root user. If migrating from a root minion to a non root minion the minion cache should be cleared and the minion pki directory will need to be changed to the ownership of the new user.

```
sudo_user: root
```

`pidfile`

Default: `/var/run/salt-minion.pid`

The location of the daemon's process ID file

```
pidfile: /var/run/salt-minion.pid
```

root_dir

Default: /

This directory is prepended to the following options: *pki_dir*, *cachedir*, *log_file*, *sock_dir*, and pidfile.

```
root_dir: /
```

conf_file

Default: /etc/salt/minion

The path to the minion's configuration file.

```
conf_file: /etc/salt/minion
```

pki_dir

Default: /etc/salt/pki/minion

The directory used to store the minion's public and private keys.

```
pki_dir: /etc/salt/pki/minion
```

id

Default: the system's hostname

See also:

Salt Walkthrough

The **Setting up a Salt Minion** section contains detailed information on how the hostname is determined.

Explicitly declare the id for this minion to use. Since Salt uses detached ids it is possible to run multiple minions on the same machine but with different ids.

```
id: foo.bar.com
```

minion_id_caching

New in version 0.17.2.

Default: True

Caches the minion id to a file when the minion's *id* is not statically defined in the minion config. This setting prevents potential problems when automatic minion id resolution changes, which can cause the minion to lose connection with the master. To turn off minion id caching, set this config to False.

For more information, please see Issue #7558 and Pull Request #8488.

```
minion_id_caching: True
```

append_domain

Default: None

Append a domain to a hostname in the event that it does not exist. This is useful for systems where socket.getfqdn() does not actually result in a FQDN (for instance, Solaris).

```
append_domain: foo.org
```

minion_id_lowercase

Default: False

Convert minion id to lowercase when it is being generated. Helpful when some hosts get the minion id in uppercase. Cached ids will remain the same and not converted.

```
minion_id_lowercase: True
```

cachedir

Default: /var/cache/salt/minion

The location for minion cache data.

This directory may contain sensitive data and should be protected accordingly.

```
cachedir: /var/cache/salt/minion
```

color_theme

Default: ""

Specifies a path to the color theme to use for colored command line output.

```
color_theme: /etc/salt/color_theme
```

append_minionid_config_dirs

Default: [] (the empty list) for regular minions, ['cachedir'] for proxy minions.

Append minion_id to these configuration directories. Helps with multiple proxies and minions running on the same machine. Allowed elements in the list: pki_dir, cachedir, extension_modules. Normally not needed unless running several proxies and/or minions on the same machine.

```
append_minionid_config_dirs:
  - pki_dir
  - cachedir
```

verify_env

Default: True

Verify and set permissions on configuration directories at startup.

```
verify_env: True
```

Note: When set to True the verify_env option requires WRITE access to the configuration directory (/etc/salt/). In certain situations such as mounting /etc/salt/ as read-only for templating this will create a stack trace when state.apply is called.

cache_jobs

Default: False

The minion can locally cache the return data from jobs sent to it, this can be a good way to keep track of the minion side of the jobs the minion has executed. By default this feature is disabled, to enable set cache_jobs to True.

```
cache_jobs: False
```

grains

Default: (empty)

See also:

Grains in the Minion Config

Statically assigns grains to the minion.

```
grains:
  roles:
    - webserver
    - memcache
  deployment: datacenter4
  cabinet: 13
  cab_u: 14-15
```

grains_cache

Default: False

The minion can locally cache grain data instead of refreshing the data each time the grain is referenced. By default this feature is disabled, to enable set grains_cache to True.

```
grains_cache: False
```

grains_deep_merge

New in version 2016.3.0.

Default: False

The grains can be merged, instead of overridden, using this option. This allows custom grains to defined different subvalues of a dictionary grain. By default this feature is disabled, to enable set grains_deep_merge to `True`.

```
grains_deep_merge: False
```

For example, with these custom grains functions:

```
def custom1_k1():
    return {'custom1': {'k1': 'v1'}}

def custom1_k2():
    return {'custom1': {'k2': 'v2'}}
```

Without `grains_deep_merge`, the result would be:

```
custom1:
  k1: v1
```

With `grains_deep_merge`, the result will be:

```
custom1:
  k1: v1
  k2: v2
```

grains_refresh_every

Default: `0`

The `grains_refresh_every` setting allows for a minion to periodically check its grains to see if they have changed and, if so, to inform the master of the new grains. This operation is moderately expensive, therefore care should be taken not to set this value too low.

Note: This value is expressed in minutes.

A value of 10 minutes is a reasonable default.

```
grains_refresh_every: 0
```

fibre_channel_grains

Default: `False`

The `fibre_channel_grains` setting will enable the `fc_wwn` grain for Fibre Channel WWN's on the minion. Since this grain is expensive, it is disabled by default.

```
fibre_channel_grains: True
```

iscsi_grains

Default: `False`

The `iscsi_grains` setting will enable the `iscsi_iqn` grain on the minion. Since this grain is expensive, it is disabled by default.

```
iscsi_grains: True
```

mine_enabled

New in version 2015.8.10.

Default: `True`

Determines whether or not the salt minion should run scheduled mine updates. If this is set to False then the mine update function will not get added to the scheduler for the minion.

```
mine_enabled: True
```

mine_return_job

New in version 2015.8.10.

Default: `False`

Determines whether or not scheduled mine updates should be accompanied by a job return for the job cache.

```
mine_return_job: False
```

mine_functions

Default: Empty

Designate which functions should be executed at mine_interval intervals on each minion. *See this documentation on the Salt Mine* for more information. Note these can be defined in the pillar for a minion as well.

example minion configuration file

```
mine_functions:
  test.ping: []
  network.ip_addrs:
    interface: eth0
    cidr: '10.0.0.0/8'
```

mine_interval

Default: `60`

The number of minutes between mine updates.

```
mine_interval: 60
```

sock_dir

Default: `/var/run/salt/minion`

The directory where Unix sockets will be kept.

```
sock_dir: /var/run/salt/minion
```

enable_gpu_grains

Default: `True`

Enable GPU hardware data for your master. Be aware that the minion can take a while to start up when lspci and/or dmidecode is used to populate the grains for the minion, so this can be set to `False` if you do not need these grains.

```
enable_gpu_grains: False
```

outputter_dirs

Default: `[]`

A list of additional directories to search for salt outputters in.

```
outputter_dirs: []
```

backup_mode

Default: `''`

Make backups of files replaced by `file.managed` and `file.recurse` state modules under *cachedir* in `file_backup` subdirectory preserving original paths. Refer to *File State Backups documentation* for more details.

```
backup_mode: minion
```

acceptance_wait_time

Default: `10`

The number of seconds to wait until attempting to re-authenticate with the master.

```
acceptance_wait_time: 10
```

acceptance_wait_time_max

Default: `0`

The maximum number of seconds to wait until attempting to re-authenticate with the master. If set, the wait will increase by *acceptance_wait_time* seconds each iteration.

```
acceptance_wait_time_max: 0
```

rejected_retry

Default: `False`

If the master rejects the minion's public key, retry instead of exiting. Rejected keys will be handled the same as waiting on acceptance.

```
rejected_retry: False
```

random_reauth_delay

Default: `10`

When the master key changes, the minion will try to re-auth itself to receive the new master key. In larger environments this can cause a syn-flood on the master because all minions try to re-auth immediately. To prevent this and have a minion wait for a random amount of time, use this optional parameter. The wait-time will be a random number of seconds between 0 and the defined value.

```
random_reauth_delay: 60
```

master_tries

New in version 2016.3.0.

Default: `1`

The number of attempts to connect to a master before giving up. Set this to `-1` for unlimited attempts. This allows for a master to have downtime and the minion to reconnect to it later when it comes back up. In `failover' mode, which is set in the *master_type* configuration, this value is the number of attempts for each set of masters. In this mode, it will cycle through the list of masters for each attempt.

`master_tries` is different than *auth_tries* because `auth_tries` attempts to retry auth attempts with a single master. `auth_tries` is under the assumption that you can connect to the master but not gain authorization from it. `master_tries` will still cycle through all of the masters in a given try, so it is appropriate if you expect occasional downtime from the master(s).

```
master_tries: 1
```

auth_tries

New in version 2014.7.0.

Default: `7`

The number of attempts to authenticate to a master before giving up. Or, more technically, the number of consecutive SaltReqTimeoutErrors that are acceptable when trying to authenticate to the master.

```
auth_tries: 7
```

auth_timeout

New in version 2014.7.0.

Default: `60`

When waiting for a master to accept the minion's public key, salt will continuously attempt to reconnect until successful. This is the timeout value, in seconds, for each individual attempt. After this timeout expires, the minion will wait for *acceptance_wait_time* seconds before trying again. Unless your master is under unusually heavy load, this should be left at the default.

```
auth_timeout: 60
```

auth_safemode

New in version 2014.7.0.

Default: `False`

If authentication fails due to SaltReqTimeoutError during a ping_interval, this setting, when set to `True`, will cause a sub-minion process to restart.

```
auth_safemode: False
```

ping_interval

Default: `0`

Instructs the minion to ping its master(s) every n number of minutes. Used primarily as a mitigation technique against minion disconnects.

```
ping_interval: 0
```

random_startup_delay

Default: `0`

The maximum bound for an interval in which a minion will randomly sleep upon starting up prior to attempting to connect to a master. This can be used to splay connection attempts for cases where many minions starting up at once may place undue load on a master.

For example, setting this to `5` will tell a minion to sleep for a value between `0` and `5` seconds.

```
random_startup_delay: 5
```

recon_default

Default: `1000`

The interval in milliseconds that the socket should wait before trying to reconnect to the master (1000ms = 1 second).

```
recon_default: 1000
```

recon_max

Default: `10000`

The maximum time a socket should wait. Each interval the time to wait is calculated by doubling the previous time. If recon_max is reached, it starts again at the recon_default.

Short example:

- reconnect 1: the socket will wait `recon_default' milliseconds
- reconnect 2: `recon_default' * 2

- reconnect 3: (`recon_default' * 2) * 2
- reconnect 4: value from previous interval * 2
- reconnect 5: value from previous interval * 2
- reconnect x: if value >= recon_max, it starts again with recon_default

```
recon_max: 10000
```

recon_randomize

Default: True

Generate a random wait time on minion start. The wait time will be a random value between recon_default and recon_default + recon_max. Having all minions reconnect with the same recon_default and recon_max value kind of defeats the purpose of being able to change these settings. If all minions have the same values and the setup is quite large (several thousand minions), they will still flood the master. The desired behavior is to have time-frame within all minions try to reconnect.

```
recon_randomize: True
```

loop_interval

Default: 1

The loop_interval sets how long in seconds the minion will wait between evaluating the scheduler and running cleanup tasks. This defaults to 1 second on the minion scheduler.

```
loop_interval: 1
```

pub_ret

Default: True

Some installations choose to start all job returns in a cache or a returner and forgo sending the results back to a master. In this workflow, jobs are most often executed with --async from the Salt CLI and then results are evaluated by examining job caches on the minions or any configured returners. WARNING: Setting this to False will **disable** returns back to the master.

```
pub_ret: True
```

return_retry_timer

Default: 5

The default timeout for a minion return attempt.

```
return_retry_timer: 5
```

`return_retry_timer_max`

Default: `10`

The maximum timeout for a minion return attempt. If non-zero the minion return retry timeout will be a random int between `return_retry_timer` and `return_retry_timer_max`

```
return_retry_timer_max: 10
```

`cache_sreqs`

Default: `True`

The connection to the master ret_port is kept open. When set to False, the minion creates a new connection for every return to the master.

```
cache_sreqs: True
```

`ipc_mode`

Default: `ipc`

Windows platforms lack POSIX IPC and must rely on slower TCP based inter- process communications. Set ipc_mode to `tcp` on such systems.

```
ipc_mode: ipc
```

`tcp_pub_port`

Default: `4510`

Publish port used when *`ipc_mode`* is set to `tcp`.

```
tcp_pub_port: 4510
```

`tcp_pull_port`

Default: `4511`

Pull port used when *`ipc_mode`* is set to `tcp`.

```
tcp_pull_port: 4511
```

`transport`

Default: `zeromq`

Changes the underlying transport layer. ZeroMQ is the recommended transport while additional transport layers are under development. Supported values are `zeromq`, `raet` (experimental), and `tcp` (experimental). This setting has a significant impact on performance and should not be changed unless you know what you are doing!

```
transport: zeromq
```

syndic_finger

Default: `''`

The key fingerprint of the higher-level master for the syndic to verify it is talking to the intended master.

```
syndic_finger: 'ab:30:65:2a:d6:9e:20:4f:d8:b2:f3:a7:d4:65:50:10'
```

proxy_host

Default: `''`

The hostname used for HTTP proxy access.

```
proxy_host: proxy.my-domain
```

proxy_port

Default: `0`

The port number used for HTTP proxy access.

```
proxy_port: 31337
```

proxy_username

Default: `''`

The username used for HTTP proxy access.

```
proxy_username: charon
```

proxy_password

Default: `''`

The password used for HTTP proxy access.

```
proxy_password: obolus
```

3.2.2 Docker Configuration

docker.update_mine

New in version 2017.7.8,2018.3.3.

Changed in version Fluorine: The default value is now `False`

Default: `True`

If enabled, when containers are added, removed, stopped, started, etc., the *mine* will be updated with the results of *docker.ps verbose=True all=True host=True*. This mine data is used by *mine.get_docker*. Set this option to `False` to keep Salt from updating the mine with this information.

Note: This option can also be set in Grains or Pillar data, with Grains overriding Pillar and the minion config file overriding Grains.

Note: Disabling this will of course keep *mine.get_docker* from returning any information for a given minion.

```
docker.update_mine: False
```

docker.compare_container_networks

New in version 2018.3.0.

Default: `{'static': ['Aliases','Links','IPAMConfig'],'automatic': ['IPAddress','Gateway','GlobalIPv6Address','IPv6Gateway']}`

Specifies which keys are examined by *docker.compare_container_networks*.

Note: This should not need to be modified unless new features added to Docker result in new keys added to the network configuration which must be compared to determine if two containers have different network configs. This config option exists solely as a way to allow users to continue using Salt to manage their containers after an API change, without waiting for a new Salt release to catch up to the changes in the Docker API.

```
docker.compare_container_networks:
  static:
    - Aliases
    - Links
    - IPAMConfig
  automatic:
    - IPAddress
    - Gateway
    - GlobalIPv6Address
    - IPv6Gateway
```

optimization_order

Default: `[0,1,2]`

In cases where Salt is distributed without .py files, this option determines the priority of optimization level(s) Salt's module loader should prefer.

Note: This option is only supported on Python 3.5+.

```
optimization_order:
  - 2
  - 0
  - 1
```

3.2.3 Minion Execution Module Management

disable_modules

Default: [] (all execution modules are enabled by default)

The event may occur in which the administrator desires that a minion should not be able to execute a certain module.

However, the sys module is built into the minion and cannot be disabled.

This setting can also tune the minion. Because all modules are loaded into system memory, disabling modules will lower the minion's memory footprint.

Modules should be specified according to their file name on the system and not by their virtual name. For example, to disable cmd, use the string cmdmod which corresponds to salt.modules.cmdmod.

```
disable_modules:
  - test
  - solr
```

disable_returners

Default: [] (all returners are enabled by default)

If certain returners should be disabled, this is the place

```
disable_returners:
  - mongo_return
```

whitelist_modules

Default: [] (Module whitelisting is disabled. Adding anything to the config option will cause only the listed modules to be enabled. Modules not in the list will not be loaded.)

This option is the reverse of disable_modules. If enabled, only execution modules in this list will be loaded and executed on the minion.

Note that this is a very large hammer and it can be quite difficult to keep the minion working the way you think it should since Salt uses many modules internally itself. At a bare minimum you need the following enabled or else the minion won't start.

```
whitelist_modules:
  - cmdmod
  - test
  - config
```

module_dirs

Default: []

A list of extra directories to search for Salt modules

```
module_dirs:
  - /var/lib/salt/modules
```

returner_dirs

Default: []

A list of extra directories to search for Salt returners

```
returner_dirs:
  - /var/lib/salt/returners
```

states_dirs

Default: []

A list of extra directories to search for Salt states

```
states_dirs:
  - /var/lib/salt/states
```

grains_dirs

Default: []

A list of extra directories to search for Salt grains

```
grains_dirs:
  - /var/lib/salt/grains
```

render_dirs

Default: []

A list of extra directories to search for Salt renderers

```
render_dirs:
  - /var/lib/salt/renderers
```

utils_dirs

Default: []

A list of extra directories to search for Salt utilities

```
utils_dirs:
  - /var/lib/salt/utils
```

cython_enable

Default: False

Set this value to true to enable auto-loading and compiling of .pyx modules, This setting requires that gcc and cython are installed on the minion.

```
cython_enable: False
```

`enable_zip_modules`

New in version 2015.8.0.

Default: `False`

Set this value to true to enable loading of zip archives as extension modules. This allows for packing module code with specific dependencies to avoid conflicts and/or having to install specific modules' dependencies in system libraries.

```
enable_zip_modules: False
```

`providers`

Default: (empty)

A module provider can be statically overwritten or extended for the minion via the `providers` option. This can be done *on an individual basis in an SLS file*, or globally here in the minion config, like below.

```
providers:
  service: systemd
```

`modules_max_memory`

Default: `-1`

Specify a max size (in bytes) for modules on import. This feature is currently only supported on *NIX operating systems and requires psutil.

```
modules_max_memory: -1
```

`extmod_whitelist/extmod_blacklist`

New in version 2017.7.0.

By using this dictionary, the modules that are synced to the minion's extmod cache using *saltutil.sync_** can be limited. If nothing is set to a specific type, then all modules are accepted. To block all modules of a specific type, whitelist an empty list.

```
extmod_whitelist:
  modules:
    - custom_module
  engines:
    - custom_engine
  pillars: []

extmod_blacklist:
  modules:
    - specific_module
```

Valid options:

- beacons

- clouds
- sdb
- modules
- states
- grains
- renderers
- returners
- proxy
- engines
- output
- utils
- pillar

3.2.4 Top File Settings

These parameters only have an effect if running a masterless minion.

state_top

Default: `top.sls`

The state system uses a ``top'' file to tell the minions what environment to use and what modules to use. The state_top file is defined relative to the root of the base environment.

```
state_top: top.sls
```

state_top_saltenv

This option has no default value. Set it to an environment name to ensure that *only* the top file from that environment is considered during a *highstate*.

Note: Using this value does not change the merging strategy. For instance, if *top_file_merging_strategy* is set to `merge`, and *state_top_saltenv* is set to `foo`, then any sections for environments other than `foo` in the top file for the `foo` environment will be ignored. With *state_top_saltenv* set to `base`, all states from all environments in the `base` top file will be applied, while all other top files are ignored. The only way to set *state_top_saltenv* to something other than `base` and not have the other environments in the targeted top file ignored, would be to set *top_file_merging_strategy* to `merge_all`.

```
state_top_saltenv: dev
```

top_file_merging_strategy

Changed in version 2016.11.0: A merge_all strategy has been added.

Default: merge

When no specific fileserver environment (a.k.a. saltenv) has been specified for a *highstate*, all environments' top files are inspected. This config option determines how the SLS targets in those top files are handled.

When set to merge, the base environment's top file is evaluated first, followed by the other environments' top files. The first target expression (e.g. '*') for a given environment is kept, and when the same target expression is used in a different top file evaluated later, it is ignored. Because base is evaluated first, it is authoritative. For example, if there is a target for '*' for the foo environment in both the base and foo environment's top files, the one in the foo environment would be ignored. The environments will be evaluated in no specific order (aside from base coming first). For greater control over the order in which the environments are evaluated, use *env_order*. Note that, aside from the base environment's top file, any sections in top files that do not match that top file's environment will be ignored. So, for example, a section for the qa environment would be ignored if it appears in the dev environment's top file. To keep use cases like this from being ignored, use the merge_all strategy.

When set to same, then for each environment, only that environment's top file is processed, with the others being ignored. For example, only the dev environment's top file will be processed for the dev environment, and any SLS targets defined for dev in the base environment's (or any other environment's) top file will be ignored. If an environment does not have a top file, then the top file from the *default_top* config parameter will be used as a fallback.

When set to merge_all, then all states in all environments in all top files will be applied. The order in which individual SLS files will be executed will depend on the order in which the top files were evaluated, and the environments will be evaluated in no specific order. For greater control over the order in which the environments are evaluated, use *env_order*.

```
top_file_merging_strategy: same
```

env_order

Default: []

When *top_file_merging_strategy* is set to merge, and no environment is specified for a *highstate*, this config option allows for the order in which top files are evaluated to be explicitly defined.

```
env_order:
  - base
  - dev
  - qa
```

default_top

Default: base

When *top_file_merging_strategy* is set to same, and no environment is specified for a *highstate* (i.e. *environment* is not set for the minion), this config option specifies a fallback environment in which to look for a top file if an environment lacks one.

```
default_top: dev
```

startup_states

Default: ''

States to run when the minion daemon starts. To enable, set startup_states to:

- highstate: Execute state.highstate
- sls: Read in the sls_list option and execute the named sls files
- top: Read top_file option and execute based on that file on the Master

```
startup_states: ''
```

sls_list

Default: []

List of states to run when the minion starts up if startup_states is set to sls.

```
sls_list:
  - edit.vim
  - hyper
```

top_file

Default: ''

Top file to execute if startup_states is set to top.

```
top_file: ''
```

3.2.5 State Management Settings

renderer

Default: yaml_jinja

The default renderer used for local state executions

```
renderer: yaml_jinja
```

test

Default: False

Set all state calls to only test if they are going to actually make changes or just post what changes are going to be made.

```
test: False
```

state_verbose

Default: True

Controls the verbosity of state runs. By default, the results of all states are returned, but setting this value to False will cause salt to only display output for states that failed or states that have changes.

```
state_verbose: True
```

state_output

Default: full

The state_output setting controls which results will be output full multi line:

- full, terse - each state will be full/terse
- mixed - only states with errors will be full
- changes - states with changes and errors will be full

full_id, mixed_id, changes_id and terse_id are also allowed; when set, the state ID will be used as name in the output.

```
state_output: full
```

state_output_diff

Default: False

The state_output_diff setting changes whether or not the output from successful states is returned. Useful when even the terse output of these states is cluttering the logs. Set it to True to ignore them.

```
state_output_diff: False
```

autoload_dynamic_modules

Default: True

autoload_dynamic_modules turns on automatic loading of modules found in the environments on the master. This is turned on by default. To turn off auto-loading modules when states run, set this value to False.

```
autoload_dynamic_modules: True
```

Default: True

clean_dynamic_modules keeps the dynamic modules on the minion in sync with the dynamic modules on the master. This means that if a dynamic module is not on the master it will be deleted from the minion. By default this is enabled and can be disabled by changing this value to False.

```
clean_dynamic_modules: True
```

Note: If extmod_whitelist is specified, modules which are not whitelisted will also be cleaned here.

saltenv

Changed in version 2018.3.0: Renamed from `environment` to `saltenv`. If `environment` is used, `saltenv` will take its value. If both are used, `environment` will be ignored and `saltenv` will be used.

Normally the minion is not isolated to any single environment on the master when running states, but the environment can be isolated on the minion side by statically setting it. Remember that the recommended way to manage environments is to isolate via the top file.

```
saltenv: dev
```

lock_saltenv

New in version 2018.3.0.

Default: `False`

For purposes of running states, this option prevents using the `saltenv` argument to manually set the environment. This is useful to keep a minion which has the *saltenv* option set to `dev` from running states from an environment other than `dev`.

```
lock_saltenv: True
```

snapper_states

Default: False

The *snapper_states* value is used to enable taking snapper snapshots before and after salt state runs. This allows for state runs to be rolled back.

For snapper states to function properly snapper needs to be installed and enabled.

```
snapper_states: True
```

snapper_states_config

Default: `root`

Snapper can execute based on a snapper configuration. The configuration needs to be set up before snapper can use it. The default configuration is `root`, this default makes snapper run on SUSE systems using the default configuration set up at install time.

```
snapper_states_config: root
```

3.2.6 File Directory Settings

file_client

Default: `remote`

The client defaults to looking on the master server for files, but can be directed to look on the minion by setting this parameter to `local`.

```
file_client: remote
```

use_master_when_local

Default: `False`

When using a local *file_client*, this parameter is used to allow the client to connect to a master for remote execution.

```
use_master_when_local: False
```

file_roots

Default:

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

When using a local *file_client*, this parameter is used to setup the fileserver's environments. This parameter operates identically to the *master config parameter* of the same name.

```
file_roots:
  base:
    - /srv/salt
  dev:
    - /srv/salt/dev/services
    - /srv/salt/dev/states
  prod:
    - /srv/salt/prod/services
    - /srv/salt/prod/states
```

fileserver_followsymlinks

New in version 2014.1.0.

Default: `True`

By default, the file_server follows symlinks when walking the filesystem tree. Currently this only applies to the default roots fileserver_backend.

```
fileserver_followsymlinks: True
```

fileserver_ignoresymlinks

New in version 2014.1.0.

Default: `False`

If you do not want symlinks to be treated as the files they are pointing to, set `fileserver_ignoresymlinks` to `True`. By default this is set to False. When set to `True`, any detected symlink while listing files on the Master will not be returned to the Minion.

```
fileserver_ignoresymlinks: False
```

fileserver_limit_traversal

New in version 2014.1.0.

Default: `False`

By default, the Salt fileserver recurses fully into all defined environments to attempt to find files. To limit this behavior so that the fileserver only traverses directories with SLS files and special Salt directories like _modules, set `fileserver_limit_traversal` to `True`. This might be useful for installations where a file root has a very large number of files and performance is impacted.

```
fileserver_limit_traversal: False
```

hash_type

Default: `sha256`

The hash_type is the hash to use when discovering the hash of a file on the local fileserver. The default is sha256, but md5, sha1, sha224, sha384, and sha512 are also supported.

```
hash_type: sha256
```

3.2.7 Pillar Configuration

pillar_roots

Default:

```
base:
  - /srv/pillar
```

When using a local *file_client*, this parameter is used to setup the pillar environments.

```
pillar_roots:
  base:
    - /srv/pillar
  dev:
    - /srv/pillar/dev
  prod:
    - /srv/pillar/prod
```

on_demand_ext_pillar

New in version 2016.3.6,2016.11.3,2017.7.0.

Default: `['libvirt','virtkey']`

When using a local *file_client*, this option controls which external pillars are permitted to be used on-demand using *pillar.ext*.

```
on_demand_ext_pillar:
  - libvirt
  - virtkey
  - git
```

Warning: This will allow a masterless minion to request specific pillar data via `pillar.ext`, and may be considered a security risk. However, pillar data generated in this way will not affect the *in-memory pillar data*, so this risk is limited to instances in which states/modules/etc. (built-in or custom) rely upon pillar data generated by `pillar.ext`.

decrypt_pillar

New in version 2017.7.0.

Default: `[]`

A list of paths to be recursively decrypted during pillar compilation.

```
decrypt_pillar:
  - 'foo:bar': gpg
  - 'lorem:ipsum:dolor'
```

Entries in this list can be formatted either as a simple string, or as a key/value pair, with the key being the pillar location, and the value being the renderer to use for pillar decryption. If the former is used, the renderer specified by `decrypt_pillar_default` will be used.

decrypt_pillar_delimiter

New in version 2017.7.0.

Default: `:`

The delimiter used to distinguish nested data structures in the `decrypt_pillar` option.

```
decrypt_pillar_delimiter: '|'
decrypt_pillar:
  - 'foo|bar': gpg
  - 'lorem|ipsum|dolor'
```

decrypt_pillar_default

New in version 2017.7.0.

Default: `gpg`

The default renderer used for decryption, if one is not specified for a given pillar key in `decrypt_pillar`.

```
decrypt_pillar_default: my_custom_renderer
```

decrypt_pillar_renderers

New in version 2017.7.0.

Default: ['gpg']

List of renderers which are permitted to be used for pillar decryption.

```
decrypt_pillar_renderers:
  - gpg
  - my_custom_renderer
```

pillarenv

Default: None

Isolates the pillar environment on the minion side. This functions the same as the environment setting, but for pillar instead of states.

```
pillarenv: dev
```

pillarenv_from_saltenv

New in version 2017.7.0.

Default: False

When set to True, the *pillarenv* value will assume the value of the effective saltenv when running states. This essentially makes `salt '*' state.sls mysls saltenv=dev` equivalent to `salt '*' state.sls mysls saltenv=dev pillarenv=dev`. If *pillarenv* is set, either in the minion config file or via the CLI, it will override this option.

```
pillarenv_from_saltenv: True
```

pillar_raise_on_missing

New in version 2015.5.0.

Default: False

Set this option to True to force a KeyError to be raised whenever an attempt to retrieve a named value from pillar fails. When this option is set to False, the failed attempt returns an empty string.

minion_pillar_cache

New in version 2016.3.0.

Default: False

The minion can locally cache rendered pillar data under *cachedir*/pillar. This allows a temporarily disconnected minion to access previously cached pillar data by invoking salt-call with the --local and --pillar_root=:conf_minion:*cachedir*/pillar options. Before enabling this setting consider that the rendered pillar may contain security sensitive data. Appropriate access restrictions should be in place. By default the saved pillar data will be readable only by the user account running salt. By default this feature is disabled, to enable set minion_pillar_cache to True.

```
minion_pillar_cache: False
```

file_recv_max_size

New in version 2014.7.0.

Default: `100`

Set a hard-limit on the size of the files that can be pushed to the master. It will be interpreted as megabytes.

```
file_recv_max_size: 100
```

pass_to_ext_pillars

Specify a list of configuration keys whose values are to be passed to external pillar functions.

Suboptions can be specified using the `:' notation (i.e. `option:suboption`)

The values are merged and included in the `extra_minion_data` optional parameter of the external pillar function. The `extra_minion_data` parameter is passed only to the external pillar functions that have it explicitly specified in their definition.

If the config contains

```
opt1: value1
opt2:
  subopt1: value2
  subopt2: value3

pass_to_ext_pillars:
  - opt1
  - opt2: subopt1
```

the `extra_minion_data` parameter will be

```
{'opt1': 'value1',
 'opt2': {'subopt1': 'value2'}}
```

3.2.8 Security Settings

open_mode

Default: `False`

Open mode can be used to clean out the PKI key received from the Salt master, turn on open mode, restart the minion, then turn off open mode and restart the minion to clean the keys.

```
open_mode: False
```

master_finger

Default: `''`

Fingerprint of the master public key to validate the identity of your Salt master before the initial key exchange. The master fingerprint can be found by running ``salt-key -F master'' on the Salt master.

```
master_finger: 'ba:30:65:2a:d6:9e:20:4f:d8:b2:f3:a7:d4:65:11:13'
```

keysize

Default: 2048

The size of key that should be generated when creating new keys.

```
keysize: 2048
```

permissive_pki_access

Default: False

Enable permissive access to the salt keys. This allows you to run the master or minion as root, but have a non-root group be given access to your pki_dir. To make the access explicit, root must belong to the group you've given access to. This is potentially quite insecure.

```
permissive_pki_access: False
```

verify_master_pubkey_sign

Default: False

Enables verification of the master-public-signature returned by the master in auth-replies. Please see the tutorial on how to configure this properly Multimaster-PKI with Failover Tutorial

New in version 2014.7.0.

```
verify_master_pubkey_sign: True
```

If this is set to True, *master_sign_pubkey* must be also set to True in the master configuration file.

master_sign_key_name

Default: master_sign

The filename without the *.pub* suffix of the public key that should be used for verifying the signature from the master. The file must be located in the minion's pki directory.

New in version 2014.7.0.

```
master_sign_key_name: <filename_without_suffix>
```

autosign_grains

New in version 2018.3.0.

Default: not defined

The grains that should be sent to the master on authentication to decide if the minion's key should be accepted automatically.

Please see the *Autoaccept Minions from Grains* documentation for more information.

```
autosign_grains:
  - uuid
  - server_id
```

always_verify_signature

Default: `False`

If `verify_master_pubkey_sign` is enabled, the signature is only verified if the public-key of the master changes. If the signature should always be verified, this can be set to `True`.

New in version 2014.7.0.

```
always_verify_signature: True
```

cmd_blacklist_glob

Default: `[]`

If `cmd_blacklist_glob` is enabled then any shell command called over remote execution or via salt-call will be checked against the glob matches found in the *cmd_blacklist_glob* list and any matched shell command will be blocked.

Note: This blacklist is only applied to direct executions made by the *salt* and *salt-call* commands. This does NOT blacklist commands called from states or shell commands executed from other modules.

New in version 2016.11.0.

```
cmd_blacklist_glob:
  - 'rm * '
  - 'cat /etc/* '
```

cmd_whitelist_glob

Default: `[]`

If `cmd_whitelist_glob` is enabled then any shell command called over remote execution or via salt-call will be checked against the glob matches found in the *cmd_whitelist_glob* list and any shell command NOT found in the list will be blocked. If *cmd_whitelist_glob* is NOT SET, then all shell commands are permitted.

Note: This whitelist is only applied to direct executions made by the *salt* and *salt-call* commands. This does NOT restrict commands called from states or shell commands executed from other modules.

New in version 2016.11.0.

```
cmd_whitelist_glob:
  - 'ls * '
  - 'cat /etc/fstab'
```

ssl

New in version 2016.11.0.

Default: `None`

TLS/SSL connection options. This could be set to a dictionary containing arguments corresponding to python `ssl.wrap_socket` method. For details see Tornado and Python documentation.

Note: to set enum arguments values like `cert_reqs` and `ssl_version` use constant names without ssl module prefix: `CERT_REQUIRED` or `PROTOCOL_SSLv23`.

```
ssl:
    keyfile: <path_to_keyfile>
    certfile: <path_to_certfile>
    ssl_version: PROTOCOL_TLSv1_2
```

3.2.9 Reactor Settings

reactor

Default: `[]`

Defines a salt reactor. See the *Reactor* documentation for more information.

```
reactor: []
```

reactor_refresh_interval

Default: `60`

The TTL for the cache of the reactor configuration.

```
reactor_refresh_interval: 60
```

reactor_worker_threads

Default: `10`

The number of workers for the runner/wheel in the reactor.

```
reactor_worker_threads: 10
```

reactor_worker_hwm

Default: `10000`

The queue size for workers in the reactor.

```
reactor_worker_hwm: 10000
```

3.2.10 Thread Settings

multiprocessing

Default: `True`

If `multiprocessing` is enabled when a minion receives a publication a new process is spawned and the command is executed therein. Conversely, if `multiprocessing` is disabled the new publication will be run executed in a thread.

```
multiprocessing: True
```

process_count_max

New in version 2018.3.0.

Default: `-1`

Limit the maximum amount of processes or threads created by `salt-minion`. This is useful to avoid resource exhaustion in case the minion receives more publications than it is able to handle, as it limits the number of spawned processes or threads. `-1` is the default and disables the limit.

```
process_count_max: -1
```

3.2.11 Minion Logging Settings

log_file

Default: `/var/log/salt/minion`

The minion log can be sent to a regular file, local path name, or network location. See also *log_file*.

Examples:

```
log_file: /var/log/salt/minion
```

```
log_file: file:///dev/log
```

```
log_file: udp://loghost:10514
```

log_level

Default: `warning`

The level of messages to send to the console. See also *log_level*.

```
log_level: warning
```

log_level_logfile

Default: info

The level of messages to send to the log file. See also *log_level_logfile*. When it is not set explicitly it will inherit the level set by *log_level* option.

```
log_level_logfile: warning
```

log_datefmt

Default: %H:%M:%S

The date and time format used in console log messages. See also *log_datefmt*.

```
log_datefmt: '%H:%M:%S'
```

log_datefmt_logfile

Default: %Y-%m-%d %H:%M:%S

The date and time format used in log file messages. See also *log_datefmt_logfile*.

```
log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'
```

log_fmt_console

Default: [%(levelname)-8s] %(message)s

The format of the console logging messages. See also *log_fmt_console*.

Note: Log colors are enabled in log_fmt_console rather than the color config since the logging system is loaded before the minion config.

Console log colors are specified by these additional formatters:

%(colorlevel)s %(colorname)s %(colorprocess)s %(colormsg)s

Since it is desirable to include the surrounding brackets, `[' and `]', in the coloring of the messages, these color formatters also include padding as well. Color LogRecord attributes are only available for console logging.

```
log_fmt_console: '%(colorlevel)s %(colormsg)s'
log_fmt_console: '[%(levelname)-8s] %(message)s'
```

log_fmt_logfile

Default: %(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s

The format of the log file logging messages. See also *log_fmt_logfile*.

```
log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'
```

log_granular_levels

Default: {}

This can be used to control logging levels more specifically. See also log_granular_levels.

zmq_monitor

Default: False

To diagnose issues with minions disconnecting or missing returns, ZeroMQ supports the use of monitor sockets to log connection events. This feature requires ZeroMQ 4.0 or higher.

To enable ZeroMQ monitor sockets, set `zmq_monitor' to `True' and log at a debug level or higher.

A sample log event is as follows:

```
[DEBUG   ] ZeroMQ event: {'endpoint': 'tcp://127.0.0.1:4505', 'event': 512,
'value': 27, 'description': 'EVENT_DISCONNECTED'}
```

All events logged will include the string ZeroMQ event. A connection event should be logged as the minion starts up and initially connects to the master. If not, check for debug log level and that the necessary version of ZeroMQ is installed.

tcp_authentication_retries

Default: 5

The number of times to retry authenticating with the salt master when it comes back online.

Zeromq does a lot to make sure when connections come back online that they reauthenticate. The tcp transport should try to connect with a new connection if the old one times out on reauthenticating.

-1 for infinite tries.

failhard

Default: False

Set the global failhard flag. This informs all states to stop running states at the moment a single state fails

```
failhard: False
```

3.2.12 Include Configuration

Configuration can be loaded from multiple files. The order in which this is done is:

1. The minion config file itself
2. The files matching the glob in default_include
3. The files matching the glob in include (if defined)

Each successive step overrides any values defined in the previous steps. Therefore, any config options defined in one of the default_include files would override the same value in the minion config file, and any options defined in include would override both.

default_include

Default: minion.d/*.conf

The minion can include configuration from other files. Per default the minion will automatically include all config files from *minion.d/*.conf* where minion.d is relative to the directory of the minion configuration file.

Note: Salt creates files in the minion.d directory for its own use. These files are prefixed with an underscore. A common example of this is the _schedule.conf file.

include

Default: not defined

The minion can include configuration from other files. To enable this, pass a list of paths to this option. The paths can be either relative or absolute; if relative, they are considered to be relative to the directory the main minion configuration file lives in. Paths can make use of shell-style globbing. If no files are matched by a path passed to this option then the minion will log a warning message.

```
# Include files from a minion.d directory in the same
# directory as the minion config file
include: minion.d/*.conf

# Include a single extra file into the configuration
include: /etc/roles/webserver

# Include several files and the minion.d directory
include:
  - extra_config
  - minion.d/*
  - /etc/roles/webserver
```

3.2.13 Keepalive Settings

tcp_keepalive

Default: True

The tcp keepalive interval to set on TCP ports. This setting can be used to tune Salt connectivity issues in messy network environments with misbehaving firewalls.

```
tcp_keepalive: True
```

tcp_keepalive_cnt

Default: -1

Sets the ZeroMQ TCP keepalive count. May be used to tune issues with minion disconnects.

```
tcp_keepalive_cnt: -1
```

tcp_keepalive_idle

Default: `300`

Sets ZeroMQ TCP keepalive idle. May be used to tune issues with minion disconnects.

```
tcp_keepalive_idle: 300
```

tcp_keepalive_intvl

Default: `-1`

Sets ZeroMQ TCP keepalive interval. May be used to tune issues with minion disconnects.

```
tcp_keepalive_intvl': -1
```

3.2.14 Frozen Build Update Settings

These options control how *salt.modules.saltutil.update()* works with esky frozen apps. For more information look at https://github.com/cloudmatrix/esky/.

update_url

Default: `False` (Update feature is disabled)

The url to use when looking for application updates. Esky depends on directory listings to search for new versions. A webserver running on your Master is a good starting point for most setups.

```
update_url: 'http://salt.example.com/minion-updates'
```

update_restart_services

Default: `[]` (service restarting on update is disabled)

A list of services to restart when the minion software is updated. This would typically just be a list containing the minion's service name, but you may have other services that need to go with it.

```
update_restart_services: ['salt-minion']
```

winrepo_cache_expire_min

New in version 2016.11.0.

Default: `0`

If set to a nonzero integer, then passing `refresh=True` to functions in the *windows pkg module* will not refresh the windows repo metadata if the age of the metadata is less than this value. The exception to this is *pkg.refresh_db*, which will always refresh the metadata, regardless of age.

```
winrepo_cache_expire_min: 1800
```

winrepo_cache_expire_max

New in version 2016.11.0.

Default: 21600

If the windows repo metadata is older than this value, and the metadata is needed by a function in the *windows pkg module*, the metadata will be refreshed.

```
winrepo_cache_expire_max: 86400
```

3.2.15 Minion Windows Software Repo Settings

Important: To use these config options, the minion can be running in master-minion or masterless mode.

winrepo_source_dir

Default: salt://win/repo-ng/

The source location for the winrepo sls files.

```
winrepo_source_dir: salt://win/repo-ng/
```

3.2.16 Standalone Minion Windows Software Repo Settings

Important: To use these config options, the minion must be running in masterless mode (set *file_client* to local).

winrepo_dir

Changed in version 2015.8.0: Renamed from win_repo to winrepo_dir. Also, this option did not have a default value until this version.

Default: C:\salt\srv\salt\win\repo

Location on the minion where the *winrepo_remotes* are checked out.

```
winrepo_dir: 'D:\winrepo'
```

winrepo_dir_ng

New in version 2015.8.0: A new *ng* repo was added.

Default: /srv/salt/win/repo-ng

Location on the minion where the *winrepo_remotes_ng* are checked out for 2015.8.0 and later minions.

```
winrepo_dir_ng: /srv/salt/win/repo-ng
```

winrepo_cachefile

Changed in version 2015.8.0: Renamed from win_repo_cachefile to winrepo_cachefile. Also, this option did not have a default value until this version.

Default: winrepo.p

Path relative to *winrepo_dir* where the winrepo cache should be created.

```
winrepo_cachefile: winrepo.p
```

winrepo_remotes

Changed in version 2015.8.0: Renamed from win_gitrepos to winrepo_remotes. Also, this option did not have a default value until this version.

New in version 2015.8.0.

Default: ['https://github.com/saltstack/salt-winrepo.git']

List of git repositories to checkout and include in the winrepo

```
winrepo_remotes:
  - https://github.com/saltstack/salt-winrepo.git
```

To specify a specific revision of the repository, prepend a commit ID to the URL of the repository:

```
winrepo_remotes:
  - '<commit_id> https://github.com/saltstack/salt-winrepo.git'
```

Replace <commit_id> with the SHA1 hash of a commit ID. Specifying a commit ID is useful in that it allows one to revert back to a previous version in the event that an error is introduced in the latest revision of the repo.

winrepo_remotes_ng

New in version 2015.8.0: A new *ng* repo was added.

Default: ['https://github.com/saltstack/salt-winrepo-ng.git']

List of git repositories to checkout and include in the winrepo for 2015.8.0 and later minions.

```
winrepo_remotes_ng:
  - https://github.com/saltstack/salt-winrepo-ng.git
```

To specify a specific revision of the repository, prepend a commit ID to the URL of the repository:

```
winrepo_remotes_ng:
  - '<commit_id> https://github.com/saltstack/salt-winrepo-ng.git'
```

Replace <commit_id> with the SHA1 hash of a commit ID. Specifying a commit ID is useful in that it allows one to revert back to a previous version in the event that an error is introduced in the latest revision of the repo.

ssh_merge_pillar

New in version 2018.3.2.

Default: True

Merges the compiled pillar data with the pillar data already available globally. This is useful when using `salt-ssh` or `salt-call --local` and overriding the pillar data in a state file:

```
apply_showpillar:
  module.run:
    - name: state.apply
    - mods:
      - showpillar
    - kwargs:
        pillar:
          test: "foo bar"
```

If set to `True` the `showpillar` state will have access to the global pillar data.

If set to `False` only the overriding pillar data will be available to the `showpillar` state.

3.3 Configuring the Salt Proxy Minion

The Salt system is amazingly simple and easy to configure. The two components of the Salt system each have a respective configuration file. The **`salt-master`** is configured via the master configuration file, and the **`salt-proxy`** is configured via the proxy configuration file.

See also:

example proxy minion configuration file

The Salt Minion configuration is very simple. Typically, the only value that needs to be set is the master value so the proxy knows where to locate its master.

By default, the salt-proxy configuration will be in `/etc/salt/proxy`. A notable exception is FreeBSD, where the configuration will be in `/usr/local/etc/salt/proxy`.

3.3.1 Proxy-specific Configuration Options

add_proxymodule_to_opts

New in version 2015.8.2.

Changed in version 2016.3.0.

Default: `False`

Add the proxymodule LazyLoader object to opts.

```
add_proxymodule_to_opts: True
```

proxy_merge_grains_in_module

New in version 2016.3.0.

Changed in version 2017.7.0.

Default: `True`

If a proxymodule has a function called `grains`, then call it during regular grains loading and merge the results with the proxy's grains dictionary. Otherwise it is assumed that the module calls the grains function in a custom way and returns the data elsewhere.

```
proxy_merge_grains_in_module: False
```

proxy_keep_alive

New in version 2017.7.0.

Default: `True`

Whether the connection with the remote device should be restarted when dead. The proxy module must implement the `alive` function, otherwise the connection is considered alive.

```
proxy_keep_alive: False
```

proxy_keep_alive_interval

New in version 2017.7.0.

Default: `1`

The frequency of keepalive checks, in minutes. It requires the *proxy_keep_alive* option to be enabled (and the proxy module to implement the `alive` function).

```
proxy_keep_alive_interval: 5
```

proxy_always_alive

New in version 2017.7.0.

Default: `True`

Whether the proxy should maintain the connection with the remote device. Similarly to *proxy_keep_alive*, this option is very specific to the design of the proxy module. When *proxy_always_alive* is set to `False`, the connection with the remote device is not maintained and has to be closed after every command.

```
proxy_always_alive: False
```

proxy_merge_pillar_in_opts

New in version 2017.7.3.

Default: `False`.

Whether the pillar data to be merged into the proxy configuration options. As multiple proxies can run on the same server, we may need different configuration options for each, while there's one single configuration file. The solution is merging the pillar data of each proxy minion into the opts.

```
proxy_merge_pillar_in_opts: True
```

proxy_deep_merge_pillar_in_opts

New in version 2017.7.3.

Default: `False`.

Deep merge of pillar data into configuration opts. This option is evaluated only when `proxy_merge_pillar_in_opts` is enabled.

proxy_merge_pillar_in_opts_strategy

New in version 2017.7.3.

Default: `smart`.

The strategy used when merging pillar configuration into opts. This option is evaluated only when `proxy_merge_pillar_in_opts` is enabled.

proxy_mines_pillar

New in version 2017.7.3.

Default: `True`.

Allow enabling mine details using pillar data. This evaluates the mine configuration under the pillar, for the following regular minion options that are also equally available on the proxy minion: *mine_interval*, and `mine_functions`.

3.4 Configuration file examples

- *Example master configuration file*
- *Example minion configuration file*
- *Example proxy minion configuration file*

3.4.1 Example master configuration file

```
##### Primary configuration settings #####
##########################################
# This configuration file is used to manage the behavior of the Salt Master.
# Values that are commented out but have an empty line after the comment are
# defaults that do not need to be set in the config. If there is no blank line
# after the comment then the value is presented as an example and is not the
# default.

# Per default, the master will automatically include all config files
# from master.d/*.conf (master.d is a directory in the same directory
# as the main master config file).
#default_include: master.d/*.conf

# The address of the interface to bind to:
```

```
#interface: 0.0.0.0

# Whether the master should listen for IPv6 connections. If this is set to True,
# the interface option must be adjusted, too. (For example: "interface: '::'")
#ipv6: False

# The tcp port used by the publisher:
#publish_port: 4505

# The user under which the salt master will run. Salt will update all
# permissions to allow the specified user to run the master. The exception is
# the job cache, which must be deleted if this user is changed. If the
# modified files cause conflicts, set verify_env to False.
#user: root

# The port used by the communication interface. The ret (return) port is the
# interface used for the file server, authentication, job returns, etc.
#ret_port: 4506

# Specify the location of the daemon process ID file:
#pidfile: /var/run/salt-master.pid

# The root directory prepended to these options: pki_dir, cachedir,
# sock_dir, log_file, autosign_file, autoreject_file, extension_modules,
# key_logfile, pidfile, autosign_grains_dir:
#root_dir: /

# The path to the master's configuration file.
#conf_file: /etc/salt/master

# Directory used to store public key data:
#pki_dir: /etc/salt/pki/master

# Key cache. Increases master speed for large numbers of accepted
# keys. Available options: 'sched'. (Updates on a fixed schedule.)
# Note that enabling this feature means that minions will not be
# available to target for up to the length of the maintanence loop
# which by default is 60s.
#key_cache: ''

# Directory to store job and cache data:
# This directory may contain sensitive data and should be protected accordingly.
#
#cachedir: /var/cache/salt/master

# Directory for custom modules. This directory can contain subdirectories for
# each of Salt's module types such as "runners", "output", "wheel", "modules",
# "states", "returners", "engines", "utils", etc.
#extension_modules: /var/cache/salt/master/extmods

# Directory for custom modules. This directory can contain subdirectories for
# each of Salt's module types such as "runners", "output", "wheel", "modules",
# "states", "returners", "engines", "utils", etc.
# Like 'extension_modules' but can take an array of paths
#module_dirs: []

# Verify and set permissions on configuration directories at startup:
#verify_env: True
```

```
# Set the number of hours to keep old job information in the job cache:
#keep_jobs: 24

# The number of seconds to wait when the client is requesting information
# about running jobs.
#gather_job_timeout: 10

# Set the default timeout for the salt command and api. The default is 5
# seconds.
#timeout: 5

# The loop_interval option controls the seconds for the master's maintenance
# process check cycle. This process updates file server backends, cleans the
# job cache and executes the scheduler.
#loop_interval: 60

# Set the default outputter used by the salt command. The default is "nested".
#output: nested

# To set a list of additional directories to search for salt outputters, set the
# outputter_dirs option.
#outputter_dirs: []

# Set the default output file used by the salt command. Default is to output
# to the CLI and not to a file. Functions the same way as the "--out-file"
# CLI option, only sets this to a single file for all salt commands.
#output_file: None

# Return minions that timeout when running commands like test.ping
#show_timeout: True

# Tell the client to display the jid when a job is published.
#show_jid: False

# By default, output is colored. To disable colored output, set the color value
# to False.
#color: True

# Do not strip off the colored output from nested results and state outputs
# (true by default).
# strip_colors: False

# To display a summary of the number of minions targeted, the number of
# minions returned, and the number of minions that did not return, set the
# cli_summary value to True. (False by default.)
#
#cli_summary: False

# Set the directory used to hold unix sockets:
#sock_dir: /var/run/salt/master

# The master can take a while to start up when lspci and/or dmidecode is used
# to populate the grains for the master. Enable if you want to see GPU hardware
# data for your master.
# enable_gpu_grains: False

# The master maintains a job cache. While this is a great addition, it can be
```

```
# a burden on the master for larger deployments (over 5000 minions).
# Disabling the job cache will make previously executed jobs unavailable to
# the jobs system and is not generally recommended.
#job_cache: True

# Cache minion grains, pillar and mine data via the cache subsystem in the
# cachedir or a database.
#minion_data_cache: True

# Cache subsystem module to use for minion data cache.
#cache: localfs
# Enables a fast in-memory cache booster and sets the expiration time.
#memcache_expire_seconds: 0
# Set a memcache limit in items (bank + key) per cache storage (driver + driver_opts).
#memcache_max_items: 1024
# Each time a cache storage got full cleanup all the expired items not just the oldest
↪one.
#memcache_full_cleanup: False
# Enable collecting the memcache stats and log it on `debug` log level.
#memcache_debug: False

# Store all returns in the given returner.
# Setting this option requires that any returner-specific configuration also
# be set. See various returners in salt/returners for details on required
# configuration values. (See also, event_return_queue below.)
#
#event_return: mysql

# On busy systems, enabling event_returns can cause a considerable load on
# the storage system for returners. Events can be queued on the master and
# stored in a batched fashion using a single transaction for multiple events.
# By default, events are not queued.
#event_return_queue: 0

# Only return events matching tags in a whitelist, supports glob matches.
#event_return_whitelist:
#   - salt/master/a_tag
#   - salt/run/*/ret

# Store all event returns **except** the tags in a blacklist, supports globs.
#event_return_blacklist:
#   - salt/master/not_this_tag
#   - salt/wheel/*/ret

# Passing very large events can cause the minion to consume large amounts of
# memory. This value tunes the maximum size of a message allowed onto the
# master event bus. The value is expressed in bytes.
#max_event_size: 1048576

# By default, the master AES key rotates every 24 hours. The next command
# following a key rotation will trigger a key refresh from the minion which may
# result in minions which do not respond to the first command after a key refresh.
#
# To tell the master to ping all minions immediately after an AES key refresh, set
# ping_on_rotate to True. This should mitigate the issue where a minion does not
# appear to initially respond after a key is rotated.
#
# Note that ping_on_rotate may cause high load on the master immediately after
```

```
# the key rotation event as minions reconnect. Consider this carefully if this
# salt master is managing a large number of minions.
#
# If disabled, it is recommended to handle this event by listening for the
# 'aes_key_rotate' event with the 'key' tag and acting appropriately.
# ping_on_rotate: False

# By default, the master deletes its cache of minion data when the key for that
# minion is removed. To preserve the cache after key deletion, set
# 'preserve_minion_cache' to True.
#
# WARNING: This may have security implications if compromised minions auth with
# a previous deleted minion ID.
#preserve_minion_cache: False

# Allow or deny minions from requesting their own key revocation
#allow_minion_key_revoke: True

# If max_minions is used in large installations, the master might experience
# high-load situations because of having to check the number of connected
# minions for every authentication. This cache provides the minion-ids of
# all connected minions to all MWorker-processes and greatly improves the
# performance of max_minions.
# con_cache: False

# The master can include configuration from other files. To enable this,
# pass a list of paths to this option. The paths can be either relative or
# absolute; if relative, they are considered to be relative to the directory
# the main master configuration file lives in (this file). Paths can make use
# of shell-style globbing. If no files are matched by a path passed to this
# option, then the master will log a warning message.
#
# Include a config file from some other path:
# include: /etc/salt/extra_config
#
# Include config from several files and directories:
# include:
#   - /etc/salt/extra_config


#####  Large-scale tuning settings   #####
##########################################
# Max open files
#
# Each minion connecting to the master uses AT LEAST one file descriptor, the
# master subscription connection. If enough minions connect you might start
# seeing on the console (and then salt-master crashes):
#   Too many open files (tcp_listener.cpp:335)
#   Aborted (core dumped)
#
# By default this value will be the one of `ulimit -Hn`, ie, the hard limit for
# max open files.
#
# If you wish to set a different value than the default one, uncomment and
# configure this setting. Remember that this value CANNOT be higher than the
# hard limit. Raising the hard limit depends on your OS and/or distribution,
# a good way to find the limit is to search the internet. For example:
#   raise max open files hard limit debian
```

```
#
#max_open_files: 100000

# The number of worker threads to start. These threads are used to manage
# return calls made from minions to the master. If the master seems to be
# running slowly, increase the number of threads. This setting can not be
# set lower than 3.
#worker_threads: 5

# Set the ZeroMQ high water marks
# http://api.zeromq.org/3-2:zmq-setsockopt

# The listen queue size / backlog
#zmq_backlog: 1000

# The publisher interface ZeroMQPubServerChannel
#pub_hwm: 1000

# The master may allocate memory per-event and not
# reclaim it.
# To set a high-water mark for memory allocation, use
# ipc_write_buffer to set a high-water mark for message
# buffering.
# Value: In bytes. Set to 'dynamic' to have Salt select
# a value for you. Default is disabled.
# ipc_write_buffer: 'dynamic'

# These two batch settings, batch_safe_limit and batch_safe_size, are used to
# automatically switch to a batch mode execution. If a command would have been
# sent to more than <batch_safe_limit> minions, then run the command in
# batches of <batch_safe_size>. If no batch_safe_size is specified, a default
# of 8 will be used. If no batch_safe_limit is specified, then no automatic
# batching will occur.
#batch_safe_limit: 100
#batch_safe_size: 8

# Master stats enables stats events to be fired from the master at close
# to the defined interval
#master_stats: False
#master_stats_event_iter: 60


#####        Security settings       #####
##########################################
# Enable passphrase protection of Master private key.  Although a string value
# is acceptable; passwords should be stored in an external vaulting mechanism
# and retrieved via sdb. See https://docs.saltstack.com/en/latest/topics/sdb/.
# Passphrase protection is off by default but an example of an sdb profile and
# query is as follows.
# masterkeyring:
#  driver: keyring
#  service: system
#
# key_pass: sdb://masterkeyring/key_pass

# Enable passphrase protection of the Master signing_key. This only applies if
# master_sign_pubkey is set to True.  This is disabled by default.
# master_sign_pubkey: True
```

```
# signing_key_pass: sdb://masterkeyring/signing_pass

# Enable "open mode", this mode still maintains encryption, but turns off
# authentication, this is only intended for highly secure environments or for
# the situation where your keys end up in a bad state. If you run in open mode
# you do so at your own risk!
#open_mode: False

# Enable auto_accept, this setting will automatically accept all incoming
# public keys from the minions. Note that this is insecure.
#auto_accept: False

# The size of key that should be generated when creating new keys.
#keysize: 2048

# Time in minutes that an incoming public key with a matching name found in
# pki_dir/minion_autosign/keyid is automatically accepted. Expired autosign keys
# are removed when the master checks the minion_autosign directory.
# 0 equals no timeout
# autosign_timeout: 120

# If the autosign_file is specified, incoming keys specified in the
# autosign_file will be automatically accepted. This is insecure.  Regular
# expressions as well as globing lines are supported. The file must be readonly
# except for the owner. Use permissive_pki_access to allow the group write access.
#autosign_file: /etc/salt/autosign.conf

# Works like autosign_file, but instead allows you to specify minion IDs for
# which keys will automatically be rejected. Will override both membership in
# the autosign_file and the auto_accept setting.
#autoreject_file: /etc/salt/autoreject.conf

# If the autosign_grains_dir is specified, incoming keys from minons with grain
# values matching those defined in files in this directory will be accepted
# automatically. This is insecure. Minions need to be configured to send the grains.
#autosign_grains_dir: /etc/salt/autosign_grains

# Enable permissive access to the salt keys. This allows you to run the
# master or minion as root, but have a non-root group be given access to
# your pki_dir. To make the access explicit, root must belong to the group
# you've given access to. This is potentially quite insecure. If an autosign_file
# is specified, enabling permissive_pki_access will allow group access to that
# specific file.
#permissive_pki_access: False

# Allow users on the master access to execute specific commands on minions.
# This setting should be treated with care since it opens up execution
# capabilities to non root users. By default this capability is completely
# disabled.
#publisher_acl:
#  larry:
#    - test.ping
#    - network.*
#
# Blacklist any of the following users or modules
#
# This example would blacklist all non sudo users, including root from
# running any commands. It would also blacklist any use of the "cmd"
```

```
# module. This is completely disabled by default.
#
#
# Check the list of configured users in client ACL against users on the
# system and throw errors if they do not exist.
#client_acl_verify: True
#
#publisher_acl_blacklist:
#  users:
#    - root
#    - '^(?!sudo_).*$'   #  all non sudo users
#  modules:
#    - cmd

# Enforce publisher_acl & publisher_acl_blacklist when users have sudo
# access to the salt command.
#
#sudo_acl: False

# The external auth system uses the Salt auth modules to authenticate and
# validate users to access areas of the Salt system.
#external_auth:
#  pam:
#    fred:
#      - test.*
#
# Time (in seconds) for a newly generated token to live. Default: 12 hours
#token_expire: 43200
#
# Allow eauth users to specify the expiry time of the tokens they generate.
# A boolean applies to all users or a dictionary of whitelisted eauth backends
# and usernames may be given.
# token_expire_user_override:
#   pam:
#     - fred
#     - tom
#   ldap:
#     - gary
#
#token_expire_user_override: False

# Set to True to enable keeping the calculated user's auth list in the token
# file. This is disabled by default and the auth list is calculated or requested
# from the eauth driver each time.
#keep_acl_in_token: False

# Auth subsystem module to use to get authorized access list for a user. By default it's
# the same module used for external authentication.
#eauth_acl_module: django

# Allow minions to push files to the master. This is disabled by default, for
# security purposes.
#file_recv: False

# Set a hard-limit on the size of the files that can be pushed to the master.
# It will be interpreted as megabytes. Default: 100
#file_recv_max_size: 100
```

```
# Signature verification on messages published from the master.
# This causes the master to cryptographically sign all messages published to its event
# bus, and minions then verify that signature before acting on the message.
#
# This is False by default.
#
# Note that to facilitate interoperability with masters and minions that are different
# versions, if sign_pub_messages is True but a message is received by a minion with
# no signature, it will still be accepted, and a warning message will be logged.
# Conversely, if sign_pub_messages is False, but a minion receives a signed
# message it will be accepted, the signature will not be checked, and a warning message
# will be logged. This behavior went away in Salt 2014.1.0 and these two situations
# will cause minion to throw an exception and drop the message.
# sign_pub_messages: False

# Signature verification on messages published from minions
# This requires that minions cryptographically sign the messages they
# publish to the master.  If minions are not signing, then log this information
# at loglevel 'INFO' and drop the message without acting on it.
# require_minion_sign_messages: False

# The below will drop messages when their signatures do not validate.
# Note that when this option is False but `require_minion_sign_messages` is True
# minions MUST sign their messages but the validity of their signatures
# is ignored.
# These two config options exist so a Salt infrastructure can be moved
# to signing minion messages gradually.
# drop_messages_signature_fail: False

# Use TLS/SSL encrypted connection between master and minion.
# Can be set to a dictionary containing keyword arguments corresponding to Python's
# 'ssl.wrap_socket' method.
# Default is None.
#ssl:
#    keyfile: <path_to_keyfile>
#    certfile: <path_to_certfile>
#    ssl_version: PROTOCOL_TLSv1_2

#####     Salt-SSH Configuration     #####
##########################################
# Define the default salt-ssh roster module to use
#roster: flat

# Pass in an alternative location for the salt-ssh `flat` roster file
#roster_file: /etc/salt/roster

# Define locations for `flat` roster files so they can be chosen when using Salt API.
# An administrator can place roster files into these locations. Then when
# calling Salt API, parameter 'roster_file' should contain a relative path to
# these locations. That is, "roster_file=/foo/roster" will be resolved as
# "/etc/salt/roster.d/foo/roster" etc. This feature prevents passing insecure
# custom rosters through the Salt API.
#
#rosters:
# - /etc/salt/roster.d
# - /opt/salt/some/more/rosters

# The ssh password to log in with.
```

```
#ssh_passwd: ''

#The target system's ssh port number.
#ssh_port: 22

# Comma-separated list of ports to scan.
#ssh_scan_ports: 22

# Scanning socket timeout for salt-ssh.
#ssh_scan_timeout: 0.01

# Boolean to run command via sudo.
#ssh_sudo: False

# Number of seconds to wait for a response when establishing an SSH connection.
#ssh_timeout: 60

# The user to log in as.
#ssh_user: root

# The log file of the salt-ssh command:
#ssh_log_file: /var/log/salt/ssh

# Pass in minion option overrides that will be inserted into the SHIM for
# salt-ssh calls. The local minion config is not used for salt-ssh. Can be
# overridden on a per-minion basis in the roster (`minion_opts`)
#ssh_minion_opts:
#  gpg_keydir: /root/gpg

# Set this to True to default to using ~/.ssh/id_rsa for salt-ssh
# authentication with minions
#ssh_use_home_key: False

# Set this to True to default salt-ssh to run with ``-o IdentitiesOnly=yes``.
# This option is intended for situations where the ssh-agent offers many
# different identities and allows ssh to ignore those identities and use the
# only one specified in options.
#ssh_identities_only: False

# List-only nodegroups for salt-ssh. Each group must be formed as either a
# comma-separated list, or a YAML list. This option is useful to group minions
# into easy-to-target groups when using salt-ssh. These groups can then be
# targeted with the normal -N argument to salt-ssh.
#ssh_list_nodegroups: {}

# salt-ssh has the ability to update the flat roster file if a minion is not
# found in the roster.  Set this to True to enable it.
#ssh_update_roster: False

#####    Master Module Management    #####
##########################################
# Manage how master side modules are loaded.

# Add any additional locations to look for master runners:
#runner_dirs: []

# Add any additional locations to look for master utils:
#utils_dirs: []
```

```
# Enable Cython for master side modules:
#cython_enable: False


#####      State System settings     #####
##########################################
# The state system uses a "top" file to tell the minions what environment to
# use and what modules to use. The state_top file is defined relative to the
# root of the base environment as defined in "File Server settings" below.
#state_top: top.sls

# The master_tops option replaces the external_nodes option by creating
# a plugable system for the generation of external top data. The external_nodes
# option is deprecated by the master_tops option.
#
# To gain the capabilities of the classic external_nodes system, use the
# following configuration:
# master_tops:
#   ext_nodes: <Shell command which returns yaml>
#
#master_tops: {}

# The renderer to use on the minions to render the state data
#renderer: yaml_jinja

# Default Jinja environment options for all templates except sls templates
#jinja_env:
#  block_start_string: '{%'
#  block_end_string: '%}'
#  variable_start_string: '{{'
#  variable_end_string: '}}'
#  comment_start_string: '{#'
#  comment_end_string: '#}'
#  line_statement_prefix:
#  line_comment_prefix:
#  trim_blocks: False
#  lstrip_blocks: False
#  newline_sequence: '\n'
#  keep_trailing_newline: False

# Jinja environment options for sls templates
#jinja_sls_env:
#  block_start_string: '{%'
#  block_end_string: '%}'
#  variable_start_string: '{{'
#  variable_end_string: '}}'
#  comment_start_string: '{#'
#  comment_end_string: '#}'
#  line_statement_prefix:
#  line_comment_prefix:
#  trim_blocks: False
#  lstrip_blocks: False
#  newline_sequence: '\n'
#  keep_trailing_newline: False

# The failhard option tells the minions to stop immediately after the first
# failure detected in the state execution, defaults to False
```

```
#failhard: False

# The state_verbose and state_output settings can be used to change the way
# state system data is printed to the display. By default all data is printed.
# The state_verbose setting can be set to True or False, when set to False
# all data that has a result of True and no changes will be suppressed.
#state_verbose: True

# The state_output setting controls which results will be output full multi line
# full, terse - each state will be full/terse
# mixed - only states with errors will be full
# changes - states with changes and errors will be full
# full_id, mixed_id, changes_id and terse_id are also allowed;
# when set, the state ID will be used as name in the output
#state_output: full

# The state_output_diff setting changes whether or not the output from
# successful states is returned. Useful when even the terse output of these
# states is cluttering the logs. Set it to True to ignore them.
#state_output_diff: False

# Automatically aggregate all states that have support for mod_aggregate by
# setting to 'True'. Or pass a list of state module names to automatically
# aggregate just those types.
#
# state_aggregate:
#   - pkg
#
#state_aggregate: False

# Send progress events as each function in a state run completes execution
# by setting to 'True'. Progress events are in the format
# 'salt/job/<JID>/prog/<MID>/<RUN NUM>'.
#state_events: False

#####      File Server settings      #####
##########################################
# Salt runs a lightweight file server written in zeromq to deliver files to
# minions. This file server is built into the master daemon and does not
# require a dedicated port.

# The file server works on environments passed to the master, each environment
# can have multiple root directories, the subdirectories in the multiple file
# roots cannot match, otherwise the downloaded files will not be able to be
# reliably ensured. A base environment is required to house the top file.
# Example:
# file_roots:
#   base:
#     - /srv/salt/
#   dev:
#     - /srv/salt/dev/services
#     - /srv/salt/dev/states
#   prod:
#     - /srv/salt/prod/services
#     - /srv/salt/prod/states
#
#file_roots:
#  base:
```

```
#    - /srv/salt
#

# The master_roots setting configures a master-only copy of the file_roots dictionary,
# used by the state compiler.
#master_roots: /srv/salt-master

# When using multiple environments, each with their own top file, the
# default behaviour is an unordered merge. To prevent top files from
# being merged together and instead to only use the top file from the
# requested environment, set this value to 'same'.
#top_file_merging_strategy: merge

# To specify the order in which environments are merged, set the ordering
# in the env_order option. Given a conflict, the last matching value will
# win.
#env_order: ['base', 'dev', 'prod']

# If top_file_merging_strategy is set to 'same' and an environment does not
# contain a top file, the top file in the environment specified by default_top
# will be used instead.
#default_top: base

# The hash_type is the hash to use when discovering the hash of a file on
# the master server. The default is sha256, but md5, sha1, sha224, sha384 and
# sha512 are also supported.
#
# WARNING: While md5 and sha1 are also supported, do not use them due to the
# high chance of possible collisions and thus security breach.
#
# Prior to changing this value, the master should be stopped and all Salt
# caches should be cleared.
#hash_type: sha256

# The buffer size in the file server can be adjusted here:
#file_buffer_size: 1048576

# A regular expression (or a list of expressions) that will be matched
# against the file path before syncing the modules and states to the minions.
# This includes files affected by the file.recurse state.
# For example, if you manage your custom modules and states in subversion
# and don't want all the '.svn' folders and content synced to your minions,
# you could set this to '/\.svn($|/)'. By default nothing is ignored.
#file_ignore_regex:
#  - '/\.svn($|/)'
#  - '/\.git($|/)'

# A file glob (or list of file globs) that will be matched against the file
# path before syncing the modules and states to the minions. This is similar
# to file_ignore_regex above, but works on globs instead of regex. By default
# nothing is ignored.
# file_ignore_glob:
#  - '*.pyc'
#  - '*/somefolder/*.bak'
#  - '*.swp'

# File Server Backend
#
```

```
# Salt supports a modular fileserver backend system, this system allows
# the salt master to link directly to third party systems to gather and
# manage the files available to minions. Multiple backends can be
# configured and will be searched for the requested file in the order in which
# they are defined here. The default setting only enables the standard backend
# "roots" which uses the "file_roots" option.
#fileserver_backend:
#  - roots
#
# To use multiple backends list them in the order they are searched:
#fileserver_backend:
#  - git
#  - roots
#
# Uncomment the line below if you do not want the file_server to follow
# symlinks when walking the filesystem tree. This is set to True
# by default. Currently this only applies to the default roots
# fileserver_backend.
#fileserver_followsymlinks: False
#
# Uncomment the line below if you do not want symlinks to be
# treated as the files they are pointing to. By default this is set to
# False. By uncommenting the line below, any detected symlink while listing
# files on the Master will not be returned to the Minion.
#fileserver_ignoresymlinks: True
#
# By default, the Salt fileserver recurses fully into all defined environments
# to attempt to find files. To limit this behavior so that the fileserver only
# traverses directories with SLS files and special Salt directories like _modules,
# enable the option below. This might be useful for installations where a file root
# has a very large number of files and performance is impacted. Default is False.
# fileserver_limit_traversal: False
#
# The fileserver can fire events off every time the fileserver is updated,
# these are disabled by default, but can be easily turned on by setting this
# flag to True
#fileserver_events: False

# Git File Server Backend Configuration
#
# Optional parameter used to specify the provider to be used for gitfs. Must be
# either pygit2 or gitpython. If unset, then both will be tried (in that
# order), and the first one with a compatible version installed will be the
# provider that is used.
#
#gitfs_provider: pygit2

# Along with gitfs_password, is used to authenticate to HTTPS remotes.
# gitfs_user: ''

# Along with gitfs_user, is used to authenticate to HTTPS remotes.
# This parameter is not required if the repository does not use authentication.
#gitfs_password: ''

# By default, Salt will not authenticate to an HTTP (non-HTTPS) remote.
# This parameter enables authentication over HTTP. Enable this at your own risk.
#gitfs_insecure_auth: False
```

```
# Along with gitfs_privkey (and optionally gitfs_passphrase), is used to
# authenticate to SSH remotes. This parameter (or its per-remote counterpart)
# is required for SSH remotes.
#gitfs_pubkey: ''

# Along with gitfs_pubkey (and optionally gitfs_passphrase), is used to
# authenticate to SSH remotes. This parameter (or its per-remote counterpart)
# is required for SSH remotes.
#gitfs_privkey: ''

# This parameter is optional, required only when the SSH key being used to
# authenticate is protected by a passphrase.
#gitfs_passphrase: ''

# When using the git fileserver backend at least one git remote needs to be
# defined. The user running the salt master will need read access to the repo.
#
# The repos will be searched in order to find the file requested by a client
# and the first repo to have the file will return it.
# When using the git backend branches and tags are translated into salt
# environments.
# Note: file:// repos will be treated as a remote, so refs you want used must
# exist in that repo as *local* refs.
#gitfs_remotes:
#  - git://github.com/saltstack/salt-states.git
#  - file:///var/git/saltmaster
#
# The gitfs_ssl_verify option specifies whether to ignore ssl certificate
# errors when contacting the gitfs backend. You might want to set this to
# false if you're using a git backend that uses a self-signed certificate but
# keep in mind that setting this flag to anything other than the default of True
# is a security concern, you may want to try using the ssh transport.
#gitfs_ssl_verify: True
#
# The gitfs_root option gives the ability to serve files from a subdirectory
# within the repository. The path is defined relative to the root of the
# repository and defaults to the repository root.
#gitfs_root: somefolder/otherfolder
#
# The refspecs fetched by gitfs remotes
#gitfs_refspecs:
#  - '+refs/heads/*:refs/remotes/origin/*'
#  - '+refs/tags/*:refs/tags/*'
#
#
#####         Pillar settings        #####
##########################################
# Salt Pillars allow for the building of global data that can be made selectively
# available to different minions based on minion grain filtering. The Salt
# Pillar is laid out in the same fashion as the file server, with environments,
# a top file and sls files. However, pillar data does not need to be in the
# highstate format, and is generally just key/value pairs.
#pillar_roots:
#  base:
#    - /srv/pillar
#
#ext_pillar:
#  - hiera: /etc/hiera.yaml
```

```
#  - cmd_yaml: cat /etc/salt/yaml


# A list of paths to be recursively decrypted during pillar compilation.
# Entries in this list can be formatted either as a simple string, or as a
# key/value pair, with the key being the pillar location, and the value being
# the renderer to use for pillar decryption. If the former is used, the
# renderer specified by decrypt_pillar_default will be used.
#decrypt_pillar:
#  - 'foo:bar': gpg
#  - 'lorem:ipsum:dolor'

# The delimiter used to distinguish nested data structures in the
# decrypt_pillar option.
#decrypt_pillar_delimiter: ':'

# The default renderer used for decryption, if one is not specified for a given
# pillar key in decrypt_pillar.
#decrypt_pillar_default: gpg

# List of renderers which are permitted to be used for pillar decryption.
#decrypt_pillar_renderers:
#  - gpg

# The ext_pillar_first option allows for external pillar sources to populate
# before file system pillar. This allows for targeting file system pillar from
# ext_pillar.
#ext_pillar_first: False

# The external pillars permitted to be used on-demand using pillar.ext
#on_demand_ext_pillar:
#  - libvirt
#  - virtkey

# The pillar_gitfs_ssl_verify option specifies whether to ignore ssl certificate
# errors when contacting the pillar gitfs backend. You might want to set this to
# false if you're using a git backend that uses a self-signed certificate but
# keep in mind that setting this flag to anything other than the default of True
# is a security concern, you may want to try using the ssh transport.
#pillar_gitfs_ssl_verify: True

# The pillar_opts option adds the master configuration file data to a dict in
# the pillar called "master". This is used to set simple configurations in the
# master config file that can then be used on minions.
#pillar_opts: False

# The pillar_safe_render_error option prevents the master from passing pillar
# render errors to the minion. This is set on by default because the error could
# contain templating data which would give that minion information it shouldn't
# have, like a password! When set true the error message will only show:
#   Rendering SLS 'my.sls' failed. Please see master log for details.
#pillar_safe_render_error: True

# The pillar_source_merging_strategy option allows you to configure merging strategy
# between different sources. It accepts five values: none, recurse, aggregate,
↪overwrite,
# or smart. None will not do any merging at all. Recurse will merge recursively
↪mapping of data.
```

```
# Aggregate instructs aggregation of elements between sources that use the #!yamlex␣
↪renderer. Overwrite
# will overwrite elements according the order in which they are processed. This is
# behavior of the 2014.1 branch and earlier. Smart guesses the best strategy based
# on the "renderer" setting and is the default value.
#pillar_source_merging_strategy: smart

# Recursively merge lists by aggregating them instead of replacing them.
#pillar_merge_lists: False

# Set this option to True to force the pillarenv to be the same as the effective
# saltenv when running states. If pillarenv is specified this option will be
# ignored.
#pillarenv_from_saltenv: False

# Set this option to 'True' to force a 'KeyError' to be raised whenever an
# attempt to retrieve a named value from pillar fails. When this option is set
# to 'False', the failed attempt returns an empty string. Default is 'False'.
#pillar_raise_on_missing: False

# Git External Pillar (git_pillar) Configuration Options
#
# Specify the provider to be used for git_pillar. Must be either pygit2 or
# gitpython. If unset, then both will be tried in that same order, and the
# first one with a compatible version installed will be the provider that
# is used.
#git_pillar_provider: pygit2

# If the desired branch matches this value, and the environment is omitted
# from the git_pillar configuration, then the environment for that git_pillar
# remote will be base.
#git_pillar_base: master

# If the branch is omitted from a git_pillar remote, then this branch will
# be used instead
#git_pillar_branch: master

# Environment to use for git_pillar remotes. This is normally derived from
# the branch/tag (or from a per-remote env parameter), but if set this will
# override the process of deriving the env from the branch/tag name.
#git_pillar_env: ''

# Path relative to the root of the repository where the git_pillar top file
# and SLS files are located.
#git_pillar_root: ''

# Specifies whether or not to ignore SSL certificate errors when contacting
# the remote repository.
#git_pillar_ssl_verify: False

# When set to False, if there is an update/checkout lock for a git_pillar
# remote and the pid written to it is not running on the master, the lock
# file will be automatically cleared and a new lock will be obtained.
#git_pillar_global_lock: True

# Git External Pillar Authentication Options
#
# Along with git_pillar_password, is used to authenticate to HTTPS remotes.
```

```
#git_pillar_user: ''

# Along with git_pillar_user, is used to authenticate to HTTPS remotes.
# This parameter is not required if the repository does not use authentication.
#git_pillar_password: ''

# By default, Salt will not authenticate to an HTTP (non-HTTPS) remote.
# This parameter enables authentication over HTTP.
#git_pillar_insecure_auth: False

# Along with git_pillar_privkey (and optionally git_pillar_passphrase),
# is used to authenticate to SSH remotes.
#git_pillar_pubkey: ''

# Along with git_pillar_pubkey (and optionally git_pillar_passphrase),
# is used to authenticate to SSH remotes.
#git_pillar_privkey: ''

# This parameter is optional, required only when the SSH key being used
# to authenticate is protected by a passphrase.
#git_pillar_passphrase: ''

# The refspecs fetched by git_pillar remotes
#git_pillar_refspecs:
#  - '+refs/heads/*:refs/remotes/origin/*'
#  - '+refs/tags/*:refs/tags/*'

# A master can cache pillars locally to bypass the expense of having to render them
# for each minion on every request. This feature should only be enabled in cases
# where pillar rendering time is known to be unsatisfactory and any attendant security
# concerns about storing pillars in a master cache have been addressed.
#
# When enabling this feature, be certain to read through the additional ``pillar_cache_
↪*``
# configuration options to fully understand the tunable parameters and their
↪implications.
#
# Note: setting ``pillar_cache: True`` has no effect on targeting Minions with Pillars.
# See https://docs.saltstack.com/en/latest/topics/targeting/pillar.html
#pillar_cache: False

# If and only if a master has set ``pillar_cache: True``, the cache TTL controls the
↪amount
# of time, in seconds, before the cache is considered invalid by a master and a fresh
# pillar is recompiled and stored.
#pillar_cache_ttl: 3600

# If and only if a master has set `pillar_cache: True`, one of several storage providers
# can be utilized.
#
# `disk`: The default storage backend. This caches rendered pillars to the master cache.
#         Rendered pillars are serialized and deserialized as msgpack structures for
↪speed.
#         Note that pillars are stored UNENCRYPTED. Ensure that the master cache
#         has permissions set appropriately. (Same defaults are provided.)
#
# memory: [EXPERIMENTAL] An optional backend for pillar caches which uses a pure-Python
#         in-memory data structure for maximal performance. There are several caveats,
```

```
#         however. First, because each master worker contains its own in-memory cache,
#         there is no guarantee of cache consistency between minion requests. This
#         works best in situations where the pillar rarely if ever changes. Secondly,
#         and perhaps more importantly, this means that unencrypted pillars will
#         be accessible to any process which can examine the memory of the ``salt-
↪master``!
#         This may represent a substantial security risk.
#
#pillar_cache_backend: disk


######        Reactor Settings        #####
###########################################
# Define a salt reactor. See https://docs.saltstack.com/en/latest/topics/reactor/
#reactor: []

#Set the TTL for the cache of the reactor configuration.
#reactor_refresh_interval: 60

#Configure the number of workers for the runner/wheel in the reactor.
#reactor_worker_threads: 10

#Define the queue size for workers in the reactor.
#reactor_worker_hwm: 10000


#####          Syndic settings       #####
##########################################
# The Salt syndic is used to pass commands through a master from a higher
# master. Using the syndic is simple. If this is a master that will have
# syndic servers(s) below it, then set the "order_masters" setting to True.
#
# If this is a master that will be running a syndic daemon for passthrough, then
# the "syndic_master" setting needs to be set to the location of the master server
# to receive commands from.

# Set the order_masters setting to True if this master will command lower
# masters' syndic interfaces.
#order_masters: False

# If this master will be running a salt syndic daemon, syndic_master tells
# this master where to receive commands from.
#syndic_master: masterofmasters

# This is the 'ret_port' of the MasterOfMaster:
#syndic_master_port: 4506

# PID file of the syndic daemon:
#syndic_pidfile: /var/run/salt-syndic.pid

# The log file of the salt-syndic daemon:
#syndic_log_file: /var/log/salt/syndic

# The behaviour of the multi-syndic when connection to a master of masters failed.
# Can specify ``random`` (default) or ``ordered``. If set to ``random``, masters
# will be iterated in random order. If ``ordered`` is specified, the configured
# order will be used.
#syndic_failover: random
```

```
# The number of seconds for the salt client to wait for additional syndics to
# check in with their lists of expected minions before giving up.
#syndic_wait: 5


#####      Peer Publish settings     #####
##########################################
# Salt minions can send commands to other minions, but only if the minion is
# allowed to. By default "Peer Publication" is disabled, and when enabled it
# is enabled for specific minions and specific commands. This allows secure
# compartmentalization of commands based on individual minions.

# The configuration uses regular expressions to match minions and then a list
# of regular expressions to match functions. The following will allow the
# minion authenticated as foo.example.com to execute functions from the test
# and pkg modules.
#peer:
#  foo.example.com:
#    - test.*
#    - pkg.*
#
# This will allow all minions to execute all commands:
#peer:
#  .*:
#    - .*
#
# This is not recommended, since it would allow anyone who gets root on any
# single minion to instantly have root on all of the minions!

# Minions can also be allowed to execute runners from the salt master.
# Since executing a runner from the minion could be considered a security risk,
# it needs to be enabled. This setting functions just like the peer setting
# except that it opens up runners instead of module functions.
#
# All peer runner support is turned off by default and must be enabled before
# using. This will enable all peer runners for all minions:
#peer_run:
#  .*:
#    - .*
#
# To enable just the manage.up runner for the minion foo.example.com:
#peer_run:
#  foo.example.com:
#    - manage.up
#
#
#####         Mine settings     #####
#####################################
# Restrict mine.get access from minions. By default any minion has a full access
# to get all mine data from master cache. In acl definion below, only pcre matches
# are allowed.
# mine_get:
#   .*:
#     - .*
#
# The example below enables minion foo.example.com to get 'network.interfaces' mine
# data only, minions web* to get all network.* and disk.* mine data and all other
```

```
# minions won't get any mine data.
# mine_get:
#   foo.example.com:
#     - network.interfaces
#   web.*:
#     - network.*
#     - disk.*


#####         Logging settings       #####
##########################################
# The location of the master log file
# The master log can be sent to a regular file, local path name, or network
# location. Remote logging works best when configured to use rsyslogd(8) (e.g.:
# ``file:///dev/log``), with rsyslogd(8) configured for network logging. The URI
# format is: <file|udp|tcp>://<host|socketpath>:<port-if-required>/<log-facility>
#log_file: /var/log/salt/master
#log_file: file:///dev/log
#log_file: udp://loghost:10514

#log_file: /var/log/salt/master
#key_logfile: /var/log/salt/key

# The level of messages to send to the console.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
#
# The following log levels are considered INSECURE and may log sensitive data:
# ['garbage', 'trace', 'debug']
#
#log_level: warning

# The level of messages to send to the log file.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
# If using 'log_granular_levels' this must be set to the highest desired level.
#log_level_logfile: warning

# The date and time format used in log messages. Allowed date/time formatting
# can be seen here: http://docs.python.org/library/time.html#time.strftime
#log_datefmt: '%H:%M:%S'
#log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'

# The format of the console logging messages. Allowed formatting options can
# be seen here: http://docs.python.org/library/logging.html#logrecord-attributes
#
# Console log colors are specified by these additional formatters:
#
# %(colorlevel)s
# %(colorname)s
# %(colorprocess)s
# %(colormsg)s
#
# Since it is desirable to include the surrounding brackets, '[' and ']', in
# the coloring of the messages, these color formatters also include padding as
# well.  Color LogRecord attributes are only available for console logging.
#
#log_fmt_console: '%(colorlevel)s %(colormsg)s'
#log_fmt_console: '[%(levelname)-8s] %(message)s'
#
```

```
#log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'

# This can be used to control logging levels more specificically.  This
# example sets the main salt library at the 'warning' level, but sets
# 'salt.modules' to log at the 'debug' level:
#   log_granular_levels:
#     'salt': 'warning'
#     'salt.modules': 'debug'
#
#log_granular_levels: {}


#####         Node Groups           ######
##########################################
# Node groups allow for logical groupings of minion nodes. A group consists of
# a group name and a compound target. Nodgroups can reference other nodegroups
# with 'N@' classifier. Ensure that you do not have circular references.
#
#nodegroups:
#  group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl*.domain.com'
#  group2: 'G@os:Debian and foo.domain.com'
#  group3: 'G@os:Debian and N@group1'
#  group4:
#    - 'G@foo:bar'
#    - 'or'
#    - 'G@foo:baz'


#####     Range Cluster settings     #####
##########################################
# The range server (and optional port) that serves your cluster information
# https://github.com/ytoolshed/range/wiki/%22yamlfile%22-module-file-spec
#
#range_server: range:80


#####  Windows Software Repo settings #####
###########################################
# Location of the repo on the master:
#winrepo_dir_ng: '/srv/salt/win/repo-ng'
#
# List of git repositories to include with the local repo:
#winrepo_remotes_ng:
#  - 'https://github.com/saltstack/salt-winrepo-ng.git'


#####  Windows Software Repo settings - Pre 2015.8 #####
########################################################
# Legacy repo settings for pre-2015.8 Windows minions.
#
# Location of the repo on the master:
#winrepo_dir: '/srv/salt/win/repo'
#
# Location of the master's repo cache file:
#winrepo_mastercachefile: '/srv/salt/win/repo/winrepo.p'
#
# List of git repositories to include with the local repo:
#winrepo_remotes:
```

```
#  - 'https://github.com/saltstack/salt-winrepo.git'

# The refspecs fetched by winrepo remotes
#winrepo_refspecs:
#  - '+refs/heads/*:refs/remotes/origin/*'
#  - '+refs/tags/*:refs/tags/*'
#

#####      Returner settings          ######
############################################
# Which returner(s) will be used for minion's result:
#return: mysql


######    Miscellaneous  settings     ######
############################################
# Default match type for filtering events tags: startswith, endswith, find, regex,
↪fnmatch
#event_match_type: startswith

# Save runner returns to the job cache
#runner_returns: True

# Permanently include any available Python 3rd party modules into thin and minimal Salt
# when they are generated for Salt-SSH or other purposes.
# The modules should be named by the names they are actually imported inside the Python.
# The value of the parameters can be either one module or a comma separated list of them.
#thin_extra_mods: foo,bar
#min_extra_mods: foo,bar,baz


######      Keepalive settings        ######
############################################
# Warning: Failure to set TCP keepalives on the salt-master can result in
# not detecting the loss of a minion when the connection is lost or when
# it's host has been terminated without first closing the socket.
# Salt's Presence System depends on this connection status to know if a minion
# is "present".
# ZeroMQ now includes support for configuring SO_KEEPALIVE if supported by
# the OS. If connections between the minion and the master pass through
# a state tracking device such as a firewall or VPN gateway, there is
# the risk that it could tear down the connection the master and minion
# without informing either party that their connection has been taken away.
# Enabling TCP Keepalives prevents this from happening.

# Overall state of TCP Keepalives, enable (1 or True), disable (0 or False)
# or leave to the OS defaults (-1), on Linux, typically disabled. Default True, enabled.
#tcp_keepalive: True

# How long before the first keepalive should be sent in seconds. Default 300
# to send the first keepalive after 5 minutes, OS default (-1) is typically 7200 seconds
# on Linux see /proc/sys/net/ipv4/tcp_keepalive_time.
#tcp_keepalive_idle: 300

# How many lost probes are needed to consider the connection lost. Default -1
# to use OS defaults, typically 9 on Linux, see /proc/sys/net/ipv4/tcp_keepalive_probes.
#tcp_keepalive_cnt: -1
```

```
# How often, in seconds, to send keepalives after the first one. Default -1 to
# use OS defaults, typically 75 seconds on Linux, see
# /proc/sys/net/ipv4/tcp_keepalive_intvl.
#tcp_keepalive_intvl: -1
```

3.4.2 Example minion configuration file

```
##### Primary configuration settings #####
##########################################
# This configuration file is used to manage the behavior of the Salt Minion.
# With the exception of the location of the Salt Master Server, values that are
# commented out but have an empty line after the comment are defaults that need
# not be set in the config. If there is no blank line after the comment, the
# value is presented as an example and is not the default.

# Per default the minion will automatically include all config files
# from minion.d/*.conf (minion.d is a directory in the same directory
# as the main minion config file).
#default_include: minion.d/*.conf

# Set the location of the salt master server. If the master server cannot be
# resolved, then the minion will fail to start.
#master: salt

# Set http proxy information for the minion when doing requests
#proxy_host:
#proxy_port:
#proxy_username:
#proxy_password:

# If multiple masters are specified in the 'master' setting, the default behavior
# is to always try to connect to them in the order they are listed. If random_master is
# set to True, the order will be randomized instead. This can be helpful in distributing
# the load of many minions executing salt-call requests, for example, from a cron job.
# If only one master is listed, this setting is ignored and a warning will be logged.
# NOTE: If master_type is set to failover, use master_shuffle instead.
#random_master: False

# Use if master_type is set to failover.
#master_shuffle: False

# Minions can connect to multiple masters simultaneously (all masters
# are "hot"), or can be configured to failover if a master becomes
# unavailable.  Multiple hot masters are configured by setting this
# value to "str".  Failover masters can be requested by setting
# to "failover".  MAKE SURE TO SET master_alive_interval if you are
# using failover.
# Setting master_type to 'disable' let's you have a running minion (with engines and
# beacons) without a master connection
# master_type: str

# Poll interval in seconds for checking if the master is still there.  Only
# respected if master_type above is "failover". To disable the interval entirely,
# set the value to -1. (This may be necessary on machines which have high numbers
# of TCP connections, such as load balancers.)
# master_alive_interval: 30
```

```
# If the minion is in multi-master mode and the master_type configuration option
# is set to "failover", this setting can be set to "True" to force the minion
# to fail back to the first master in the list if the first master is back online.
#master_failback: False

# If the minion is in multi-master mode, the "master_type" configuration is set to
# "failover", and the "master_failback" option is enabled, the master failback
# interval can be set to ping the top master with this interval, in seconds.
#master_failback_interval: 0

# Set whether the minion should connect to the master via IPv6:
#ipv6: False

# Set the number of seconds to wait before attempting to resolve
# the master hostname if name resolution fails. Defaults to 30 seconds.
# Set to zero if the minion should shutdown and not retry.
# retry_dns: 30

# Set the number of times to attempt to resolve
# the master hostname if name resolution fails. Defaults to None,
# which will attempt the resolution indefinitely.
# retry_dns_count: 3

# Set the port used by the master reply and authentication server.
#master_port: 4506

# The user to run salt.
#user: root

# The user to run salt remote execution commands as via sudo. If this option is
# enabled then sudo will be used to change the active user executing the remote
# command. If enabled the user will need to be allowed access via the sudoers
# file for the user that the salt minion is configured to run as. The most
# common option would be to use the root user. If this option is set the user
# option should also be set to a non-root user. If migrating from a root minion
# to a non root minion the minion cache should be cleared and the minion pki
# directory will need to be changed to the ownership of the new user.
#sudo_user: root

# Specify the location of the daemon process ID file.
#pidfile: /var/run/salt-minion.pid

# The root directory prepended to these options: pki_dir, cachedir, log_file,
# sock_dir, pidfile.
#root_dir: /

# The path to the minion's configuration file.
#conf_file: /etc/salt/minion

# The directory to store the pki information in
#pki_dir: /etc/salt/pki/minion

# Explicitly declare the id for this minion to use, if left commented the id
# will be the hostname as returned by the python call: socket.getfqdn()
# Since salt uses detached ids it is possible to run multiple minions on the
# same machine but with different ids, this can be useful for salt compute
# clusters.
```

```
#id:

# Cache the minion id to a file when the minion's id is not statically defined
# in the minion config. Defaults to "True". This setting prevents potential
# problems when automatic minion id resolution changes, which can cause the
# minion to lose connection with the master. To turn off minion id caching,
# set this config to ``False``.
#minion_id_caching: True

# Append a domain to a hostname in the event that it does not exist.  This is
# useful for systems where socket.getfqdn() does not actually result in a
# FQDN (for instance, Solaris).
#append_domain:

# Custom static grains for this minion can be specified here and used in SLS
# files just like all other grains. This example sets 4 custom grains, with
# the 'roles' grain having two values that can be matched against.
#grains:
#  roles:
#    - webserver
#    - memcache
#  deployment: datacenter4
#  cabinet: 13
#  cab_u: 14-15
#
# Where cache data goes.
# This data may contain sensitive data and should be protected accordingly.
#cachedir: /var/cache/salt/minion

# Append minion_id to these directories.  Helps with
# multiple proxies and minions running on the same machine.
# Allowed elements in the list: pki_dir, cachedir, extension_modules
# Normally not needed unless running several proxies and/or minions on the same machine
# Defaults to ['cachedir'] for proxies, [] (empty list) for regular minions
#append_minionid_config_dirs:

# Verify and set permissions on configuration directories at startup.
#verify_env: True

# The minion can locally cache the return data from jobs sent to it, this
# can be a good way to keep track of jobs the minion has executed
# (on the minion side). By default this feature is disabled, to enable, set
# cache_jobs to True.
#cache_jobs: False

# Set the directory used to hold unix sockets.
#sock_dir: /var/run/salt/minion

# The minion can take a while to start up when lspci and/or dmidecode is used
# to populate the grains for the minion. Set this to False if you do not need
# GPU hardware grains for your minion.
# enable_gpu_grains: True

# Set the default outputter used by the salt-call command. The default is
# "nested".
#output: nested

# To set a list of additional directories to search for salt outputters, set the
```

```
# outputter_dirs option.
#outputter_dirs: []

# By default output is colored. To disable colored output, set the color value
# to False.
#color: True

# Do not strip off the colored output from nested results and state outputs
# (true by default).
# strip_colors: False

# Backup files that are replaced by file.managed and file.recurse under
# 'cachedir'/file_backup relative to their original location and appended
# with a timestamp. The only valid setting is "minion". Disabled by default.
#
# Alternatively this can be specified for each file in state files:
# /etc/ssh/sshd_config:
#   file.managed:
#     - source: salt://ssh/sshd_config
#     - backup: minion
#
#backup_mode: minion

# When waiting for a master to accept the minion's public key, salt will
# continuously attempt to reconnect until successful. This is the time, in
# seconds, between those reconnection attempts.
#acceptance_wait_time: 10

# If this is nonzero, the time between reconnection attempts will increase by
# acceptance_wait_time seconds per iteration, up to this maximum. If this is
# set to zero, the time between reconnection attempts will stay constant.
#acceptance_wait_time_max: 0

# If the master rejects the minion's public key, retry instead of exiting.
# Rejected keys will be handled the same as waiting on acceptance.
#rejected_retry: False

# When the master key changes, the minion will try to re-auth itself to receive
# the new master key. In larger environments this can cause a SYN flood on the
# master because all minions try to re-auth immediately. To prevent this and
# have a minion wait for a random amount of time, use this optional parameter.
# The wait-time will be a random number of seconds between 0 and the defined value.
#random_reauth_delay: 60


# To avoid overloading a master when many minions startup at once, a randomized
# delay may be set to tell the minions to wait before connecting to the master.
# This value is the number of seconds to choose from for a random number. For
# example, setting this value to 60 will choose a random number of seconds to delay
# on startup between zero seconds and sixty seconds. Setting to '0' will disable
# this feature.
#random_startup_delay: 0

# When waiting for a master to accept the minion's public key, salt will
# continuously attempt to reconnect until successful. This is the timeout value,
# in seconds, for each individual attempt. After this timeout expires, the minion
# will wait for acceptance_wait_time seconds before trying again. Unless your master
# is under unusually heavy load, this should be left at the default.
```

```
#auth_timeout: 60

# Number of consecutive SaltReqTimeoutError that are acceptable when trying to
# authenticate.
#auth_tries: 7

# The number of attempts to connect to a master before giving up.
# Set this to -1 for unlimited attempts. This allows for a master to have
# downtime and the minion to reconnect to it later when it comes back up.
# In 'failover' mode, it is the number of attempts for each set of masters.
# In this mode, it will cycle through the list of masters for each attempt.
#
# This is different than auth_tries because auth_tries attempts to
# retry auth attempts with a single master. auth_tries is under the
# assumption that you can connect to the master but not gain
# authorization from it. master_tries will still cycle through all
# the masters in a given try, so it is appropriate if you expect
# occasional downtime from the master(s).
#master_tries: 1

# If authentication fails due to SaltReqTimeoutError during a ping_interval,
# cause sub minion process to restart.
#auth_safemode: False

# Ping Master to ensure connection is alive (minutes).
#ping_interval: 0

# To auto recover minions if master changes IP address (DDNS)
#    auth_tries: 10
#    auth_safemode: False
#    ping_interval: 2
#
# Minions won't know master is missing until a ping fails. After the ping fail,
# the minion will attempt authentication and likely fails out and cause a restart.
# When the minion restarts it will resolve the masters IP and attempt to reconnect.

# If you don't have any problems with syn-floods, don't bother with the
# three recon_* settings described below, just leave the defaults!
#
# The ZeroMQ pull-socket that binds to the masters publishing interface tries
# to reconnect immediately, if the socket is disconnected (for example if
# the master processes are restarted). In large setups this will have all
# minions reconnect immediately which might flood the master (the ZeroMQ-default
# is usually a 100ms delay). To prevent this, these three recon_* settings
# can be used.
# recon_default: the interval in milliseconds that the socket should wait before
#                trying to reconnect to the master (1000ms = 1 second)
#
# recon_max: the maximum time a socket should wait. each interval the time to wait
#            is calculated by doubling the previous time. if recon_max is reached,
#            it starts again at recon_default. Short example:
#
#            reconnect 1: the socket will wait 'recon_default' milliseconds
#            reconnect 2: 'recon_default' * 2
#            reconnect 3: ('recon_default' * 2) * 2
#            reconnect 4: value from previous interval * 2
#            reconnect 5: value from previous interval * 2
#            reconnect x: if value >= recon_max, it starts again with recon_default
```

```
#
# recon_randomize: generate a random wait time on minion start. The wait time will
#                  be a random value between recon_default and recon_default +
#                  recon_max. Having all minions reconnect with the same recon_default
#                  and recon_max value kind of defeats the purpose of being able to
#                  change these settings. If all minions have the same values and your
#                  setup is quite large (several thousand minions), they will still
#                  flood the master. The desired behavior is to have timeframe within
#                  all minions try to reconnect.
#
# Example on how to use these settings. The goal: have all minions reconnect within a
# 60 second timeframe on a disconnect.
# recon_default: 1000
# recon_max: 59000
# recon_randomize: True
#
# Each minion will have a randomized reconnect value between 'recon_default'
# and 'recon_default + recon_max', which in this example means between 1000ms
# 60000ms (or between 1 and 60 seconds). The generated random-value will be
# doubled after each attempt to reconnect. Lets say the generated random
# value is 11 seconds (or 11000ms).
# reconnect 1: wait 11 seconds
# reconnect 2: wait 22 seconds
# reconnect 3: wait 33 seconds
# reconnect 4: wait 44 seconds
# reconnect 5: wait 55 seconds
# reconnect 6: wait time is bigger than 60 seconds (recon_default + recon_max)
# reconnect 7: wait 11 seconds
# reconnect 8: wait 22 seconds
# reconnect 9: wait 33 seconds
# reconnect x: etc.
#
# In a setup with ~6000 thousand hosts these settings would average the reconnects
# to about 100 per second and all hosts would be reconnected within 60 seconds.
# recon_default: 100
# recon_max: 5000
# recon_randomize: False
#
#
# The loop_interval sets how long in seconds the minion will wait between
# evaluating the scheduler and running cleanup tasks.  This defaults to 1
# second on the minion scheduler.
#loop_interval: 1

# Some installations choose to start all job returns in a cache or a returner
# and forgo sending the results back to a master. In this workflow, jobs
# are most often executed with --async from the Salt CLI and then results
# are evaluated by examining job caches on the minions or any configured returners.
# WARNING: Setting this to False will **disable** returns back to the master.
#pub_ret: True


# The grains can be merged, instead of overridden, using this option.
# This allows custom grains to defined different subvalues of a dictionary
# grain. By default this feature is disabled, to enable set grains_deep_merge
# to ``True``.
#grains_deep_merge: False
```

```
# The grains_refresh_every setting allows for a minion to periodically check
# its grains to see if they have changed and, if so, to inform the master
# of the new grains. This operation is moderately expensive, therefore
# care should be taken not to set this value too low.
#
# Note: This value is expressed in __minutes__!
#
# A value of 10 minutes is a reasonable default.
#
# If the value is set to zero, this check is disabled.
#grains_refresh_every: 1

# Cache grains on the minion. Default is False.
#grains_cache: False

# Cache rendered pillar data on the minion. Default is False.
# This may cause 'cachedir'/pillar to contain sensitive data that should be
# protected accordingly.
#minion_pillar_cache: False

# Grains cache expiration, in seconds. If the cache file is older than this
# number of seconds then the grains cache will be dumped and fully re-populated
# with fresh data. Defaults to 5 minutes. Will have no effect if 'grains_cache'
# is not enabled.
# grains_cache_expiration: 300

# Determines whether or not the salt minion should run scheduled mine updates.
# Defaults to "True". Set to "False" to disable the scheduled mine updates
# (this essentially just does not add the mine update function to the minion's
# scheduler).
#mine_enabled: True

# Determines whether or not scheduled mine updates should be accompanied by a job
# return for the job cache. Defaults to "False". Set to "True" to include job
# returns in the job cache for mine updates.
#mine_return_job: False

# Example functions that can be run via the mine facility
# NO mine functions are established by default.
# Note these can be defined in the minion's pillar as well.
#mine_functions:
#  test.ping: []
#  network.ip_addrs:
#    interface: eth0
#    cidr: '10.0.0.0/8'

# The number of minutes between mine updates.
#mine_interval: 60

# Windows platforms lack posix IPC and must rely on slower TCP based inter-
# process communications. Set ipc_mode to 'tcp' on such systems
#ipc_mode: ipc

# Overwrite the default tcp ports used by the minion when in tcp mode
#tcp_pub_port: 4510
#tcp_pull_port: 4511

# Passing very large events can cause the minion to consume large amounts of
```

```
# memory. This value tunes the maximum size of a message allowed onto the
# minion event bus. The value is expressed in bytes.
#max_event_size: 1048576

# To detect failed master(s) and fire events on connect/disconnect, set
# master_alive_interval to the number of seconds to poll the masters for
# connection events.
#
#master_alive_interval: 30

# The minion can include configuration from other files. To enable this,
# pass a list of paths to this option. The paths can be either relative or
# absolute; if relative, they are considered to be relative to the directory
# the main minion configuration file lives in (this file). Paths can make use
# of shell-style globbing. If no files are matched by a path passed to this
# option then the minion will log a warning message.
#
# Include a config file from some other path:
# include: /etc/salt/extra_config
#
# Include config from several files and directories:
#include:
#  - /etc/salt/extra_config
#  - /etc/roles/webserver

# The syndic minion can verify that it is talking to the correct master via the
# key fingerprint of the higher-level master with the "syndic_finger" config.
#syndic_finger: ''
#
#
#
#####   Minion module management     #####
##########################################
# Disable specific modules. This allows the admin to limit the level of
# access the master has to the minion.  The default here is the empty list,
# below is an example of how this needs to be formatted in the config file
#disable_modules:
#  - cmdmod
#  - test
#disable_returners: []

# This is the reverse of disable_modules.  The default, like disable_modules, is the
↪empty list,
# but if this option is set to *anything* then *only* those modules will load.
# Note that this is a very large hammer and it can be quite difficult to keep the
↪minion working
# the way you think it should since Salt uses many modules internally itself.  At a
↪bare minimum
# you need the following enabled or else the minion won't start.
#whitelist_modules:
#  - cmdmod
#  - test
#  - config

# Modules can be loaded from arbitrary paths. This enables the easy deployment
# of third party modules. Modules for returners and minions can be loaded.
# Specify a list of extra directories to search for minion modules and
# returners. These paths must be fully qualified!
```

```
#module_dirs: []
#returner_dirs: []
#states_dirs: []
#render_dirs: []
#utils_dirs: []
#
# A module provider can be statically overwritten or extended for the minion
# via the providers option, in this case the default module will be
# overwritten by the specified module. In this example the pkg module will
# be provided by the yumpkg5 module instead of the system default.
#providers:
#  pkg: yumpkg5
#
# Enable Cython modules searching and loading. (Default: False)
#cython_enable: False
#
# Specify a max size (in bytes) for modules on import. This feature is currently
# only supported on *nix operating systems and requires psutil.
# modules_max_memory: -1


#####    State Management Settings    #####
###########################################
# The state management system executes all of the state templates on the minion
# to enable more granular control of system state management. The type of
# template and serialization used for state management needs to be configured
# on the minion, the default renderer is yaml_jinja. This is a yaml file
# rendered from a jinja template, the available options are:
# yaml_jinja
# yaml_mako
# yaml_wempy
# json_jinja
# json_mako
# json_wempy
#
#renderer: yaml_jinja
#
# The failhard option tells the minions to stop immediately after the first
# failure detected in the state execution. Defaults to False.
#failhard: False
#
# Reload the modules prior to a highstate run.
#autoload_dynamic_modules: True
#
# clean_dynamic_modules keeps the dynamic modules on the minion in sync with
# the dynamic modules on the master, this means that if a dynamic module is
# not on the master it will be deleted from the minion. By default, this is
# enabled and can be disabled by changing this value to False.
#clean_dynamic_modules: True
#
# Normally, the minion is not isolated to any single environment on the master
# when running states, but the environment can be isolated on the minion side
# by statically setting it. Remember that the recommended way to manage
# environments is to isolate via the top file.
#environment: None
#
# Isolates the pillar environment on the minion side. This functions the same
# as the environment setting, but for pillar instead of states.
```

```
#pillarenv: None
#
# Set this option to True to force the pillarenv to be the same as the
# effective saltenv when running states. Note that if pillarenv is specified,
# this option will be ignored.
#pillarenv_from_saltenv: False
#
# Set this option to 'True' to force a 'KeyError' to be raised whenever an
# attempt to retrieve a named value from pillar fails. When this option is set
# to 'False', the failed attempt returns an empty string. Default is 'False'.
#pillar_raise_on_missing: False
#
# If using the local file directory, then the state top file name needs to be
# defined, by default this is top.sls.
#state_top: top.sls
#
# Run states when the minion daemon starts. To enable, set startup_states to:
# 'highstate' -- Execute state.highstate
# 'sls' -- Read in the sls_list option and execute the named sls files
# 'top' -- Read top_file option and execute based on that file on the Master
#startup_states: ''
#
# List of states to run when the minion starts up if startup_states is 'sls':
#sls_list:
#  - edit.vim
#  - hyper
#
# Top file to execute if startup_states is 'top':
#top_file: ''

# Automatically aggregate all states that have support for mod_aggregate by
# setting to True. Or pass a list of state module names to automatically
# aggregate just those types.
#
# state_aggregate:
#   - pkg
#
#state_aggregate: False

#####     File Directory Settings    #####
##########################################
# The Salt Minion can redirect all file server operations to a local directory,
# this allows for the same state tree that is on the master to be used if
# copied completely onto the minion. This is a literal copy of the settings on
# the master but used to reference a local directory on the minion.

# Set the file client. The client defaults to looking on the master server for
# files, but can be directed to look at the local file directory setting
# defined below by setting it to "local". Setting a local file_client runs the
# minion in masterless mode.
#file_client: remote

# The file directory works on environments passed to the minion, each environment
# can have multiple root directories, the subdirectories in the multiple file
# roots cannot match, otherwise the downloaded files will not be able to be
# reliably ensured. A base environment is required to house the top file.
# Example:
# file_roots:
```

```
#   base:
#     - /srv/salt/
#   dev:
#     - /srv/salt/dev/services
#     - /srv/salt/dev/states
#   prod:
#     - /srv/salt/prod/services
#     - /srv/salt/prod/states
#
#file_roots:
#  base:
#    - /srv/salt

# Uncomment the line below if you do not want the file_server to follow
# symlinks when walking the filesystem tree. This is set to True
# by default. Currently this only applies to the default roots
# fileserver_backend.
#fileserver_followsymlinks: False
#
# Uncomment the line below if you do not want symlinks to be
# treated as the files they are pointing to. By default this is set to
# False. By uncommenting the line below, any detected symlink while listing
# files on the Master will not be returned to the Minion.
#fileserver_ignoresymlinks: True
#
# By default, the Salt fileserver recurses fully into all defined environments
# to attempt to find files. To limit this behavior so that the fileserver only
# traverses directories with SLS files and special Salt directories like _modules,
# enable the option below. This might be useful for installations where a file root
# has a very large number of files and performance is negatively impacted. Default
# is False.
#fileserver_limit_traversal: False

# The hash_type is the hash to use when discovering the hash of a file on
# the local fileserver. The default is sha256, but md5, sha1, sha224, sha384
# and sha512 are also supported.
#
# WARNING: While md5 and sha1 are also supported, do not use them due to the
# high chance of possible collisions and thus security breach.
#
# Warning: Prior to changing this value, the minion should be stopped and all
# Salt caches should be cleared.
#hash_type: sha256

# The Salt pillar is searched for locally if file_client is set to local. If
# this is the case, and pillar data is defined, then the pillar_roots need to
# also be configured on the minion:
#pillar_roots:
#  base:
#    - /srv/pillar

# Set a hard-limit on the size of the files that can be pushed to the master.
# It will be interpreted as megabytes. Default: 100
#file_recv_max_size: 100
#
#
######        Security settings       #####
###########################################
```

```
# Enable "open mode", this mode still maintains encryption, but turns off
# authentication, this is only intended for highly secure environments or for
# the situation where your keys end up in a bad state. If you run in open mode
# you do so at your own risk!
#open_mode: False

# The size of key that should be generated when creating new keys.
#keysize: 2048

# Enable permissive access to the salt keys.  This allows you to run the
# master or minion as root, but have a non-root group be given access to
# your pki_dir.  To make the access explicit, root must belong to the group
# you've given access to. This is potentially quite insecure.
#permissive_pki_access: False

# The state_verbose and state_output settings can be used to change the way
# state system data is printed to the display. By default all data is printed.
# The state_verbose setting can be set to True or False, when set to False
# all data that has a result of True and no changes will be suppressed.
#state_verbose: True

# The state_output setting controls which results will be output full multi line
# full, terse - each state will be full/terse
# mixed - only states with errors will be full
# changes - states with changes and errors will be full
# full_id, mixed_id, changes_id and terse_id are also allowed;
# when set, the state ID will be used as name in the output
#state_output: full

# The state_output_diff setting changes whether or not the output from
# successful states is returned. Useful when even the terse output of these
# states is cluttering the logs. Set it to True to ignore them.
#state_output_diff: False

# The state_output_profile setting changes whether profile information
# will be shown for each state run.
#state_output_profile: True

# Fingerprint of the master public key to validate the identity of your Salt master
# before the initial key exchange. The master fingerprint can be found by running
# "salt-key -f master.pub" on the Salt master.
#master_finger: ''

# Use TLS/SSL encrypted connection between master and minion.
# Can be set to a dictionary containing keyword arguments corresponding to Python's
# 'ssl.wrap_socket' method.
# Default is None.
#ssl:
#    keyfile: <path_to_keyfile>
#    certfile: <path_to_certfile>
#    ssl_version: PROTOCOL_TLSv1_2

# Grains to be sent to the master on authentication to check if the minion's key
# will be accepted automatically. Needs to be configured on the master.
#autosign_grains:
#  - uuid
#  - server_id
```

```
######         Reactor Settings        #####
############################################
# Define a salt reactor. See https://docs.saltstack.com/en/latest/topics/reactor/
#reactor: []

#Set the TTL for the cache of the reactor configuration.
#reactor_refresh_interval: 60

#Configure the number of workers for the runner/wheel in the reactor.
#reactor_worker_threads: 10

#Define the queue size for workers in the reactor.
#reactor_worker_hwm: 10000


######          Thread settings        #####
############################################
# Disable multiprocessing support, by default when a minion receives a
# publication a new process is spawned and the command is executed therein.
#
# WARNING: Disabling multiprocessing may result in substantial slowdowns
# when processing large pillars. See https://github.com/saltstack/salt/issues/38758
# for a full explanation.
#multiprocessing: True

# Limit the maximum amount of processes or threads created by salt-minion.
# This is useful to avoid resource exhaustion in case the minion receives more
# publications than it is able to handle, as it limits the number of spawned
# processes or threads. -1 is the default and disables the limit.
#process_count_max: -1


#####         Logging settings       #####
##########################################
# The location of the minion log file
# The minion log can be sent to a regular file, local path name, or network
# location. Remote logging works best when configured to use rsyslogd(8) (e.g.:
# ``file:///dev/log``), with rsyslogd(8) configured for network logging. The URI
# format is: <file|udp|tcp>://<host|socketpath>:<port-if-required>/<log-facility>
#log_file: /var/log/salt/minion
#log_file: file:///dev/log
#log_file: udp://loghost:10514
#
#log_file: /var/log/salt/minion
#key_logfile: /var/log/salt/key

# The level of messages to send to the console.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
#
# The following log levels are considered INSECURE and may log sensitive data:
# ['garbage', 'trace', 'debug']
#
# Default: 'warning'
#log_level: warning

# The level of messages to send to the log file.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
```

```
# If using 'log_granular_levels' this must be set to the highest desired level.
# Default: 'warning'
#log_level_logfile:

# The date and time format used in log messages. Allowed date/time formatting
# can be seen here: http://docs.python.org/library/time.html#time.strftime
#log_datefmt: '%H:%M:%S'
#log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'

# The format of the console logging messages. Allowed formatting options can
# be seen here: http://docs.python.org/library/logging.html#logrecord-attributes
#
# Console log colors are specified by these additional formatters:
#
# %(colorlevel)s
# %(colorname)s
# %(colorprocess)s
# %(colormsg)s
#
# Since it is desirable to include the surrounding brackets, '[' and ']', in
# the coloring of the messages, these color formatters also include padding as
# well.  Color LogRecord attributes are only available for console logging.
#
#log_fmt_console: '%(colorlevel)s %(colormsg)s'
#log_fmt_console: '[%(levelname)-8s] %(message)s'
#
#log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'

# This can be used to control logging levels more specifically.  This
# example sets the main salt library at the 'warning' level, but sets
# 'salt.modules' to log at the 'debug' level:
#   log_granular_levels:
#     'salt': 'warning'
#     'salt.modules': 'debug'
#
#log_granular_levels: {}

# To diagnose issues with minions disconnecting or missing returns, ZeroMQ
# supports the use of monitor sockets to log connection events. This
# feature requires ZeroMQ 4.0 or higher.
#
# To enable ZeroMQ monitor sockets, set 'zmq_monitor' to 'True' and log at a
# debug level or higher.
#
# A sample log event is as follows:
#
# [DEBUG   ] ZeroMQ event: {'endpoint': 'tcp://127.0.0.1:4505', 'event': 512,
# 'value': 27, 'description': 'EVENT_DISCONNECTED'}
#
# All events logged will include the string 'ZeroMQ event'. A connection event
# should be logged as the minion starts up and initially connects to the
# master. If not, check for debug log level and that the necessary version of
# ZeroMQ is installed.
#
#zmq_monitor: False

# Number of times to try to authenticate with the salt master when reconnecting
# to the master
```

```
#tcp_authentication_retries: 5

######      Module configuration      #####
###########################################
# Salt allows for modules to be passed arbitrary configuration data, any data
# passed here in valid yaml format will be passed on to the salt minion modules
# for use. It is STRONGLY recommended that a naming convention be used in which
# the module name is followed by a . and then the value. Also, all top level
# data must be applied via the yaml dict construct, some examples:
#
# You can specify that all modules should run in test mode:
#test: True
#
# A simple value for the test module:
#test.foo: foo
#
# A list for the test module:
#test.bar: [baz,quo]
#
# A dict for the test module:
#test.baz: {spam: sausage, cheese: bread}
#
#
######      Update settings          ######
###########################################
# Using the features in Esky, a salt minion can both run as a frozen app and
# be updated on the fly. These options control how the update process
# (saltutil.update()) behaves.
#
# The url for finding and downloading updates. Disabled by default.
#update_url: False
#
# The list of services to restart after a successful update. Empty by default.
#update_restart_services: []


######      Keepalive settings        ######
############################################
# ZeroMQ now includes support for configuring SO_KEEPALIVE if supported by
# the OS. If connections between the minion and the master pass through
# a state tracking device such as a firewall or VPN gateway, there is
# the risk that it could tear down the connection the master and minion
# without informing either party that their connection has been taken away.
# Enabling TCP Keepalives prevents this from happening.

# Overall state of TCP Keepalives, enable (1 or True), disable (0 or False)
# or leave to the OS defaults (-1), on Linux, typically disabled. Default True, enabled.
#tcp_keepalive: True

# How long before the first keepalive should be sent in seconds. Default 300
# to send the first keepalive after 5 minutes, OS default (-1) is typically 7200 seconds
# on Linux see /proc/sys/net/ipv4/tcp_keepalive_time.
#tcp_keepalive_idle: 300

# How many lost probes are needed to consider the connection lost. Default -1
# to use OS defaults, typically 9 on Linux, see /proc/sys/net/ipv4/tcp_keepalive_probes.
#tcp_keepalive_cnt: -1
```

```
# How often, in seconds, to send keepalives after the first one. Default -1 to
# use OS defaults, typically 75 seconds on Linux, see
# /proc/sys/net/ipv4/tcp_keepalive_intvl.
#tcp_keepalive_intvl: -1


######   Windows Software settings    ######
############################################
# Location of the repository cache file on the master:
#win_repo_cachefile: 'salt://win/repo/winrepo.p'


######      Returner  settings        ######
############################################
# Default Minion returners. Can be a comma delimited string or a list:
#
#return: mysql
#
#return: mysql,slack,redis
#
#return:
#  - mysql
#  - hipchat
#  - slack


######    Miscellaneous  settings     ######
############################################
# Default match type for filtering events tags: startswith, endswith, find, regex,
↪fnmatch
#event_match_type: startswith
```

3.4.3 Example proxy minion configuration file

```
##### Primary configuration settings #####
##########################################
# This configuration file is used to manage the behavior of all Salt Proxy
# Minions on this host.
# With the exception of the location of the Salt Master Server, values that are
# commented out but have an empty line after the comment are defaults that need
# not be set in the config. If there is no blank line after the comment, the
# value is presented as an example and is not the default.

# Per default the minion will automatically include all config files
# from minion.d/*.conf (minion.d is a directory in the same directory
# as the main minion config file).
#default_include: minion.d/*.conf

# Backwards compatibility option for proxymodules created before 2015.8.2
# This setting will default to 'False' in the 2016.3.0 release
# Setting this to True adds proxymodules to the __opts__ dictionary.
# This breaks several Salt features (basically anything that serializes
# __opts__ over the wire) but retains backwards compatibility.
#add_proxymodule_to_opts: True

# Set the location of the salt master server. If the master server cannot be
```

```
# resolved, then the minion will fail to start.
#master: salt

# If a proxymodule has a function called 'grains', then call it during
# regular grains loading and merge the results with the proxy's grains
# dictionary.  Otherwise it is assumed that the module calls the grains
# function in a custom way and returns the data elsewhere
#
# Default to False for 2016.3 and 2016.11. Switch to True for 2017.7.0.
# proxy_merge_grains_in_module: True

# If a proxymodule has a function called 'alive' returning a boolean
# flag reflecting the state of the connection with the remove device,
# when this option is set as True, a scheduled job on the proxy will
# try restarting the connection. The polling frequency depends on the
# next option, 'proxy_keep_alive_interval'. Added in 2017.7.0.
# proxy_keep_alive: True

# The polling interval (in minutes) to check if the underlying connection
# with the remote device is still alive. This option requires
# 'proxy_keep_alive' to be configured as True and the proxymodule to
# implement the 'alive' function. Added in 2017.7.0.
# proxy_keep_alive_interval: 1

# By default, any proxy opens the connection with the remote device when
# initialized. Some proxymodules allow through this option to open/close
# the session per command. This requires the proxymodule to have this
# capability. Please consult the documentation to see if the proxy type
# used can be that flexible. Added in 2017.7.0.
# proxy_always_alive: True

# If multiple masters are specified in the 'master' setting, the default behavior
# is to always try to connect to them in the order they are listed. If random_master is
# set to True, the order will be randomized instead. This can be helpful in distributing
# the load of many minions executing salt-call requests, for example, from a cron job.
# If only one master is listed, this setting is ignored and a warning will be logged.
#random_master: False

# Minions can connect to multiple masters simultaneously (all masters
# are "hot"), or can be configured to failover if a master becomes
# unavailable.  Multiple hot masters are configured by setting this
# value to "str".  Failover masters can be requested by setting
# to "failover".  MAKE SURE TO SET master_alive_interval if you are
# using failover.
# master_type: str

# Poll interval in seconds for checking if the master is still there.  Only
# respected if master_type above is "failover".
# master_alive_interval: 30

# Set whether the minion should connect to the master via IPv6:
#ipv6: False

# Set the number of seconds to wait before attempting to resolve
# the master hostname if name resolution fails. Defaults to 30 seconds.
# Set to zero if the minion should shutdown and not retry.
# retry_dns: 30
```

```
# Set the port used by the master reply and authentication server.
#master_port: 4506

# The user to run salt.
#user: root

# Setting sudo_user will cause salt to run all execution modules under an sudo
# to the user given in sudo_user.  The user under which the salt minion process
# itself runs will still be that provided in the user config above, but all
# execution modules run by the minion will be rerouted through sudo.
#sudo_user: saltdev

# Specify the location of the daemon process ID file.
#pidfile: /var/run/salt-minion.pid

# The root directory prepended to these options: pki_dir, cachedir, log_file,
# sock_dir, pidfile.
#root_dir: /

# The directory to store the pki information in
#pki_dir: /etc/salt/pki/minion

# Where cache data goes.
# This data may contain sensitive data and should be protected accordingly.
#cachedir: /var/cache/salt/minion

# Append minion_id to these directories.  Helps with
# multiple proxies and minions running on the same machine.
# Allowed elements in the list: pki_dir, cachedir, extension_modules
# Normally not needed unless running several proxies and/or minions on the same machine
# Defaults to ['cachedir'] for proxies, [] (empty list) for regular minions
# append_minionid_config_dirs:
#   - cachedir



# Verify and set permissions on configuration directories at startup.
#verify_env: True

# The minion can locally cache the return data from jobs sent to it, this
# can be a good way to keep track of jobs the minion has executed
# (on the minion side). By default this feature is disabled, to enable, set
# cache_jobs to True.
#cache_jobs: False

# Set the directory used to hold unix sockets.
#sock_dir: /var/run/salt/minion

# Set the default outputter used by the salt-call command. The default is
# "nested".
#output: nested
#
# By default output is colored. To disable colored output, set the color value
# to False.
#color: True

# Do not strip off the colored output from nested results and state outputs
# (true by default).
```

```
# strip_colors: False

# Backup files that are replaced by file.managed and file.recurse under
# 'cachedir'/file_backup relative to their original location and appended
# with a timestamp. The only valid setting is "minion". Disabled by default.
#
# Alternatively this can be specified for each file in state files:
# /etc/ssh/sshd_config:
#   file.managed:
#     - source: salt://ssh/sshd_config
#     - backup: minion
#
#backup_mode: minion

# When waiting for a master to accept the minion's public key, salt will
# continuously attempt to reconnect until successful. This is the time, in
# seconds, between those reconnection attempts.
#acceptance_wait_time: 10

# If this is nonzero, the time between reconnection attempts will increase by
# acceptance_wait_time seconds per iteration, up to this maximum. If this is
# set to zero, the time between reconnection attempts will stay constant.
#acceptance_wait_time_max: 0

# If the master rejects the minion's public key, retry instead of exiting.
# Rejected keys will be handled the same as waiting on acceptance.
#rejected_retry: False

# When the master key changes, the minion will try to re-auth itself to receive
# the new master key. In larger environments this can cause a SYN flood on the
# master because all minions try to re-auth immediately. To prevent this and
# have a minion wait for a random amount of time, use this optional parameter.
# The wait-time will be a random number of seconds between 0 and the defined value.
#random_reauth_delay: 60

# When waiting for a master to accept the minion's public key, salt will
# continuously attempt to reconnect until successful. This is the timeout value,
# in seconds, for each individual attempt. After this timeout expires, the minion
# will wait for acceptance_wait_time seconds before trying again. Unless your master
# is under unusually heavy load, this should be left at the default.
#auth_timeout: 60

# Number of consecutive SaltReqTimeoutError that are acceptable when trying to
# authenticate.
#auth_tries: 7

# If authentication fails due to SaltReqTimeoutError during a ping_interval,
# cause sub minion process to restart.
#auth_safemode: False

# Ping Master to ensure connection is alive (minutes).
#ping_interval: 0

# To auto recover minions if master changes IP address (DDNS)
#    auth_tries: 10
#    auth_safemode: False
#    ping_interval: 90
#
```

```
# Minions won't know master is missing until a ping fails. After the ping fail,
# the minion will attempt authentication and likely fails out and cause a restart.
# When the minion restarts it will resolve the masters IP and attempt to reconnect.

# If you don't have any problems with syn-floods, don't bother with the
# three recon_* settings described below, just leave the defaults!
#
# The ZeroMQ pull-socket that binds to the masters publishing interface tries
# to reconnect immediately, if the socket is disconnected (for example if
# the master processes are restarted). In large setups this will have all
# minions reconnect immediately which might flood the master (the ZeroMQ-default
# is usually a 100ms delay). To prevent this, these three recon_* settings
# can be used.
# recon_default: the interval in milliseconds that the socket should wait before
#                trying to reconnect to the master (1000ms = 1 second)
#
# recon_max: the maximum time a socket should wait. each interval the time to wait
#            is calculated by doubling the previous time. if recon_max is reached,
#            it starts again at recon_default. Short example:
#
#            reconnect 1: the socket will wait 'recon_default' milliseconds
#            reconnect 2: 'recon_default' * 2
#            reconnect 3: ('recon_default' * 2) * 2
#            reconnect 4: value from previous interval * 2
#            reconnect 5: value from previous interval * 2
#            reconnect x: if value >= recon_max, it starts again with recon_default
#
# recon_randomize: generate a random wait time on minion start. The wait time will
#                  be a random value between recon_default and recon_default +
#                  recon_max. Having all minions reconnect with the same recon_default
#                  and recon_max value kind of defeats the purpose of being able to
#                  change these settings. If all minions have the same values and your
#                  setup is quite large (several thousand minions), they will still
#                  flood the master. The desired behavior is to have timeframe within
#                  all minions try to reconnect.
#
# Example on how to use these settings. The goal: have all minions reconnect within a
# 60 second timeframe on a disconnect.
# recon_default: 1000
# recon_max: 59000
# recon_randomize: True
#
# Each minion will have a randomized reconnect value between 'recon_default'
# and 'recon_default + recon_max', which in this example means between 1000ms
# 60000ms (or between 1 and 60 seconds). The generated random-value will be
# doubled after each attempt to reconnect. Lets say the generated random
# value is 11 seconds (or 11000ms).
# reconnect 1: wait 11 seconds
# reconnect 2: wait 22 seconds
# reconnect 3: wait 33 seconds
# reconnect 4: wait 44 seconds
# reconnect 5: wait 55 seconds
# reconnect 6: wait time is bigger than 60 seconds (recon_default + recon_max)
# reconnect 7: wait 11 seconds
# reconnect 8: wait 22 seconds
# reconnect 9: wait 33 seconds
# reconnect x: etc.
#
```

```
# In a setup with ~6000 thousand hosts these settings would average the reconnects
# to about 100 per second and all hosts would be reconnected within 60 seconds.
# recon_default: 100
# recon_max: 5000
# recon_randomize: False
#
#
# The loop_interval sets how long in seconds the minion will wait between
# evaluating the scheduler and running cleanup tasks. This defaults to a
# sane 60 seconds, but if the minion scheduler needs to be evaluated more
# often lower this value
#loop_interval: 60

# The grains_refresh_every setting allows for a minion to periodically check
# its grains to see if they have changed and, if so, to inform the master
# of the new grains. This operation is moderately expensive, therefore
# care should be taken not to set this value too low.
#
# Note: This value is expressed in __minutes__!
#
# A value of 10 minutes is a reasonable default.
#
# If the value is set to zero, this check is disabled.
#grains_refresh_every: 1

# Cache grains on the minion. Default is False.
#grains_cache: False

# Grains cache expiration, in seconds. If the cache file is older than this
# number of seconds then the grains cache will be dumped and fully re-populated
# with fresh data. Defaults to 5 minutes. Will have no effect if 'grains_cache'
# is not enabled.
# grains_cache_expiration: 300

# Windows platforms lack posix IPC and must rely on slower TCP based inter-
# process communications. Set ipc_mode to 'tcp' on such systems
#ipc_mode: ipc

# Overwrite the default tcp ports used by the minion when in tcp mode
#tcp_pub_port: 4510
#tcp_pull_port: 4511

# Passing very large events can cause the minion to consume large amounts of
# memory. This value tunes the maximum size of a message allowed onto the
# minion event bus. The value is expressed in bytes.
#max_event_size: 1048576

# To detect failed master(s) and fire events on connect/disconnect, set
# master_alive_interval to the number of seconds to poll the masters for
# connection events.
#
#master_alive_interval: 30

# The minion can include configuration from other files. To enable this,
# pass a list of paths to this option. The paths can be either relative or
# absolute; if relative, they are considered to be relative to the directory
# the main minion configuration file lives in (this file). Paths can make use
# of shell-style globbing. If no files are matched by a path passed to this
```

```
# option then the minion will log a warning message.
#
# Include a config file from some other path:
# include: /etc/salt/extra_config
#
# Include config from several files and directories:
#include:
#  - /etc/salt/extra_config
#  - /etc/roles/webserver
#
#
#
#####   Minion module management     #####
##########################################
# Disable specific modules. This allows the admin to limit the level of
# access the master has to the minion.
#disable_modules: [cmd,test]
#disable_returners: []
#
# Modules can be loaded from arbitrary paths. This enables the easy deployment
# of third party modules. Modules for returners and minions can be loaded.
# Specify a list of extra directories to search for minion modules and
# returners. These paths must be fully qualified!
#module_dirs: []
#returner_dirs: []
#states_dirs: []
#render_dirs: []
#utils_dirs: []
#
# A module provider can be statically overwritten or extended for the minion
# via the providers option, in this case the default module will be
# overwritten by the specified module. In this example the pkg module will
# be provided by the yumpkg5 module instead of the system default.
#providers:
#  pkg: yumpkg5
#
# Enable Cython modules searching and loading. (Default: False)
#cython_enable: False
#
# Specify a max size (in bytes) for modules on import. This feature is currently
# only supported on *nix operating systems and requires psutil.
# modules_max_memory: -1


#####    State Management Settings    #####
###########################################
# The state management system executes all of the state templates on the minion
# to enable more granular control of system state management. The type of
# template and serialization used for state management needs to be configured
# on the minion, the default renderer is yaml_jinja. This is a yaml file
# rendered from a jinja template, the available options are:
# yaml_jinja
# yaml_mako
# yaml_wempy
# json_jinja
# json_mako
# json_wempy
#
```

```
#renderer: yaml_jinja
#
# The failhard option tells the minions to stop immediately after the first
# failure detected in the state execution. Defaults to False.
#failhard: False
#
# Reload the modules prior to a highstate run.
#autoload_dynamic_modules: True
#
# clean_dynamic_modules keeps the dynamic modules on the minion in sync with
# the dynamic modules on the master, this means that if a dynamic module is
# not on the master it will be deleted from the minion. By default, this is
# enabled and can be disabled by changing this value to False.
#clean_dynamic_modules: True
#
# Normally, the minion is not isolated to any single environment on the master
# when running states, but the environment can be isolated on the minion side
# by statically setting it. Remember that the recommended way to manage
# environments is to isolate via the top file.
#environment: None
#
# If using the local file directory, then the state top file name needs to be
# defined, by default this is top.sls.
#state_top: top.sls
#
# Run states when the minion daemon starts. To enable, set startup_states to:
# 'highstate' -- Execute state.highstate
# 'sls' -- Read in the sls_list option and execute the named sls files
# 'top' -- Read top_file option and execute based on that file on the Master
#startup_states: ''
#
# List of states to run when the minion starts up if startup_states is 'sls':
#sls_list:
#  - edit.vim
#  - hyper
#
# Top file to execute if startup_states is 'top':
#top_file: ''

# Automatically aggregate all states that have support for mod_aggregate by
# setting to True. Or pass a list of state module names to automatically
# aggregate just those types.
#
# state_aggregate:
#   - pkg
#
#state_aggregate: False

#####     File Directory Settings    #####
##########################################
# The Salt Minion can redirect all file server operations to a local directory,
# this allows for the same state tree that is on the master to be used if
# copied completely onto the minion. This is a literal copy of the settings on
# the master but used to reference a local directory on the minion.

# Set the file client. The client defaults to looking on the master server for
# files, but can be directed to look at the local file directory setting
# defined below by setting it to "local". Setting a local file_client runs the
```

```
# minion in masterless mode.
#file_client: remote

# The file directory works on environments passed to the minion, each environment
# can have multiple root directories, the subdirectories in the multiple file
# roots cannot match, otherwise the downloaded files will not be able to be
# reliably ensured. A base environment is required to house the top file.
# Example:
# file_roots:
#   base:
#     - /srv/salt/
#   dev:
#     - /srv/salt/dev/services
#     - /srv/salt/dev/states
#   prod:
#     - /srv/salt/prod/services
#     - /srv/salt/prod/states
#
#file_roots:
#  base:
#    - /srv/salt

# By default, the Salt fileserver recurses fully into all defined environments
# to attempt to find files. To limit this behavior so that the fileserver only
# traverses directories with SLS files and special Salt directories like _modules,
# enable the option below. This might be useful for installations where a file root
# has a very large number of files and performance is negatively impacted. Default
# is False.
#fileserver_limit_traversal: False

# The hash_type is the hash to use when discovering the hash of a file in
# the local fileserver. The default is sha256 but sha224, sha384 and sha512
# are also supported.
#
# WARNING: While md5 and sha1 are also supported, do not use it due to the high chance
# of possible collisions and thus security breach.
#
# WARNING: While md5 is also supported, do not use it due to the high chance
# of possible collisions and thus security breach.
#
# Warning: Prior to changing this value, the minion should be stopped and all
# Salt caches should be cleared.
#hash_type: sha256

# The Salt pillar is searched for locally if file_client is set to local. If
# this is the case, and pillar data is defined, then the pillar_roots need to
# also be configured on the minion:
#pillar_roots:
#  base:
#    - /srv/pillar
#
#
######        Security settings       #####
###########################################
# Enable "open mode", this mode still maintains encryption, but turns off
# authentication, this is only intended for highly secure environments or for
# the situation where your keys end up in a bad state. If you run in open mode
# you do so at your own risk!
```

```
#open_mode: False

# Enable permissive access to the salt keys.  This allows you to run the
# master or minion as root, but have a non-root group be given access to
# your pki_dir.  To make the access explicit, root must belong to the group
# you've given access to. This is potentially quite insecure.
#permissive_pki_access: False

# The state_verbose and state_output settings can be used to change the way
# state system data is printed to the display. By default all data is printed.
# The state_verbose setting can be set to True or False, when set to False
# all data that has a result of True and no changes will be suppressed.
#state_verbose: True

# The state_output setting controls which results will be output full multi line
# full, terse - each state will be full/terse
# mixed - only states with errors will be full
# changes - states with changes and errors will be full
# full_id, mixed_id, changes_id and terse_id are also allowed;
# when set, the state ID will be used as name in the output
#state_output: full

# The state_output_diff setting changes whether or not the output from
# successful states is returned. Useful when even the terse output of these
# states is cluttering the logs. Set it to True to ignore them.
#state_output_diff: False

# The state_output_profile setting changes whether profile information
# will be shown for each state run.
#state_output_profile: True

# Fingerprint of the master public key to validate the identity of your Salt master
# before the initial key exchange. The master fingerprint can be found by running
# "salt-key -F master" on the Salt master.
#master_finger: ''


######         Thread settings        #####
############################################
# Disable multiprocessing support, by default when a minion receives a
# publication a new process is spawned and the command is executed therein.
#multiprocessing: True


#####         Logging settings       #####
##########################################
# The location of the minion log file
# The minion log can be sent to a regular file, local path name, or network
# location. Remote logging works best when configured to use rsyslogd(8) (e.g.:
# ``file:///dev/log``), with rsyslogd(8) configured for network logging. The URI
# format is: <file|udp|tcp>://<host|socketpath>:<port-if-required>/<log-facility>
#log_file: /var/log/salt/minion
#log_file: file:///dev/log
#log_file: udp://loghost:10514
#
#log_file: /var/log/salt/minion
#key_logfile: /var/log/salt/key
```

```
# The level of messages to send to the console.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
#
# The following log levels are considered INSECURE and may log sensitive data:
# ['garbage', 'trace', 'debug']
#
# Default: 'warning'
#log_level: warning

# The level of messages to send to the log file.
# One of 'garbage', 'trace', 'debug', info', 'warning', 'error', 'critical'.
# If using 'log_granular_levels' this must be set to the highest desired level.
# Default: 'warning'
#log_level_logfile:

# The date and time format used in log messages. Allowed date/time formatting
# can be seen here: http://docs.python.org/library/time.html#time.strftime
#log_datefmt: '%H:%M:%S'
#log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'

# The format of the console logging messages. Allowed formatting options can
# be seen here: http://docs.python.org/library/logging.html#logrecord-attributes
#
# Console log colors are specified by these additional formatters:
#
# %(colorlevel)s
# %(colorname)s
# %(colorprocess)s
# %(colormsg)s
#
# Since it is desirable to include the surrounding brackets, '[' and ']', in
# the coloring of the messages, these color formatters also include padding as
# well.  Color LogRecord attributes are only available for console logging.
#
#log_fmt_console: '%(colorlevel)s %(colormsg)s'
#log_fmt_console: '[%(levelname)-8s] %(message)s'
#
#log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'

# This can be used to control logging levels more specificically.  This
# example sets the main salt library at the 'warning' level, but sets
# 'salt.modules' to log at the 'debug' level:
#   log_granular_levels:
#     'salt': 'warning'
#     'salt.modules': 'debug'
#
#log_granular_levels: {}

# To diagnose issues with minions disconnecting or missing returns, ZeroMQ
# supports the use of monitor sockets # to log connection events. This
# feature requires ZeroMQ 4.0 or higher.
#
# To enable ZeroMQ monitor sockets, set 'zmq_monitor' to 'True' and log at a
# debug level or higher.
#
# A sample log event is as follows:
#
# [DEBUG   ] ZeroMQ event: {'endpoint': 'tcp://127.0.0.1:4505', 'event': 512,
```

```
# 'value': 27, 'description': 'EVENT_DISCONNECTED'}
#
# All events logged will include the string 'ZeroMQ event'. A connection event
# should be logged on the as the minion starts up and initially connects to the
# master. If not, check for debug log level and that the necessary version of
# ZeroMQ is installed.
#
#zmq_monitor: False

######      Module configuration      #####
###########################################
# Salt allows for modules to be passed arbitrary configuration data, any data
# passed here in valid yaml format will be passed on to the salt minion modules
# for use. It is STRONGLY recommended that a naming convention be used in which
# the module name is followed by a . and then the value. Also, all top level
# data must be applied via the yaml dict construct, some examples:
#
# You can specify that all modules should run in test mode:
#test: True
#
# A simple value for the test module:
#test.foo: foo
#
# A list for the test module:
#test.bar: [baz,quo]
#
# A dict for the test module:
#test.baz: {spam: sausage, cheese: bread}
#
#
######      Update settings          ######
###########################################
# Using the features in Esky, a salt minion can both run as a frozen app and
# be updated on the fly. These options control how the update process
# (saltutil.update()) behaves.
#
# The url for finding and downloading updates. Disabled by default.
#update_url: False
#
# The list of services to restart after a successful update. Empty by default.
#update_restart_services: []


######      Keepalive settings        ######
############################################
# ZeroMQ now includes support for configuring SO_KEEPALIVE if supported by
# the OS. If connections between the minion and the master pass through
# a state tracking device such as a firewall or VPN gateway, there is
# the risk that it could tear down the connection the master and minion
# without informing either party that their connection has been taken away.
# Enabling TCP Keepalives prevents this from happening.

# Overall state of TCP Keepalives, enable (1 or True), disable (0 or False)
# or leave to the OS defaults (-1), on Linux, typically disabled. Default True, enabled.
#tcp_keepalive: True

# How long before the first keepalive should be sent in seconds. Default 300
# to send the first keepalive after 5 minutes, OS default (-1) is typically 7200 seconds
```

```
# on Linux see /proc/sys/net/ipv4/tcp_keepalive_time.
#tcp_keepalive_idle: 300

# How many lost probes are needed to consider the connection lost. Default -1
# to use OS defaults, typically 9 on Linux, see /proc/sys/net/ipv4/tcp_keepalive_probes.
#tcp_keepalive_cnt: -1

# How often, in seconds, to send keepalives after the first one. Default -1 to
# use OS defaults, typically 75 seconds on Linux, see
# /proc/sys/net/ipv4/tcp_keepalive_intvl.
#tcp_keepalive_intvl: -1


######   Windows Software settings    ######
############################################
# Location of the repository cache file on the master:
#win_repo_cachefile: 'salt://win/repo/winrepo.p'


######      Returner  settings        ######
############################################
# Which returner(s) will be used for minion's result:
#return: mysql
```

3.5 Minion Blackout Configuration

New in version 2016.3.0.

Salt supports minion blackouts. When a minion is in blackout mode, all remote execution commands are disabled. This allows production minions to be put ``on hold'', eliminating the risk of an untimely configuration change.

Minion blackouts are configured via a special pillar key, `minion_blackout`. If this key is set to `True`, then the minion will reject all incoming commands, except for `saltutil.refresh_pillar`. (The exception is important, so minions can be brought out of blackout mode)

Salt also supports an explicit whitelist of additional functions that will be allowed during blackout. This is configured with the special pillar key `minion_blackout_whitelist`, which is formed as a list:

```
minion_blackout_whitelist:
  - test.ping
  - pillar.get
```

3.6 Access Control System

New in version 0.10.4.

Salt maintains a standard system used to open granular control to non administrative users to execute Salt commands. The access control system has been applied to all systems used to configure access to non administrative control interfaces in Salt.

These interfaces include, the `peer` system, the `external auth` system and the `publisher acl` system.

The access control system mandated a standard configuration syntax used in all of the three aforementioned systems. While this adds functionality to the configuration in 0.10.4, it does not negate the old configuration.

Now specific functions can be opened up to specific minions from specific users in the case of external auth and publisher ACLs, and for specific minions in the case of the peer system.

3.6.1 Publisher ACL system

The salt publisher ACL system is a means to allow system users other than root to have access to execute select salt commands on minions from the master.

The publisher ACL system is configured in the master configuration file via the `publisher_acl` configuration option. Under the `publisher_acl` configuration option the users open to send commands are specified and then a list of the minion functions which will be made available to specified user. Both users and functions could be specified by exact match, shell glob or regular expression. This configuration is much like the *external_auth* configuration:

```
publisher_acl:
  # Allow thatch to execute anything.
  thatch:
    - .*
  # Allow fred to use test and pkg, but only on "web*" minions.
  fred:
    - web*:
      - test.*
      - pkg.*
  # Allow admin and managers to use saltutil module functions
  admin|manager_.*:
    - saltutil.*
  # Allow users to use only my_mod functions on "web*" minions with specific arguments.
  user_.*:
    - web*:
      - 'my_mod.*':
          args:
            - 'a.*'
            - 'b.*'
          kwargs:
            'kwa': 'kwa.*'
            'kwb': 'kwb'
```

Permission Issues

Directories required for `publisher_acl` must be modified to be readable by the users specified:

```
chmod 755 /var/cache/salt /var/cache/salt/master /var/cache/salt/master/jobs /var/run/
↪salt /var/run/salt/master
```

Note: In addition to the changes above you will also need to modify the permissions of /var/log/salt and the existing log file to be writable by the user(s) which will be running the commands. If you do not wish to do this then you must disable logging or Salt will generate errors as it cannot write to the logs as the system users.

If you are upgrading from earlier versions of salt you must also remove any existing user keys and re-start the Salt master:

```
rm /var/cache/salt/.*key
service salt-master restart
```

Whitelist and Blacklist

Salt's authentication systems can be configured by specifying what is allowed using a whitelist, or by specifying what is disallowed using a blacklist. If you specify a whitelist, only specified operations are allowed. If you specify a blacklist, all operations are allowed except those that are blacklisted.

See *publisher_acl* and *publisher_acl_blacklist*.

3.6.2 External Authentication System

Salt's External Authentication System (eAuth) allows for Salt to pass through command authorization to any external authentication system, such as PAM or LDAP.

Note: eAuth using the PAM external auth system requires salt-master to be run as root as this system needs root access to check authentication.

External Authentication System Configuration

The external authentication system allows for specific users to be granted access to execute specific functions on specific minions. Access is configured in the master configuration file and uses the *access control system*:

```
external_auth:
  pam:
    thatch:
      - 'web*':
        - test.*
        - network.*
    steve|admin.*:
      - .*
```

The above configuration allows the user `thatch` to execute functions in the test and network modules on the minions that match the web* target. User `steve` and the users whose logins start with `admin`, are granted unrestricted access to minion commands.

Salt respects the current PAM configuration in place, and uses the `login' service to authenticate.

Note: The PAM module does not allow authenticating as `root`.

Note: state.sls and state.highstate will return ``Failed to authenticate!'' if the request timeout is reached. Use -t flag to increase the timeout

---

To allow access to *wheel modules* or *runner modules* the following `@` syntax must be used:

```
external_auth:
  pam:
    thatch:
      - '@wheel'   # to allow access to all wheel modules
      - '@runner'  # to allow access to all runner modules
      - '@jobs'    # to allow access to the jobs runner and/or wheel module
```

Note: The runner/wheel markup is different, since there are no minions to scope the acl to.

Note: Globs will not match wheel or runners! They must be explicitly allowed with @wheel or @runner.

Warning: All users that have external authentication privileges are allowed to run *saltutil.findjob*. Be aware that this could inadvertently expose some data such as minion IDs.

Matching syntax

The structure of the `external_auth` dictionary can take the following shapes. User and function matches are exact matches, shell glob patterns or regular expressions; minion matches are compound targets.

By user:

```
external_auth:
  <eauth backend>:
    <user or group%>:
      - <regex to match function>
```

By user, by minion:

```
external_auth:
  <eauth backend>:
    <user or group%>:
      <minion compound target>:
        - <regex to match function>
```

By user, by runner/wheel:

```
external_auth:
  <eauth backend>:
    <user or group%>:
      <@runner or @wheel>:
        - <regex to match function>
```

By user, by runner+wheel module:

```
external_auth:
  <eauth backend>:
    <user or group%>:
      <@module_name>:
        - <regex to match function without module_name>
```

Groups

To apply permissions to a group of users in an external authentication system, append a `%` to the ID:

```
external_auth:
  pam:
    admins%:
```

```
      - '*':
        - 'pkg.*'
```

Limiting by function arguments

Positional arguments or keyword arguments to functions can also be whitelisted.

New in version 2016.3.0.

```
external_auth:
  pam:
    my_user:
      - '*':
        - 'my_mod.*':
            args:
              - 'a.*'
              - 'b.*'
            kwargs:
              'kwa': 'kwa.*'
              'kwb': 'kwb'
      - '@runner':
        - 'runner_mod.*':
            args:
            - 'a.*'
            - 'b.*'
            kwargs:
              'kwa': 'kwa.*'
              'kwb': 'kwb'
```

The rules:

1. The arguments values are matched as regexp.
2. If arguments restrictions are specified the only matched are allowed.
3. If an argument isn't specified any value is allowed.
4. To skip an arg use ``everything'' regexp `.*`. I.e. if `arg0` and `arg2` should be limited but `arg1` and other arguments could have any value use:

   ```
   args:
     - 'value0'
     - '.*'
     - 'value2'
   ```

Usage

The external authentication system can then be used from the command-line by any user on the same system as the master with the `-a` option:

```
$ salt -a pam web\* test.ping
```

The system will ask the user for the credentials required by the authentication system and then publish the command.

Tokens

With external authentication alone, the authentication credentials will be required with every call to Salt. This can be alleviated with Salt tokens.

Tokens are short term authorizations and can be easily created by just adding a `-T` option when authenticating:

```
$ salt -T -a pam web\* test.ping
```

Now a token will be created that has an expiration of 12 hours (by default). This token is stored in a file named `salt_token` in the active user's home directory.

Once the token is created, it is sent with all subsequent communications. User authentication does not need to be entered again until the token expires.

Token expiration time can be set in the Salt master config file.

LDAP and Active Directory

Note: LDAP usage requires that you have installed python-ldap.

Salt supports both user and group authentication for LDAP (and Active Directory accessed via its LDAP interface)

OpenLDAP and similar systems

LDAP configuration happens in the Salt master configuration file.

Server configuration values and their defaults:

```
# Server to auth against
auth.ldap.server: localhost

# Port to connect via
auth.ldap.port: 389

# Use TLS when connecting
auth.ldap.tls: False

# LDAP scope level, almost always 2
auth.ldap.scope: 2

# Server specified in URI format
auth.ldap.uri: ''    # Overrides .ldap.server, .ldap.port, .ldap.tls above

# Verify server's TLS certificate
auth.ldap.no_verify: False

# Bind to LDAP anonymously to determine group membership
# Active Directory does not allow anonymous binds without special configuration
# In addition, if auth.ldap.anonymous is True, empty bind passwords are not permitted.
auth.ldap.anonymous: False

# FOR TESTING ONLY, this is a VERY insecure setting.
# If this is True, the LDAP bind password will be ignored and
# access will be determined by group membership alone with
```

```
# the group memberships being retrieved via anonymous bind
auth.ldap.auth_by_group_membership_only: False

# Require authenticating user to be part of this Organizational Unit
# This can be blank if your LDAP schema does not use this kind of OU
auth.ldap.groupou: 'Groups'

# Object Class for groups.  An LDAP search will be done to find all groups of this
# class to which the authenticating user belongs.
auth.ldap.groupclass: 'posixGroup'

# Unique ID attribute name for the user
auth.ldap.accountattributename: 'memberUid'

# These are only for Active Directory
auth.ldap.activedirectory: False
auth.ldap.persontype: 'person'

auth.ldap.minion_stripdomains: []

# Redhat Identity Policy Audit
auth.ldap.freeipa: False
```

Authenticating to the LDAP Server

There are two phases to LDAP authentication. First, Salt authenticates to search for a users' Distinguished Name and group membership. The user it authenticates as in this phase is often a special LDAP system user with read-only access to the LDAP directory. After Salt searches the directory to determine the actual user's DN and groups, it re-authenticates as the user running the Salt commands.

If you are already aware of the structure of your DNs and permissions in your LDAP store are set such that users can look up their own group memberships, then the first and second users can be the same. To tell Salt this is the case, omit the `auth.ldap.bindpw` parameter. Note this is not the same thing as using an anonymous bind. Most LDAP servers will not permit anonymous bind, and as mentioned above, if *auth.ldap.anonymous* is False you cannot use an empty password.

You can template the `binddn` like this:

```
auth.ldap.basedn: dc=saltstack,dc=com
auth.ldap.binddn: uid={{ username }},cn=users,cn=accounts,dc=saltstack,dc=com
```

Salt will use the password entered on the salt command line in place of the bindpw.

To use two separate users, specify the LDAP lookup user in the binddn directive, and include a bindpw like so

```
auth.ldap.binddn: uid=ldaplookup,cn=sysaccounts,cn=etc,dc=saltstack,dc=com
auth.ldap.bindpw: mypassword
```

As mentioned before, Salt uses a filter to find the DN associated with a user. Salt substitutes the `{{ username }}` value for the username when querying LDAP

```
auth.ldap.filter: uid={{ username }}
```

Determining Group Memberships (OpenLDAP / non-Active Directory)

For OpenLDAP, to determine group membership, one can specify an OU that contains group data. This is prepended to the basedn to create a search path. Then the results are filtered against `auth.ldap.groupclass`, default `posixGroup`, and the account's `name' attribute, `memberUid` by default.

```
auth.ldap.groupou: Groups
```

Note that as of 2017.7, auth.ldap.groupclass can refer to either a groupclass or an objectClass. For some LDAP servers (notably OpenLDAP without the `memberOf` overlay enabled) to determine group membership we need to know both the `objectClass` and the `memberUid` attributes. Usually for these servers you will want a `auth.ldap.groupclass` of `posixGroup` and an `auth.ldap.groupattribute` of `memberUid`.

LDAP servers with the `memberOf` overlay will have entries similar to `auth.ldap.groupclass: person` and `auth.ldap.groupattribute: memberOf`.

When using the `ldap('DC=domain,DC=com')` eauth operator, sometimes the records returned from LDAP or Active Directory have fully-qualified domain names attached, while minion IDs instead are simple hostnames. The parameter below allows the administrator to strip off a certain set of domain names so the hostnames looked up in the directory service can match the minion IDs.

```
auth.ldap.minion_stripdomains: ['.external.bigcorp.com', '.internal.bigcorp.com']
```

Determining Group Memberships (Active Directory)

Active Directory handles group membership differently, and does not utilize the `groupou` configuration variable. AD needs the following options in the master config:

```
auth.ldap.activedirectory: True
auth.ldap.filter: sAMAccountName={{username}}
auth.ldap.accountattributename: sAMAccountName
auth.ldap.groupclass: group
auth.ldap.persontype: person
```

To determine group membership in AD, the username and password that is entered when LDAP is requested as the eAuth mechanism on the command line is used to bind to AD's LDAP interface. If this fails, then it doesn't matter what groups the user belongs to, he or she is denied access. Next, the `distinguishedName` of the user is looked up with the following LDAP search:

```
(&(<value of auth.ldap.accountattributename>={{username}})
  (objectClass=<value of auth.ldap.persontype>)
)
```

This should return a distinguishedName that we can use to filter for group membership. Then the following LDAP query is executed:

```
(&(member=<distinguishedName from search above>)
  (objectClass=<value of auth.ldap.groupclass>)
)
```

```
external_auth:
  ldap:
    test_ldap_user:
        - '*':
            - test.ping
```

To configure a LDAP group, append a % to the ID:

```
external_auth:
  ldap:
    test_ldap_group%:
      - '*':
        - test.echo
```

In addition, if there are a set of computers in the directory service that should be part of the eAuth definition, they can be specified like this:

```
external_auth:
  ldap:
    test_ldap_group%:
      - ldap('DC=corp,DC=example,DC=com'):
        - test.echo
```

The string inside `ldap()` above is any valid LDAP/AD tree limiter. `OU=` in particular is permitted as long as it would return a list of computer objects.

3.6.3 Peer Communication

Salt 0.9.0 introduced the capability for Salt minions to publish commands. The intent of this feature is not for Salt minions to act as independent brokers one with another, but to allow Salt minions to pass commands to each other.

In Salt 0.10.0 the ability to execute runners from the master was added. This allows for the master to return collective data from runners back to the minions via the peer interface.

The peer interface is configured through two options in the master configuration file. For minions to send commands from the master the `peer` configuration is used. To allow for minions to execute runners from the master the `peer_run` configuration is used.

Since this presents a viable security risk by allowing minions access to the master publisher the capability is turned off by default. The minions can be allowed access to the master publisher on a per minion basis based on regular expressions. Minions with specific ids can be allowed access to certain Salt modules and functions.

Peer Configuration

The configuration is done under the `peer` setting in the Salt master configuration file, here are a number of configuration possibilities.

The simplest approach is to enable all communication for all minions, this is only recommended for very secure environments.

```
peer:
  .*:
    - .*
```

This configuration will allow minions with IDs ending in example.com access to the test, ps, and pkg module functions.

```
peer:
  .*example.com:
    - test.*
    - ps.*
    - pkg.*
```

The configuration logic is simple, a regular expression is passed for matching minion ids, and then a list of expressions matching minion functions is associated with the named minion. For instance, this configuration will also allow minions ending with foo.org access to the publisher.

```
peer:
  .*example.com:
    - test.*
    - ps.*
    - pkg.*
  .*foo.org:
    - test.*
    - ps.*
    - pkg.*
```

Note: Functions are matched using regular expressions.

Peer Runner Communication

Configuration to allow minions to execute runners from the master is done via the `peer_run` option on the master. The `peer_run` configuration follows the same logic as the `peer` option. The only difference is that access is granted to runner modules.

To open up access to all minions to all runners:

```
peer_run:
  .*:
    - .*
```

This configuration will allow minions with IDs ending in example.com access to the manage and jobs runner functions.

```
peer_run:
  .*example.com:
    - manage.*
    - jobs.*
```

Note: Functions are matched using regular expressions.

Using Peer Communication

The publish module was created to manage peer communication. The publish module comes with a number of functions to execute peer communication in different ways. Currently there are three functions in the publish module. These examples will show how to test the peer system via the salt-call command.

To execute test.ping on all minions:

```
# salt-call publish.publish \* test.ping
```

To execute the manage.up runner:

```
# salt-call publish.runner manage.up
```

To match minions using other matchers, use `tgt_type`:

```
# salt-call publish.publish 'webserv* and not G@os:Ubuntu' test.ping tgt_type='compound
↪'
```

Note: In pre-2017.7.0 releases, use `expr_form` instead of `tgt_type`.

3.6.4 When to Use Each Authentication System

`publisher_acl` is useful for allowing local system users to run Salt commands without giving them root access. If you can log into the Salt master directly, then `publisher_acl` allows you to use Salt without root privileges. If the local system is configured to authenticate against a remote system, like LDAP or Active Directory, then `publisher_acl` will interact with the remote system transparently.

`external_auth` is useful for `salt-api` or for making your own scripts that use Salt's Python API. It can be used at the CLI (with the `-a` flag) but it is more cumbersome as there are more steps involved. The only time it is useful at the CLI is when the local system is *not* configured to authenticate against an external service *but* you still want Salt to authenticate against an external service.

3.6.5 Examples

The access controls are manifested using matchers in these configurations:

```
publisher_acl:
  fred:
    - web\*:
      - pkg.list_pkgs
      - test.*
      - apache.*
```

In the above example, fred is able to send commands only to minions which match the specified glob target. This can be expanded to include other functions for other minions based on standard targets (all matchers are supported except the compound one).

```
external_auth:
  pam:
    dave:
      - test.ping
      - mongo\*:
        - network.*
      - log\*:
        - network.*
        - pkg.*
      - 'G@os:RedHat':
        - kmod.*
    steve:
      - .*
```

The above allows for all minions to be hit by test.ping by dave, and adds a few functions that dave can execute on other minions. It also allows steve unrestricted access to salt commands.

Note: Functions are matched using regular expressions.

3.7 Job Management

New in version 0.9.7.

Since Salt executes jobs running on many systems, Salt needs to be able to manage jobs running on many systems.

3.7.1 The Minion proc System

Salt Minions maintain a *proc* directory in the Salt `cachedir`. The *proc* directory maintains files named after the executed job ID. These files contain the information about the current running jobs on the minion and allow for jobs to be looked up. This is located in the *proc* directory under the cachedir, with a default configuration it is under `/var/cache/salt/proc`.

3.7.2 Functions in the saltutil Module

Salt 0.9.7 introduced a few new functions to the *saltutil* module for managing jobs. These functions are:

1. `running` Returns the data of all running jobs that are found in the *proc* directory.
2. `find_job` Returns specific data about a certain job based on job id.
3. `signal_job` Allows for a given jid to be sent a signal.
4. `term_job` Sends a termination signal (SIGTERM, 15) to the process controlling the specified job.
5. `kill_job` Sends a kill signal (SIGKILL, 9) to the process controlling the specified job.

These functions make up the core of the back end used to manage jobs at the minion level.

3.7.3 The jobs Runner

A convenience runner front end and reporting system has been added as well. The jobs runner contains functions to make viewing data easier and cleaner.

The jobs runner contains a number of functions...

active

The active function runs saltutil.running on all minions and formats the return data about all running jobs in a much more usable and compact format. The active function will also compare jobs that have returned and jobs that are still running, making it easier to see what systems have completed a job and what systems are still being waited on.

```
# salt-run jobs.active
```

lookup_jid

When jobs are executed the return data is sent back to the master and cached. By default it is cached for 24 hours, but this can be configured via the `keep_jobs` option in the master configuration. Using the lookup_jid runner will display the same return data that the initial job invocation with the salt command would display.

```
# salt-run jobs.lookup_jid <job id number>
```

list_jobs

Before finding a historic job, it may be required to find the job id. `list_jobs` will parse the cached execution data and display all of the job data for jobs that have already, or partially returned.

```
# salt-run jobs.list_jobs
```

3.7.4 Scheduling Jobs

Salt's scheduling system allows incremental executions on minions or the master. The schedule system exposes the execution of any execution function on minions or any runner on the master.

Scheduling can be enabled by multiple methods:

- `schedule` option in either the master or minion config files. These require the master or minion application to be restarted in order for the schedule to be implemented.
- Minion pillar data. Schedule is implemented by refreshing the minion's pillar data, for example by using `saltutil.refresh_pillar`.
- The *schedule state* or *schedule module*

Note: The scheduler executes different functions on the master and minions. When running on the master the functions reference runner functions, when running on the minion the functions specify execution functions.

A scheduled run has no output on the minion unless the config is set to info level or higher. Refer to `minion-logging-settings`.

States are executed on the minion, as all states are. You can pass positional arguments and provide a YAML dict of named arguments.

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    args:
      - httpd
    kwargs:
      test: True
```

This will schedule the command: `state.sls httpd test=True` every 3600 seconds (every hour).

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    args:
      - httpd
    kwargs:
      test: True
    splay: 15
```

This will schedule the command: `state.sls httpd test=True` every 3600 seconds (every hour) splaying the time between 0 and 15 seconds.

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    args:
      - httpd
    kwargs:
      test: True
    splay:
      start: 10
      end: 15
```

This will schedule the command: `state.sls httpd test=True` every 3600 seconds (every hour) splaying the time between 10 and 15 seconds.

Schedule by Date and Time

New in version 2014.7.0.

Frequency of jobs can also be specified using date strings supported by the Python `dateutil` library. This requires the Python `dateutil` library to be installed.

```
schedule:
  job1:
    function: state.sls
    args:
      - httpd
    kwargs:
      test: True
    when: 5:00pm
```

This will schedule the command: `state.sls httpd test=True` at 5:00 PM minion localtime.

```
schedule:
  job1:
    function: state.sls
    args:
      - httpd
    kwargs:
      test: True
    when:
      - Monday 5:00pm
      - Tuesday 3:00pm
      - Wednesday 5:00pm
      - Thursday 3:00pm
      - Friday 5:00pm
```

This will schedule the command: `state.sls httpd test=True` at 5:00 PM on Monday, Wednesday and Friday, and 3:00 PM on Tuesday and Thursday.

```
schedule:
  job1:
    function: state.sls
    args:
      - httpd
    kwargs:
      test: True
```

```
    when:
      - 'tea time'
```

```
whens:
  tea time: 1:40pm
  deployment time: Friday 5:00pm
```

The Salt scheduler also allows custom phrases to be used for the *when* parameter. These *whens* can be stored as either pillar values or grain values.

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    args:
      - httpd
    kwargs:
      test: True
    range:
      start: 8:00am
      end: 5:00pm
```

This will schedule the command: `state.sls httpd test=True` every 3600 seconds (every hour) between the hours of 8:00 AM and 5:00 PM. The range parameter must be a dictionary with the date strings using the `dateutil` format.

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    args:
      - httpd
    kwargs:
      test: True
    range:
      invert: True
      start: 8:00am
      end: 5:00pm
```

Using the invert option for range, this will schedule the command `state.sls httpd test=True` every 3600 seconds (every hour) until the current time is between the hours of 8:00 AM and 5:00 PM. The range parameter must be a dictionary with the date strings using the `dateutil` format.

```
schedule:
  job1:
    function: pkg.install
    kwargs:
      pkgs: [{'bar': '>1.2.3'}]
      refresh: true
    once: '2016-01-07T14:30:00'
```

This will schedule the function `pkg.install` to be executed once at the specified time. The schedule entry `job1` will not be removed after the job completes, therefore use `schedule.delete` to manually remove it afterwards.

The default date format is ISO 8601 but can be overridden by also specifying the `once_fmt` option, like this:

```
schedule:
  job1:
    function: test.ping
    once: 2015-04-22T20:21:00
    once_fmt: '%Y-%m-%dT%H:%M:%S'
```

Maximum Parallel Jobs Running

New in version 2014.7.0.

The scheduler also supports ensuring that there are no more than N copies of a particular routine running. Use this for jobs that may be long-running and could step on each other or pile up in case of infrastructure outage.

The default for `maxrunning` is 1.

```
schedule:
  long_running_job:
    function: big_file_transfer
    jid_include: True
    maxrunning: 1
```

Cron-like Schedule

New in version 2014.7.0.

```
schedule:
  job1:
    function: state.sls
    cron: '*/15 * * * *'
    args:
      - httpd
    kwargs:
      test: True
```

The scheduler also supports scheduling jobs using a cron like format. This requires the Python `croniter` library.

Job Data Return

New in version 2015.5.0.

By default, data about jobs runs from the Salt scheduler is returned to the master. Setting the `return_job` parameter to False will prevent the data from being sent back to the Salt master.

```
schedule:
  job1:
    function: scheduled_job_function
    return_job: False
```

Job Metadata

New in version 2015.5.0.

It can be useful to include specific data to differentiate a job from other jobs. Using the metadata parameter special values can be associated with a scheduled job. These values are not used in the execution of the job, but can be used

to search for specific jobs later if combined with the return_job parameter. The metadata parameter must be specified as a dictionary, othewise it will be ignored.

```
schedule:
  job1:
    function: scheduled_job_function
    metadata:
      foo: bar
```

Run on Start

New in version 2015.5.0.

By default, any job scheduled based on the startup time of the minion will run the scheduled job when the minion starts up. Sometimes this is not the desired situation. Using the run_on_start parameter set to False will cause the scheduler to skip this first run and wait until the next scheduled run:

```
schedule:
  job1:
    function: state.sls
    seconds: 3600
    run_on_start: False
    args:
      - httpd
    kwargs:
      test: True
```

Until and After

New in version 2015.8.0.

```
schedule:
  job1:
    function: state.sls
    seconds: 15
    until: '12/31/2015 11:59pm'
    args:
      - httpd
    kwargs:
      test: True
```

Using the until argument, the Salt scheduler allows you to specify an end time for a scheduled job. If this argument is specified, jobs will not run once the specified time has passed. Time should be specified in a format supported by the dateutil library. This requires the Python dateutil library to be installed.

New in version 2015.8.0.

```
schedule:
  job1:
    function: state.sls
    seconds: 15
    after: '12/31/2015 11:59pm'
    args:
      - httpd
    kwargs:
      test: True
```

Using the after argument, the Salt scheduler allows you to specify an start time for a scheduled job. If this argument is specified, jobs will not run until the specified time has passed. Time should be specified in a format supported by the `dateutil` library. This requires the Python `dateutil` library to be installed.

Scheduling States

```
schedule:
  log-loadavg:
    function: cmd.run
    seconds: 3660
    args:
      - 'logger -t salt < /proc/loadavg'
    kwargs:
      stateful: False
      shell: /bin/sh
```

Scheduling Highstates

To set up a highstate to run on a minion every 60 minutes set this in the minion config or pillar:

```
schedule:
  highstate:
    function: state.highstate
    minutes: 60
```

Time intervals can be specified as seconds, minutes, hours, or days.

Scheduling Runners

Runner executions can also be specified on the master within the master configuration file:

```
schedule:
  run_my_orch:
    function: state.orchestrate
    hours: 6
    splay: 600
    args:
      - orchestration.my_orch
```

The above configuration is analogous to running `salt-run state.orch orchestration.my_orch` every 6 hours.

Scheduler With Returner

The scheduler is also useful for tasks like gathering monitoring data about a minion, this schedule option will gather status data and send it to a MySQL returner database:

```
schedule:
  uptime:
    function: status.uptime
    seconds: 60
    returner: mysql
  meminfo:
```

```
    function: status.meminfo
    minutes: 5
    returner: mysql
```

Since specifying the returner repeatedly can be tiresome, the `schedule_returner` option is available to specify one or a list of global returners to be used by the minions when scheduling.

3.8 Managing the Job Cache

The Salt Master maintains a job cache of all job executions which can be queried via the jobs runner. This job cache is called the Default Job Cache.

3.8.1 Default Job Cache

A number of options are available when configuring the job cache. The default caching system uses local storage on the Salt Master and can be found in the job cache directory (on Linux systems this is typically `/var/cache/salt/master/jobs`). The default caching system is suitable for most deployments as it does not typically require any further configuration or management.

The default job cache is a temporary cache and jobs will be stored for 24 hours. If the default cache needs to store jobs for a different period the time can be easily adjusted by changing the `keep_jobs` parameter in the Salt Master configuration file. The value passed in is measured via hours:

```
keep_jobs: 24
```

Reducing the Size of the Default Job Cache

The Default Job Cache can sometimes be a burden on larger deployments (over 5000 minions). Disabling the job cache will make previously executed jobs unavailable to the jobs system and is not generally recommended. Normally it is wise to make sure the master has access to a faster IO system or a tmpfs is mounted to the jobs dir.

However, you can disable the *`job_cache`* by setting it to `False` in the Salt Master configuration file. Setting this value to `False` means that the Salt Master will no longer cache minion returns, but a JID directory and `jid` file for each job will still be created. This JID directory is necessary for checking for and preventing JID collisions.

The default location for the job cache is in the `/var/cache/salt/master/jobs/` directory.

Setting the *`job_cache`* to `False` in addition to setting the *`keep_jobs`* option to a smaller value, such as `1`, in the Salt Master configuration file will reduce the size of the Default Job Cache, and thus the burden on the Salt Master.

Note: Changing the `keep_jobs` option sets the number of hours to keep old job information and defaults to `24` hours. Do not set this value to `0` when trying to make the cache cleaner run more frequently, as this means the cache cleaner will never run.

3.8.2 Additional Job Cache Options

Many deployments may wish to use an external database to maintain a long term register of executed jobs. Salt comes with two main mechanisms to do this, the master job cache and the external job cache.

See *Storing Job Results in an External System*.

3.9 Storing Job Results in an External System

After a job executes, job results are returned to the Salt Master by each Salt Minion. These results are stored in the *Default Job Cache*.

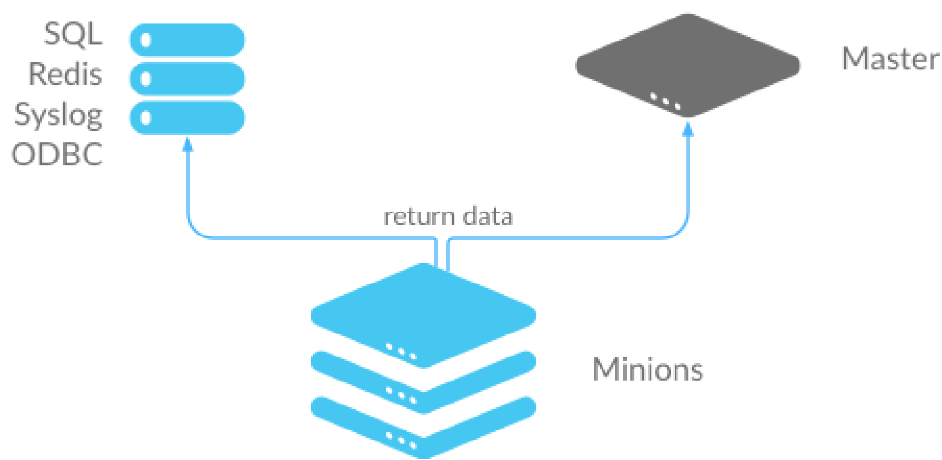
In addition to the Default Job Cache, Salt provides two additional mechanisms to send job results to other systems (databases, local syslog, and others):

- External Job Cache
- Master Job Cache

The major difference between these two mechanism is from where results are returned (from the Salt Master or Salt Minion). Configuring either of these options will also make the `Jobs Runner functions` to automatically query the remote stores for information.

3.9.1 External Job Cache - Minion-Side Returner

When an External Job Cache is configured, data is returned to the Default Job Cache on the Salt Master like usual, and then results are also sent to an External Job Cache using a Salt returner module running on the Salt Minion.

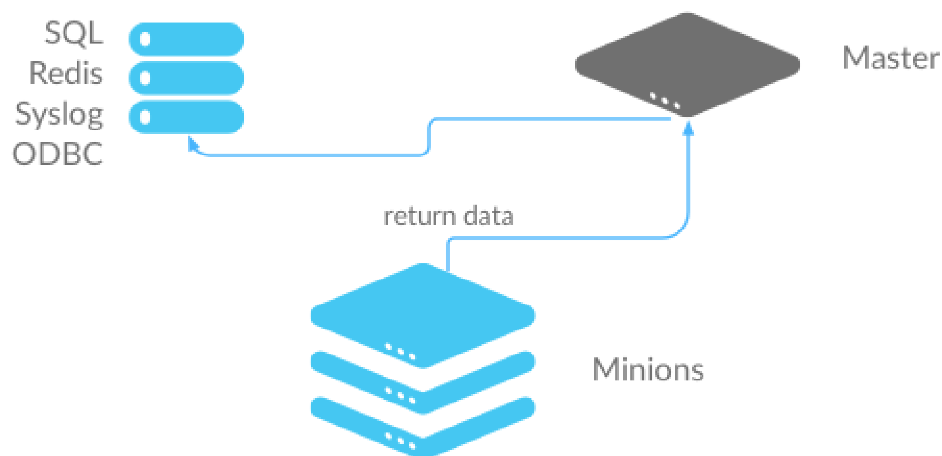


- Advantages: Data is stored without placing additional load on the Salt Master.
- Disadvantages: Each Salt Minion connects to the external job cache, which can result in a large number of connections. Also requires additional configuration to get returner module settings on all Salt Minions.

3.9.2 Master Job Cache - Master-Side Returner

New in version 2014.7.0.

Instead of configuring an External Job Cache on each Salt Minion, you can configure the Master Job Cache to send job results from the Salt Master instead. In this configuration, Salt Minions send data to the Default Job Cache as usual, and then the Salt Master sends the data to the external system using a Salt returner module running on the Salt Master.



- Advantages: A single connection is required to the external system. This is preferred for databases and similar systems.
- Disadvantages: Places additional load on your Salt Master.

3.9.3 Configure an External or Master Job Cache

Step 1: Understand Salt Returners

Before you configure a job cache, it is essential to understand Salt returner modules (``returners''). Returners are pluggable Salt Modules that take the data returned by jobs, and then perform any necessary steps to send the data to an external system. For example, a returner might establish a connection, authenticate, and then format and transfer data.

The Salt Returner system provides the core functionality used by the External and Master Job Cache systems, and the same returners are used by both systems.

Salt currently provides many different returners that let you connect to a wide variety of systems. A complete list is available at *all Salt returners*. Each returner is configured differently, so make sure you read and follow the instructions linked from that page.

For example, the MySQL returner requires:

- A database created using provided schema (structure is available at `MySQL returner`)
- A user created with privileges to the database
- Optional SSL configuration

A simpler returner, such as Slack or HipChat, requires:

- An API key/version
- The target channel/room
- The username that should be used to send the message

Step 2: Configure the Returner

After you understand the configuration and have the external system ready, the configuration requirements must be declared.

External Job Cache

The returner configuration settings can be declared in the Salt Minion configuration file, the Minion's pillar data, or the Minion's grains.

If `external_job_cache` configuration settings are specified in more than one place, the options are retrieved in the following order. The first configuration location that is found is the one that will be used.

- Minion configuration file
- Minion's grains
- Minion's pillar data

Master Job Cache

The returner configuration settings for the Master Job Cache should be declared in the Salt Master's configuration file.

Configuration File Examples

MySQL requires:

```
mysql.host: 'salt'
mysql.user: 'salt'
mysql.pass: 'salt'
mysql.db: 'salt'
mysql.port: 3306
```

Slack requires:

```
slack.channel: 'channel'
slack.api_key: 'key'
slack.from_name: 'name'
```

After you have configured the returner and added settings to the configuration file, you can enable the External or Master Job Cache.

Step 3: Enable the External or Master Job Cache

Configuration is a single line that specifies an already-configured returner to use to send all job data to an external system.

External Job Cache

To enable a returner as the External Job Cache (Minion-side), add the following line to the Salt Master configuration file:

```
ext_job_cache: <returner>
```

For example:

```
ext_job_cache: mysql
```

Note: When configuring an External Job Cache (Minion-side), the returner settings are added to the Minion configuration file, but the External Job Cache setting is configured in the Master configuration file.

Master Job Cache

To enable a returner as a Master Job Cache (Master-side), add the following line to the Salt Master configuration file:

```
master_job_cache: <returner>
```

For example:

```
master_job_cache: mysql
```

Verify that the returner configuration settings are in the Master configuration file, and be sure to restart the salt-master service after you make configuration changes. (`service salt-master restart`).

3.10 Logging

The salt project tries to get the logging to work for you and help us solve any issues you might find along the way.

If you want to get some more information on the nitty-gritty of salt's logging system, please head over to the *logging development document*, if all you're after is salt's logging configurations, please continue reading.

3.10.1 Log Levels

The log levels are ordered numerically such that setting the log level to a specific level will record all log statements at that level and higher. For example, setting `log_level: error` will log statements at `error`, `critical`, and `quiet` levels, although nothing *should* be logged at `quiet` level.

Most of the logging levels are defined by default in Python's logging library and can be found in the official Python documentation. Salt uses some more levels in addition to the standard levels. All levels available in salt are shown in the table below.

Note: Python dependencies used by salt may define and use additional logging levels. For example, the Python 2 version of the `multiprocessing` standard Python library uses the levels `subwarning`, 25 and `subdebug`, 5.

Level	Numeric value	Description
quiet	1000	Nothing should be logged at this level
critical	50	Critical errors
error	40	Errors
warning	30	Warnings
info	20	Normal log information
profile	15	Profiling information on salt performance
debug	10	Information useful for debugging both salt implementations and salt code
trace	5	More detailed code debugging information
garbage	1	Even more debugging information
all	0	Everything

3.10.2 Available Configuration Settings

log_file

The log records can be sent to a regular file, local path name, or network location. Remote logging works best when configured to use rsyslogd(8) (e.g.: `file:///dev/log`), with rsyslogd(8) configured for network logging. The format for remote addresses is:

```
<file|udp|tcp>://<host|socketpath>:<port-if-required>/<log-facility>
```

Where `log-facility` is the symbolic name of a syslog facility as defined in the `SysLogHandler documentation`. It defaults to `LOG_USER`.

Default: Dependent of the binary being executed, for example, for `salt-master`, `/var/log/salt/master`.

Examples:

```
log_file: /var/log/salt/master
```

```
log_file: /var/log/salt/minion
```

```
log_file: file:///dev/log
```

```
log_file: file:///dev/log/LOG_DAEMON
```

```
log_file: udp://loghost:10514
```

log_level

Default: `warning`

The level of log record messages to send to the console. One of `all`, `garbage`, `trace`, `debug`, `profile`, `info`, `warning`, `error`, `critical`, `quiet`.

```
log_level: warning
```

Note: Add `log_level: quiet` in salt configuration file to completely disable logging. In case of running salt in command line use `--log-level=quiet` instead.

log_level_logfile

Default: `info`

The level of messages to send to the log file. One of `all`, `garbage`, `trace`, `debug`, `profile`, `info`, `warning`, `error`, `critical`, `quiet`.

```
log_level_logfile: warning
```

log_datefmt

Default: %H:%M:%S

The date and time format used in console log messages. Allowed date/time formatting matches those used in time.strftime().

```
log_datefmt: '%H:%M:%S'
```

log_datefmt_logfile

Default: %Y-%m-%d %H:%M:%S

The date and time format used in log file messages. Allowed date/time formatting matches those used in time.strftime().

```
log_datefmt_logfile: '%Y-%m-%d %H:%M:%S'
```

log_fmt_console

Default: [%(levelname)-8s] %(message)s

The format of the console logging messages. All standard python logging LogRecord attributes can be used. Salt also provides these custom LogRecord attributes to colorize console log output:

```
'%(colorlevel)s'   # log level name colorized by level
'%(colorname)s'    # colorized module name
'%(colorprocess)s' # colorized process number
'%(colormsg)s'     # log message colorized by level
```

Note: The %(colorlevel)s, %(colorname)s, and %(colorprocess) LogRecord attributes also include padding and enclosing brackets, [and] to match the default values of their collateral non-colorized LogRecord attributes.

```
log_fmt_console: '[%(levelname)-8s] %(message)s'
```

log_fmt_logfile

Default: %(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s

The format of the log file logging messages. All standard python logging LogRecord attributes can be used. Salt also provides these custom LogRecord attributes that include padding and enclosing brackets [and]:

```
'%(bracketlevel)s'   # equivalent to [%(levelname)-8s]
'%(bracketname)s'    # equivalent to [%(name)-17s]
'%(bracketprocess)s' # equivalent to [%(process)5s]
```

```
log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'
```

log_granular_levels

Default: {}

This can be used to control logging levels more specifically, based on log call name. The example sets the main salt library at the `warning' level, sets salt.modules to log at the debug level, and sets a custom module to the all level:

```
log_granular_levels:
  'salt': 'warning'
  'salt.modules': 'debug'
  'salt.loader.saltmaster.ext.module.custom_module': 'all'
```

External Logging Handlers

Besides the internal logging handlers used by salt, there are some external which can be used, see the *external logging handlers* document.

3.11 External Logging Handlers

fluent_mod	Fluent Logging Handler
log4mongo_mod	Log4Mongo Logging Handler
logstash_mod	Logstash Logging Handler
sentry_mod	Sentry Logging Handler

3.11.1 salt.log.handlers.fluent_mod

Fluent Logging Handler

New in version 2015.8.0.

This module provides some fluentd logging handlers.

Fluent Logging Handler

In the *fluent* configuration file:

```
<source>
  type forward
  bind localhost
  port 24224
</source>
```

Then, to send logs via fluent in Logstash format, add the following to the salt (master and/or minion) configuration file:

```
fluent_handler:
  host: localhost
  port: 24224
```

To send logs via fluent in the Graylog raw json format, add the following to the salt (master and/or minion) configuration file:

```
fluent_handler:
  host: localhost
  port: 24224
  payload_type: graylog
  tags:
  - salt_master.SALT
```

The above also illustrates the *tags* option, which allows one to set descriptive (or useful) tags on records being sent. If not provided, this defaults to the single tag: `salt'. Also note that, via Graylog ``magic'', the `facility' of the logged message is set to `SALT' (the portion of the tag after the first period), while the tag itself will be set to simply `salt_master'. This is a feature, not a bug :)

Note: There is a third emitter, for the GELF format, but it is largely untested, and I don't currently have a setup supporting this config, so while it runs cleanly and outputs what LOOKS to be valid GELF, any real-world feedback on its usefulness, and correctness, will be appreciated.

Log Level

The `fluent_handler` configuration section accepts an additional setting `log_level`. If not set, the logging level used will be the one defined for `log_level` in the global configuration file section.

Inspiration

This work was inspired in fluent-logger-python

3.11.2 salt.log.handlers.log4mongo_mod

Log4Mongo Logging Handler

This module provides a logging handler for sending salt logs to MongoDB

Configuration

In the salt configuration file (e.g. /etc/salt/{master,minion}):

```
log4mongo_handler:
  host: mongodb_host
  port: 27017
  database_name: logs
  collection: salt_logs
  username: logging
  password: reindeerflotilla
  write_concern: 0
  log_level: warning
```

Log Level

If not set, the log_level will be set to the level defined in the global configuration file setting.

Inspiration

This work was inspired by the Salt logging handlers for LogStash and Sentry and by the log4mongo Python implementation.

3.11.3 salt.log.handlers.logstash_mod

Logstash Logging Handler

New in version 0.17.0.

This module provides some Logstash logging handlers.

UDP Logging Handler

For versions of Logstash before 1.2.0:

In the salt configuration file:

```
logstash_udp_handler:
  host: 127.0.0.1
  port: 9999
  version: 0
  msg_type: logstash
```

In the Logstash configuration file:

```
input {
  udp {
    type => "udp-type"
    format => "json_event"
  }
}
```

For version 1.2.0 of Logstash and newer:

In the salt configuration file:

```
logstash_udp_handler:
  host: 127.0.0.1
  port: 9999
  version: 1
  msg_type: logstash
```

In the Logstash configuration file:

```
input {
  udp {
    port => 9999
    codec => json
  }
}
```

Please read the UDP input configuration page for additional information.

ZeroMQ Logging Handler

For versions of Logstash before 1.2.0:

In the salt configuration file:

```
logstash_zmq_handler:
  address: tcp://127.0.0.1:2021
  version: 0
```

In the Logstash configuration file:

```
input {
  zeromq {
    type => "zeromq-type"
    mode => "server"
    topology => "pubsub"
    address => "tcp://0.0.0.0:2021"
    charset => "UTF-8"
    format => "json_event"
  }
}
```

For version 1.2.0 of Logstash and newer:

In the salt configuration file:

```
logstash_zmq_handler:
  address: tcp://127.0.0.1:2021
  version: 1
```

In the Logstash configuration file:

```
input {
  zeromq {
    topology => "pubsub"
    address => "tcp://0.0.0.0:2021"
    codec => json
  }
}
```

Please read the ZeroMQ input configuration page for additional information.

Important Logstash Setting

One of the most important settings that you should not forget on your Logstash configuration file regarding these logging handlers is `format`. Both the *UDP* and *ZeroMQ* inputs need to have `format` as `json_event` which is what we send over the wire.

Log Level

Both the `logstash_udp_handler` and the `logstash_zmq_handler` configuration sections accept an additional setting `log_level`. If not set, the logging level used will be the one defined for `log_level` in the global configuration file section.

HWM

The high water mark for the ZMQ socket setting. Only applicable for the `logstash_zmq_handler`.

Inspiration

This work was inspired in pylogstash, python-logstash, canary and the PyZMQ logging handler.

3.11.4 salt.log.handlers.sentry_mod

Sentry Logging Handler

New in version 0.17.0.

This module provides a Sentry logging handler. Sentry is an open source error tracking platform that provides deep context about exceptions that happen in production. Details about stack traces along with the context variables available at the time of the exeption are easily browsable and filterable from the online interface. For more details please see Sentry.

Note

The Raven library needs to be installed on the system for this logging handler to be available.

Configuring the python Sentry client, Raven, should be done under the `sentry_handler` configuration key. Additional *context* may be provided for corresponding grain item(s). At the bare minimum, you need to define the DSN. As an example:

```
sentry_handler:
  dsn: https://pub-key:secret-key@app.getsentry.com/app-id
```

More complex configurations can be achieved, for example:

```
sentry_handler:
  servers:
    - https://sentry.example.com
    - http://192.168.1.1
  project: app-id
  public_key: deadbeefdeadbeefdeadbeefdeadbeef
  secret_key: beefdeadbeefdeadbeefdeadbeefdead
  context:
    - os
    - master
    - saltversion
    - cpuarch
    - ec2.tags.environment
```

Note

The `public_key` and `secret_key` variables are not supported with Sentry > 3.0. The DSN key should be used instead.

All the client configuration keys are supported, please see the Raven client documentation.

The default logging level for the sentry handler is `ERROR`. If you wish to define a different one, define `log_level` under the `sentry_handler` configuration key:

```
sentry_handler:
  dsn: https://pub-key:secret-key@app.getsentry.com/app-id
  log_level: warning
```

The available log levels are those also available for the salt `cli` tools and configuration; `salt --help` should give you the required information.

Threaded Transports

Raven's documents rightly suggest using its threaded transport for critical applications. However, don't forget that if you start having troubles with Salt after enabling the threaded transport, please try switching to a non-threaded transport to see if that fixes your problem.

3.12 Salt File Server

Salt comes with a simple file server suitable for distributing files to the Salt minions. The file server is a stateless ZeroMQ server that is built into the Salt master.

The main intent of the Salt file server is to present files for use in the Salt state system. With this said, the Salt file server can be used for any general file transfer from the master to the minions.

3.12.1 File Server Backends

In Salt 0.12.0, the modular fileserver was introduced. This feature added the ability for the Salt Master to integrate different file server backends. File server backends allow the Salt file server to act as a transparent bridge to external resources. A good example of this is the `git` backend, which allows Salt to serve files sourced from one or more git repositories, but there are several others as well. Click *here* for a full list of Salt's fileserver backends.

Enabling a Fileserver Backend

Fileserver backends can be enabled with the *`fileserver_backend`* option.

```
fileserver_backend:
  - git
```

See the *documentation* for each backend to find the correct value to add to *`fileserver_backend`* in order to enable them.

Using Multiple Backends

If *`fileserver_backend`* is not defined in the Master config file, Salt will use the *`roots`* backend, but the *`fileserver_backend`* option supports multiple backends. When more than one backend is in use, the files from the enabled backends are merged into a single virtual filesystem. When a file is requested, the backends will be searched in order for that file, and the first backend to match will be the one which returns the file.

```
fileserver_backend:
  - roots
  - git
```

With this configuration, the environments and files defined in the *file_roots* parameter will be searched first, and if the file is not found then the git repositories defined in *gitfs_remotes* will be searched.

Defining Environments

Just as the order of the values in *fileserver_backend* matters, so too does the order in which different sources are defined within a fileserver environment. For example, given the below *file_roots* configuration, if both /srv/salt/dev/foo.txt and /srv/salt/prod/foo.txt exist on the Master, then salt://foo.txt would point to /srv/salt/dev/foo.txt in the dev environment, but it would point to /srv/salt/prod/foo.txt in the base environment.

```
file_roots:
  base:
    - /srv/salt/prod
  qa:
    - /srv/salt/qa
    - /srv/salt/prod
  dev:
    - /srv/salt/dev
    - /srv/salt/qa
    - /srv/salt/prod
```

Similarly, when using the *git* backend, if both repositories defined below have a hotfix23 branch/tag, and both of them also contain the file bar.txt in the root of the repository at that branch/tag, then salt://bar.txt in the hotfix23 environment would be served from the first repository.

```
gitfs_remotes:
  - https://mydomain.tld/repos/first.git
  - https://mydomain.tld/repos/second.git
```

Note: Environments map differently based on the fileserver backend. For instance, the mappings are explicitly defined in *roots* backend, while in the VCS backends (*git*, *hg*, *svn*) the environments are created from branches/tags/bookmarks/etc. For the *minion* backend, the files are all in a single environment, which is specified by the *minionfs_env* option.

See the documentation for each backend for a more detailed explanation of how environments are mapped.

3.12.2 Dynamic Module Distribution

New in version 0.9.5.

Custom Salt execution, state, and other modules can be distributed to Salt minions using the Salt file server.

Under the root of any environment defined via the *file_roots* option on the master server directories corresponding to the type of module can be used.

The directories are prepended with an underscore:

- _beacons
- _clouds
- _engines
- _grains

- _modules
- _output
- _proxy
- _renderers
- _returners
- _states
- _tops
- _utils

The contents of these directories need to be synced over to the minions after Python modules have been created in them. There are a number of ways to sync the modules.

Sync Via States

The minion configuration contains an option `autoload_dynamic_modules` which defaults to `True`. This option makes the state system refresh all dynamic modules when states are run. To disable this behavior set *autoload_dynamic_modules* to `False` in the minion config.

When dynamic modules are autoloaded via states, modules only pertinent to the environments matched in the master's top file are downloaded.

This is important to remember, because modules can be manually loaded from any specific environment that environment specific modules will be loaded when a state run is executed.

Sync Via the saltutil Module

The saltutil module has a number of functions that can be used to sync all or specific dynamic modules. The saltutil module function `saltutil.sync_all` will sync all module types over to a minion. For more information see: *salt.modules.saltutil*

3.12.3 Requesting Files from Specific Environments

The Salt fileserver supports multiple environments, allowing for SLS files and other files to be isolated for better organization.

For the default backend (called *roots*), environments are defined using the *roots* option. Other backends (such as *gitfs*) define environments in their own ways. For a list of available fileserver backends, see *here*.

Querystring Syntax

Any `salt://` file URL can specify its fileserver environment using a querystring syntax, like so:

```
salt://path/to/file?saltenv=foo
```

In *Reactor* configurations, this method must be used to pull files from an environment other than `base`.

In States

Minions can be instructed which environment to use both globally, and for a single state, and multiple methods for each are available:

Globally

A minion can be pinned to an environment using the *environment* option in the minion config file.

Additionally, the environment can be set for a single call to the following functions:

- *state.apply*
- *state.highstate*
- *state.sls*
- *state.top*

Note: When the `saltenv` parameter is used to trigger a *highstate* using either *state.apply* or *state.highstate*, only states from that environment will be applied.

On a Per-State Basis

Within an individual state, there are two ways of specifying the environment. The first is to add a `saltenv` argument to the state. This example will pull the file from the `config` environment:

```
/etc/foo/bar.conf:
  file.managed:
    - source: salt://foo/bar.conf
    - user: foo
    - mode: 600
    - saltenv: config
```

Another way of doing the same thing is to use the *querystring syntax* described above:

```
/etc/foo/bar.conf:
  file.managed:
    - source: salt://foo/bar.conf?saltenv=config
    - user: foo
    - mode: 600
```

Note: Specifying the environment using either of the above methods is only necessary in cases where a state from one environment needs to access files from another environment. If the SLS file containing this state was in the `config` environment, then it would look in that environment by default.

3.12.4 File Server Configuration

The Salt file server is a high performance file server written in ZeroMQ. It manages large files quickly and with little overhead, and has been optimized to handle small files in an extremely efficient manner.

The Salt file server is an environment aware file server. This means that files can be allocated within many root directories and accessed by specifying both the file path and the environment to search. The individual environments can span across multiple directory roots to create overlays and to allow for files to be organized in many flexible ways.

Environments

The Salt file server defaults to the mandatory `base` environment. This environment **MUST** be defined and is used to download files when no environment is specified.

Environments allow for files and sls data to be logically separated, but environments are not isolated from each other. This allows for logical isolation of environments by the engineer using Salt, but also allows for information to be used in multiple environments.

Directory Overlay

The `environment` setting is a list of directories to publish files from. These directories are searched in order to find the specified file and the first file found is returned.

This means that directory data is prioritized based on the order in which they are listed. In the case of this `file_roots` configuration:

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

If a file's URI is `salt://httpd/httpd.conf`, it will first search for the file at `/srv/salt/base/httpd/httpd.conf`. If the file is found there it will be returned. If the file is not found there, then `/srv/salt/failover/httpd/httpd.conf` will be used for the source.

This allows for directories to be overlaid and prioritized based on the order they are defined in the configuration.

It is also possible to have `file_roots` which supports multiple environments:

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

This example ensures that each environment will check the associated environment directory for files first. If a file is not found in the appropriate directory, the system will default to using the base directory.

Local File Server

New in version 0.9.8.

The file server can be rerouted to run from the minion. This is primarily to enable running Salt states without a Salt master. To use the local file server interface, copy the file server data to the minion and set the file_roots option on the minion to point to the directories copied from the master. Once the minion `file_roots` option has been set, change the `file_client` option to local to make sure that the local file server interface is used.

3.12.5 The cp Module

The cp module is the home of minion side file server operations. The cp module is used by the Salt state system, salt-cp, and can be used to distribute files presented by the Salt file server.

Escaping Special Characters

The `salt://` url format can potentially contain a query string, for example `salt://dir/file.txt?saltenv=base`. You can prevent the fileclient/fileserver from interpreting `?` as the initial token of a query string by referencing the file with `salt://|` rather than `salt://`.

```
/etc/marathon/conf/?checkpoint:
  file.managed:
    - source: salt://|hw/config/?checkpoint
    - makedirs: True
```

Environments

Since the file server is made to work with the Salt state system, it supports environments. The environments are defined in the master config file and when referencing an environment the file specified will be based on the root directory of the environment.

get_file

The cp.get_file function can be used on the minion to download a file from the master, the syntax looks like this:

```
# salt '*' cp.get_file salt://vimrc /etc/vimrc
```

This will instruct all Salt minions to download the vimrc file and copy it to /etc/vimrc

Template rendering can be enabled on both the source and destination file names like so:

```
# salt '*' cp.get_file "salt://{{grains.os}}/vimrc" /etc/vimrc template=jinja
```

This example would instruct all Salt minions to download the vimrc from a directory with the same name as their OS grain and copy it to /etc/vimrc

For larger files, the cp.get_file module also supports gzip compression. Because gzip is CPU-intensive, this should only be used in scenarios where the compression ratio is very high (e.g. pretty-printed JSON or YAML files).

To use compression, use the `gzip` named argument. Valid values are integers from 1 to 9, where 1 is the lightest compression and 9 the heaviest. In other words, 1 uses the least CPU on the master (and minion), while 9 uses the most.

```
# salt '*' cp.get_file salt://vimrc /etc/vimrc gzip=5
```

Finally, note that by default cp.get_file does *not* create new destination directories if they do not exist. To change this, use the `makedirs` argument:

```
# salt '*' cp.get_file salt://vimrc /etc/vim/vimrc makedirs=True
```

In this example, /etc/vim/ would be created if it didn't already exist.

get_dir

The cp.get_dir function can be used on the minion to download an entire directory from the master. The syntax is very similar to get_file:

```
# salt '*' cp.get_dir salt://etc/apache2 /etc
```

cp.get_dir supports template rendering and gzip compression arguments just like get_file:

```
# salt '*' cp.get_dir salt://etc/{{pillar.webserver}} /etc gzip=5 template=jinja
```

3.12.6 File Server Client Instance

A client instance is available which allows for modules and applications to be written which make use of the Salt file server.

The file server uses the same authentication and encryption used by the rest of the Salt system for network communication.

fileclient Module

The `salt/fileclient.py` module is used to set up the communication from the minion to the master. When creating a client instance using the fileclient module, the minion configuration needs to be passed in. When using the fileclient module from within a minion module the built in `__opts__` data can be passed:

```
import salt.minion
import salt.fileclient

def get_file(path, dest, saltenv='base'):
    '''
    Used to get a single file from the Salt master

    CLI Example:
    salt '*' cp.get_file salt://vimrc /etc/vimrc
    '''
    # Get the fileclient object
    client = salt.fileclient.get_file_client(__opts__)
    # Call get_file
    return client.get_file(path, dest, False, saltenv)
```

Creating a fileclient instance outside of a minion module where the `__opts__` data is not available, it needs to be generated:

```
import salt.fileclient
import salt.config

def get_file(path, dest, saltenv='base'):
    '''
    Used to get a single file from the Salt master
    '''
    # Get the configuration data
    opts = salt.config.minion_config('/etc/salt/minion')
    # Get the fileclient object
    client = salt.fileclient.get_file_client(opts)
    # Call get_file
    return client.get_file(path, dest, False, saltenv)
```

3.13 Git Fileserver Backend Walkthrough

Note: This walkthrough assumes basic knowledge of Salt. To get up to speed, check out the *Salt Walkthrough*.

The gitfs backend allows Salt to serve files from git repositories. It can be enabled by adding `git` to the `fileserver_backend` list, and configuring one or more repositories in `gitfs_remotes`.

Branches and tags become Salt fileserver environments.

Note: Branching and tagging can result in a lot of potentially-conflicting *top files*, for this reason it may be useful to set `top_file_merging_strategy` to `same` in the minions' config files if the top files are being managed in a GitFS repo.

3.13.1 Installing Dependencies

Both pygit2 and GitPython are supported Python interfaces to git. If compatible versions of both are installed, pygit2 will be preferred. In these cases, GitPython can be forced using the `gitfs_provider` parameter in the master config file.

Note: It is recommended to always run the most recent version of any the below dependencies. Certain features of GitFS may not be available without the most recent version of the chosen library.

pygit2

The minimum supported version of pygit2 is 0.20.3. Availability for this version of pygit2 is still limited, though the SaltStack team is working to get compatible versions available for as many platforms as possible.

For the Fedora/EPEL versions which have a new enough version packaged, the following command would be used to install pygit2:

```
# yum install python-pygit2
```

Provided a valid version is packaged for Debian/Ubuntu (which is not currently the case), the package name would be the same, and the following command would be used to install it:

```
# apt-get install python-pygit2
```

If pygit2 is not packaged for the platform on which the Master is running, the pygit2 website has installation instructions here. Keep in mind however that following these instructions will install libgit2 and pygit2 without system packages. Additionally, keep in mind that *SSH authentication in pygit2* requires libssh2 (*not* libssh) development libraries to be present before libgit2 is built. On some Debian-based distros `pkg-config` is also required to link libgit2 with libssh2.

Note: If you are receiving the error ``Unsupported URL Protocol'' in the Salt Master log when making a connection using SSH, review the libssh2 details listed above.

Additionally, version 0.21.0 of pygit2 introduced a dependency on python-cffi, which in turn depends on newer releases of libffi. Upgrading libffi is not advisable as several other applications depend on it, so on older LTS linux releases pygit2 0.20.3 and libgit2 0.20.0 is the recommended combination.

> **Warning:** pygit2 is actively developed and frequently makes non-backwards-compatible API changes, even in minor releases. It is not uncommon for pygit2 upgrades to result in errors in Salt. Please take care when upgrading pygit2, and pay close attention to the changelog, keeping an eye out for API changes. Errors can be reported on the SaltStack issue tracker.

RedHat Pygit2 Issues

The release of RedHat/CentOS 7.3 upgraded both `python-cffi` and `http-parser`, both of which are dependencies for pygit2/libgit2. Both `pygit2` and `libgit2` packages (which are from the EPEL repository) should be upgraded to the most recent versions, at least to `0.24.2`.

The below errors will show up in the master log if an incompatible `python-pygit2` package is installed:

```
2017-02-10 09:07:34,892 [salt.utils.gitfs ][ERROR ][11211] Import pygit2 failed:
↪CompileError: command 'gcc' failed with exit status 1
2017-02-10 09:07:34,907 [salt.utils.gitfs ][ERROR ][11211] gitfs is configured but
↪could not be loaded, are pygit2 and libgit2 installed?
2017-02-10 09:07:34,907 [salt.utils.gitfs ][CRITICAL][11211] No suitable gitfs
↪provider module is installed.
2017-02-10 09:07:34,912 [salt.master ][CRITICAL][11211] Master failed pre flight
↪checks, exiting
```

The below errors will show up in the master log if an incompatible `libgit2` package is installed:

```
2017-02-15 18:04:45,211 [salt.utils.gitfs ][ERROR   ][6211] Error occurred fetching
↪gitfs remote 'https://foo.com/bar.git': No Content-Type header in response
```

A restart of the `salt-master` daemon and gitfs cache directory clean up may be required to allow http(s) repositories to continue to be fetched.

GitPython

GitPython 0.3.0 or newer is required to use GitPython for gitfs. For RHEL-based Linux distros, a compatible version is available in EPEL, and can be easily installed on the master using yum:

```
# yum install GitPython
```

Ubuntu 14.04 LTS and Debian Wheezy (7.x) also have a compatible version packaged:

```
# apt-get install python-git
```

GitPython requires the `git` CLI utility to work. If installed from a system package, then git should already be installed, but if installed via pip then it may still be necessary to install git separately. For MacOS users, GitPython comes bundled in with the Salt installer, but git must still be installed for it to work properly. Git can be installed in several ways, including by installing XCode.

Warning: Keep in mind that if GitPython has been previously installed on the master using pip (even if it was subsequently uninstalled), then it may still exist in the build cache (typically `/tmp/pip-build-root/GitPython`) if the cache is not cleared after installation. The package in the build cache will override any requirement specifiers, so if you try upgrading to version 0.3.2.RC1 by running `pip install 'GitPython==0.3.2.RC1'` then it will ignore this and simply install the version from the cache directory. Therefore, it may be necessary to delete the GitPython directory from the build cache in order to ensure that the specified version is installed.

Warning: GitPython 2.0.9 and newer is not compatible with Python 2.6. If installing GitPython using pip on a machine running Python 2.6, make sure that a version earlier than 2.0.9 is installed. This can be done on the CLI by running `pip install 'GitPython<2.0.9'`, or in a *pip.installed* state using the following SLS:

```
GitPython:
  pip.installed:
    - name: 'GitPython < 2.0.9'
```

3.13.2 Simple Configuration

To use the gitfs backend, only two configuration changes are required on the master:

1. Include `gitfs` in the *fileserver_backend* list in the master config file:

   ```
   fileserver_backend:
     - gitfs
   ```

   Note: `git` also works here. Prior to the 2018.3.0 release, *only* `git` would work.

2. Specify one or more `git://`, `https://`, `file://`, or `ssh://` URLs in *gitfs_remotes* to configure which repositories to cache and search for requested files:

   ```
   gitfs_remotes:
     - https://github.com/saltstack-formulas/salt-formula.git
   ```

   SSH remotes can also be configured using scp-like syntax:

   ```
   gitfs_remotes:
     - git@github.com:user/repo.git
     - ssh://user@domain.tld/path/to/repo.git
   ```

   Information on how to authenticate to SSH remotes can be found *here*.

3. Restart the master to load the new configuration.

Note: In a master/minion setup, files from a gitfs remote are cached once by the master, so minions do not need direct access to the git repository.

3.13.3 Multiple Remotes

The `gitfs_remotes` option accepts an ordered list of git remotes to cache and search, in listed order, for requested files.

A simple scenario illustrates this cascading lookup behavior:

If the `gitfs_remotes` option specifies three remotes:

```
gitfs_remotes:
  - git://github.com/example/first.git
  - https://github.com/example/second.git
  - file:///root/third
```

And each repository contains some files:

```
first.git:
    top.sls
    edit/vim.sls
    edit/vimrc
    nginx/init.sls

second.git:
    edit/dev_vimrc
    haproxy/init.sls

third:
    haproxy/haproxy.conf
    edit/dev_vimrc
```

Salt will attempt to lookup the requested file from each gitfs remote repository in the order in which they are defined in the configuration. The **git://github.com/example/first.git** remote will be searched first. If the requested file is found, then it is served and no further searching is executed. For example:

- A request for the file **salt://haproxy/init.sls** will be served from the **https://github.com/example/second.git** git repo.
- A request for the file **salt://haproxy/haproxy.conf** will be served from the **file:///root/third** repo.

Note: This example is purposefully contrived to illustrate the behavior of the gitfs backend. This example should not be read as a recommended way to lay out files and git repos.

The **file://** prefix denotes a git repository in a local directory. However, it will still use the given **file://** URL as a remote, rather than copying the git repo to the salt cache. This means that any refs you want accessible must exist as *local* refs in the specified repo.

Warning: Salt versions prior to 2014.1.0 are not tolerant of changing the order of remotes or modifying the URI of existing remotes. In those versions, when modifying remotes it is a good idea to remove the gitfs cache directory (`/var/cache/salt/master/gitfs`) before restarting the salt-master service.

3.13.4 Per-remote Configuration Parameters

New in version 2014.7.0.

The following master config parameters are global (that is, they apply to all configured gitfs remotes):

- *gitfs_base*
- *gitfs_root*
- *gitfs_ssl_verify*
- *gitfs_mountpoint* (new in 2014.7.0)
- *gitfs_user* (**pygit2 only**, new in 2014.7.0)
- *gitfs_password* (**pygit2 only**, new in 2014.7.0)
- *gitfs_insecure_auth* (**pygit2 only**, new in 2014.7.0)
- *gitfs_pubkey* (**pygit2 only**, new in 2014.7.0)
- *gitfs_privkey* (**pygit2 only**, new in 2014.7.0)
- *gitfs_passphrase* (**pygit2 only**, new in 2014.7.0)
- *gitfs_refspecs* (new in 2017.7.0)
- *gitfs_disable_saltenv_mapping* (new in 2018.3.0)
- *gitfs_ref_types* (new in 2018.3.0)
- *gitfs_update_interval* (new in 2018.3.0)

Note: pygit2 only supports disabling SSL verification in versions 0.23.2 and newer.

These parameters can now be overridden on a per-remote basis. This allows for a tremendous amount of customization. Here's some example usage:

```
gitfs_provider: pygit2
gitfs_base: develop

gitfs_remotes:
  - https://foo.com/foo.git
  - https://foo.com/bar.git:
    - root: salt
    - mountpoint: salt://bar
    - base: salt-base
    - ssl_verify: False
    - update_interval: 120
  - https://foo.com/bar.git:
    - name: second_bar_repo
    - root: other/salt
    - mountpoint: salt://other/bar
    - base: salt-base
    - ref_types:
      - branch
  - http://foo.com/baz.git:
    - root: salt/states
    - user: joe
    - password: mysupersecretpassword
    - insecure_auth: True
    - disable_saltenv_mapping: True
    - saltenv:
      - foo:
        - ref: foo
  - http://foo.com/quux.git:
    - all_saltenvs: master
```

Important: There are two important distinctions which should be noted for per-remote configuration:

1. The URL of a remote which has per-remote configuration must be suffixed with a colon.
2. Per-remote configuration parameters are named like the global versions, with the `gitfs_` removed from the beginning. The exception being the `name`, `saltenv`, and `all_saltenvs` parameters, which are only available to per-remote configurations.

The `all_saltenvs` parameter is new in the 2018.3.0 release.

In the example configuration above, the following is true:

1. The first and fourth gitfs remotes will use the `develop` branch/tag as the `base` environment, while the second and third will use the `salt-base` branch/tag as the `base` environment.
2. The first remote will serve all files in the repository. The second remote will only serve files from the `salt` directory (and its subdirectories). The third remote will only server files from the `other/salt` directory (and its subdirectories), while the fourth remote will only serve files from the `salt/states` directory (and its subdirectories).
3. The third remote will only serve files from branches, and not from tags or SHAs.
4. The fourth remote will only have two saltenvs available: `base` (pointed at `develop`), and `foo` (pointed at `foo`).
5. The first and fourth remotes will have files located under the root of the Salt fileserver namespace (`salt://`). The files from the second remote will be located under `salt://bar`, while the files from the third remote will be located under `salt://other/bar`.
6. The second and third remotes reference the same repository and unique names need to be declared for duplicate gitfs remotes.
7. The fourth remote overrides the default behavior of *not authenticating to insecure (non-HTTPS) remotes*.
8. Because `all_saltenvs` is configured for the fifth remote, files from the branch/tag `master` will appear in every fileserver environment.

   Note: The use of `http://` (instead of `https://`) is permitted here *only* because authentication is not being used. Otherwise, the `insecure_auth` parameter must be used (as in the fourth remote) to force Salt to authenticate to an `http://` remote.

9. The first remote will wait 120 seconds between updates instead of 60.

3.13.5 Per-Saltenv Configuration Parameters

New in version 2016.11.0.

For more granular control, Salt allows the following three things to be overridden for individual saltenvs within a given repo:

- The *mountpoint*
- The *root*
- The branch/tag to be used for a given saltenv

Here is an example:

```
gitfs_root: salt

gitfs_saltenv:
  - dev:
    - mountpoint: salt://gitfs-dev
    - ref: develop

gitfs_remotes:
  - https://foo.com/bar.git:
    - saltenv:
      - staging:
        - ref: qa
        - mountpoint: salt://bar-staging
      - dev:
        - ref: development
  - https://foo.com/baz.git:
    - saltenv:
      - staging:
        - mountpoint: salt://baz-staging
```

Given the above configuration, the following is true:

1. For all gitfs remotes, files for the `dev` saltenv will be located under `salt://gitfs-dev`.
2. For the `dev` saltenv, files from the first remote will be sourced from the `development` branch, while files from the second remote will be sourced from the `develop` branch.
3. For the `staging` saltenv, files from the first remote will be located under `salt://bar-staging`, while files from the second remote will be located under `salt://baz-staging`.
4. For all gitfs remotes, and in all saltenvs, files will be served from the `salt` directory (and its subdirectories).

3.13.6 Custom Refspecs

New in version 2017.7.0.

GitFS will by default fetch remote branches and tags. However, sometimes it can be useful to fetch custom refs (such as those created for GitHub pull requests). To change the refspecs GitFS fetches, use the *`gitfs_refspecs`* config option:

```
gitfs_refspecs:
  - '+refs/heads/*:refs/remotes/origin/*'
  - '+refs/tags/*:refs/tags/*'
  - '+refs/pull/*/head:refs/remotes/origin/pr/*'
  - '+refs/pull/*/merge:refs/remotes/origin/merge/*'
```

In the above example, in addition to fetching remote branches and tags, GitHub's custom refs for pull requests and merged pull requests will also be fetched. These special `head` refs represent the head of the branch which is requesting to be merged, and the `merge` refs represent the result of the base branch after the merge.

Important: When using custom refspecs, the destination of the fetched refs *must* be under `refs/remotes/origin/`, preferably in a subdirectory like in the example above. These custom refspecs will map as environment names using their relative path underneath `refs/remotes/origin/`. For example, assuming the configuration above, the head branch for pull request 12345 would map to fileserver environment `pr/12345` (slash included).

Refspecs can be configured on a *per-remote basis*. For example, the below configuration would only alter the default refspecs for the *second* GitFS remote. The first remote would only fetch branches and tags (the default).

```
gitfs_remotes:
  - https://domain.tld/foo.git
  - https://domain.tld/bar.git:
    - refspecs:
      - '+refs/heads/*:refs/remotes/origin/*'
      - '+refs/tags/*:refs/tags/*'
      - '+refs/pull/*/head:refs/remotes/origin/pr/*'
      - '+refs/pull/*/merge:refs/remotes/origin/merge/*'
```

3.13.7 Global Remotes

New in version 2018.3.0.

The `all_saltenvs` per-remote configuration parameter overrides the logic Salt uses to map branches/tags to fileserver environments (i.e. saltenvs). This allows a single branch/tag to appear in *all* saltenvs.

This is very useful in particular when working with *salt formulas*. Prior to the addition of this feature, it was necessary to push a branch/tag to the remote repo for each saltenv in which that formula was to be used. If the formula needed to be updated, this update would need to be reflected in all of the other branches/tags. This is both inconvenient and not scalable.

With `all_saltenvs`, it is now possible to define your formula once, in a single branch.

```
gitfs_remotes:
  - http://foo.com/quux.git:
    - all_saltenvs: anything
```

3.13.8 Update Intervals

Prior to the 2018.3.0 release, GitFS would update its fileserver backends as part of a dedicated ``maintenance'' process, in which various routine maintenance tasks were performed. This tied the update interval to the `loop_interval` config option, and also forced all fileservers to update at the same interval.

Now it is possible to make GitFS update at its own interval, using `gitfs_update_interval`:

```
gitfs_update_interval: 180

gitfs_remotes:
  - https://foo.com/foo.git
  - https://foo.com/bar.git:
    - update_interval: 120
```

Using the above configuration, the first remote would update every three minutes, while the second remote would update every two minutes.

3.13.9 Configuration Order of Precedence

The order of precedence for GitFS configuration is as follows (each level overrides all levels below it):

1. Per-saltenv configuration (defined under a per-remote `saltenv` param)

```
gitfs_remotes:
  - https://foo.com/bar.git:
    - saltenv:
      - dev:
        - mountpoint: salt://bar
```

2. Global per-saltenv configuration (defined in *gitfs_saltenv*)

```
gitfs_saltenv:
  - dev:
    - mountpoint: salt://bar
```

3. Per-remote configuration parameter

```
gitfs_remotes:
  - https://foo.com/bar.git:
    - mountpoint: salt://bar
```

4. Global configuration parameter

```
gitfs_mountpoint: salt://bar
```

Note: The one exception to the above is when *all_saltenvs* is used. This value overrides all logic for mapping branches/tags to fileserver environments. So, even if *gitfs_saltenv* is used to globally override the mapping for a given saltenv, *all_saltenvs* would take precedence for any remote which uses it.

It's important to note however that any `root` and `mountpoint` values configured in *gitfs_saltenv* (or *per-saltenv configuration*) would be unaffected by this.

3.13.10 Serving from a Subdirectory

The *gitfs_root* parameter allows files to be served from a subdirectory within the repository. This allows for only part of a repository to be exposed to the Salt fileserver.

Assume the below layout:

```
.gitignore
README.txt
foo/
foo/bar/
foo/bar/one.txt
foo/bar/two.txt
foo/bar/three.txt
foo/baz/
foo/baz/top.sls
foo/baz/edit/vim.sls
foo/baz/edit/vimrc
foo/baz/nginx/init.sls
```

The below configuration would serve only the files under `foo/baz`, ignoring the other files in the repository:

```
gitfs_remotes:
  - git://mydomain.com/stuff.git

gitfs_root: foo/baz
```

The root can also be configured on a *per-remote basis*.

3.13.11 Mountpoints

New in version 2014.7.0.

The `gitfs_mountpoint` parameter will prepend the specified path to the files served from gitfs. This allows an existing repository to be used, rather than needing to reorganize a repository or design it around the layout of the Salt fileserver.

Before the addition of this feature, if a file being served up via gitfs was deeply nested within the root directory (for example, `salt://webapps/foo/files/foo.conf`, it would be necessary to ensure that the file was properly located in the remote repository, and that all of the parent directories were present (for example, the directories `webapps/foo/files/` would need to exist at the root of the repository).

The below example would allow for a file `foo.conf` at the root of the repository to be served up from the Salt fileserver path `salt://webapps/foo/files/foo.conf`.

```
gitfs_remotes:
  - https://mydomain.com/stuff.git

gitfs_mountpoint: salt://webapps/foo/files
```

Mountpoints can also be configured on a *per-remote basis*.

3.13.12 Using gitfs in Masterless Mode

Since 2014.7.0, gitfs can be used in masterless mode. To do so, simply add the gitfs configuration parameters (and set `fileserver_backend`) in the _minion_ config file instead of the master config file.

3.13.13 Using gitfs Alongside Other Backends

Sometimes it may make sense to use multiple backends; for instance, if `sls` files are stored in git but larger files are stored directly on the master.

The cascading lookup logic used for multiple remotes is also used with multiple backends. If the `fileserver_backend` option contains multiple backends:

```
fileserver_backend:
  - roots
  - git
```

Then the `roots` backend (the default backend of files in `/srv/salt`) will be searched first for the requested file; then, if it is not found on the master, each configured git remote will be searched.

3.13.14 Branches, Environments, and Top Files

When using the GitFS backend, branches, and tags will be mapped to environments using the branch/tag name as an identifier.

There is one exception to this rule: the `master` branch is implicitly mapped to the `base` environment.

So, for a typical `base`, `qa`, `dev` setup, the following branches could be used:

```
master
qa
dev
```

top.sls files from different branches will be merged into one at runtime. Since this can lead to overly complex configurations, the recommended setup is to have a separate repository, containing only the top.sls file with just one single master branch.

To map a branch other than master as the base environment, use the *gitfs_base* parameter.

```
gitfs_base: salt-base
```

The base can also be configured on a *per-remote basis*.

3.13.15 Environment Whitelist/Blacklist

New in version 2014.7.0.

The *gitfs_saltenv_whitelist* and *gitfs_saltenv_blacklist* parameters allow for greater control over which branches/tags are exposed as fileserver environments. Exact matches, globs, and regular expressions are supported, and are evaluated in that order. If using a regular expression, ^ and $ must be omitted, and the expression must match the entire branch/tag.

```
gitfs_saltenv_whitelist:
  - base
  - v1.*
  - 'mybranch\d+'
```

Note: v1.*, in this example, will match as both a glob and a regular expression (though it will have been matched as a glob, since globs are evaluated before regular expressions).

The behavior of the blacklist/whitelist will differ depending on which combination of the two options is used:

- If only *gitfs_saltenv_whitelist* is used, then **only** branches/tags which match the whitelist will be available as environments
- If only *gitfs_saltenv_blacklist* is used, then the branches/tags which match the blacklist will **not** be available as environments
- If both are used, then the branches/tags which match the whitelist, but do **not** match the blacklist, will be available as environments.

3.13.16 Authentication

pygit2

New in version 2014.7.0.

Both HTTPS and SSH authentication are supported as of version 0.20.3, which is the earliest version of pygit2 supported by Salt for gitfs.

Note: The examples below make use of per-remote configuration parameters, a feature new to Salt 2014.7.0. More information on these can be found *here*.

HTTPS

For HTTPS repositories which require authentication, the username and password can be provided like so:

```
gitfs_remotes:
  - https://domain.tld/myrepo.git:
    - user: git
    - password: mypassword
```

If the repository is served over HTTP instead of HTTPS, then Salt will by default refuse to authenticate to it. This behavior can be overridden by adding an `insecure_auth` parameter:

```
gitfs_remotes:
  - http://domain.tld/insecure_repo.git:
    - user: git
    - password: mypassword
    - insecure_auth: True
```

SSH

SSH repositories can be configured using the `ssh://` protocol designation, or using scp-like syntax. So, the following two configurations are equivalent:

- `ssh://git@github.com/user/repo.git`
- `git@github.com:user/repo.git`

Both *gitfs_pubkey* and *gitfs_privkey* (or their *per-remote counterparts*) must be configured in order to authenticate to SSH-based repos. If the private key is protected with a passphrase, it can be configured using *gitfs_passphrase* (or simply `passphrase` if being configured *per-remote*). For example:

```
gitfs_remotes:
  - git@github.com:user/repo.git:
    - pubkey: /root/.ssh/id_rsa.pub
    - privkey: /root/.ssh/id_rsa
    - passphrase: myawesomepassphrase
```

Finally, the SSH host key must be *added to the known_hosts file*.

Note: There is a known issue with public-key SSH authentication to Microsoft Visual Studio (VSTS) with pygit2. This is due to a bug or lack of support for VSTS in older libssh2 releases. Known working releases include libssh2 1.7.0 and later, and known incompatible releases include 1.5.0 and older. At the time of this writing, 1.6.0 has not been tested.

Since upgrading libssh2 would require rebuilding many other packages (curl, etc.), followed by a rebuild of libgit2 and a reinstall of pygit2, an easier workaround for systems with older libssh2 is to use GitPython with a passphraseless key for authentication.

GitPython

HTTPS

For HTTPS repositories which require authentication, the username and password can be configured in one of two ways. The first way is to include them in the URL using the format `https://<user>:<password>@<url>`,

like so:

```
gitfs_remotes:
  - https://git:mypassword@domain.tld/myrepo.git
```

The other way would be to configure the authentication in `~/.netrc`:

```
machine domain.tld
login git
password mypassword
```

If the repository is served over HTTP instead of HTTPS, then Salt will by default refuse to authenticate to it. This behavior can be overridden by adding an `insecure_auth` parameter:

```
gitfs_remotes:
  - http://git:mypassword@domain.tld/insecure_repo.git:
    - insecure_auth: True
```

SSH

Only passphrase-less SSH public key authentication is supported using GitPython. **The auth parameters (pubkey, privkey, etc.) shown in the pygit2 authentication examples above do not work with GitPython.**

```
gitfs_remotes:
  - ssh://git@github.com/example/salt-states.git
```

Since GitPython wraps the git CLI, the private key must be located in `~/.ssh/id_rsa` for the user under which the Master is running, and should have permissions of `0600`. Also, in the absence of a user in the repo URL, GitPython will (just as SSH does) attempt to login as the current user (in other words, the user under which the Master is running, usually `root`).

If a key needs to be used, then `~/.ssh/config` can be configured to use the desired key. Information on how to do this can be found by viewing the manpage for `ssh_config`. Here's an example entry which can be added to the `~/.ssh/config` to use an alternate key for gitfs:

```
Host github.com
    IdentityFile /root/.ssh/id_rsa_gitfs
```

The `Host` parameter should be a hostname (or hostname glob) that matches the domain name of the git repository.

It is also necessary to *add the SSH host key to the known_hosts file*. The exception to this would be if strict host key checking is disabled, which can be done by adding `StrictHostKeyChecking no` to the entry in `~/.ssh/config`

```
Host github.com
    IdentityFile /root/.ssh/id_rsa_gitfs
    StrictHostKeyChecking no
```

However, this is generally regarded as insecure, and is not recommended.

Adding the SSH Host Key to the known_hosts File

To use SSH authentication, it is necessary to have the remote repository's SSH host key in the `~/.ssh/known_hosts` file. If the master is also a minion, this can be done using the *ssh.set_known_host* function:

```
# salt mymaster ssh.set_known_host user=root hostname=github.com
mymaster:
    ----------
    new:
        ----------
        enc:
            ssh-rsa
        fingerprint:
            16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48
        hostname:
            |1|OiefWWqOD4kwO3BhoIGa0loR5AA=|BIXVtmcTbPER+68HvXmceodDcfI=
        key:
           ␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6p
↪yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3J
↪w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
    old:
        None
    status:
        updated
```

If not, then the easiest way to add the key is to su to the user (usually `root`) under which the salt-master runs and attempt to login to the server via SSH:

```
$ su -
Password:
# ssh github.com
The authenticity of host 'github.com (192.30.252.128)' can't be established.
RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.252.128' (RSA) to the list of known hosts.
Permission denied (publickey).
```

It doesn't matter if the login was successful, as answering `yes` will write the fingerprint to the known_hosts file.

Verifying the Fingerprint

To verify that the correct fingerprint was added, it is a good idea to look it up. One way to do this is to use `nmap`:

```
$ nmap -p 22 github.com --script ssh-hostkey

Starting Nmap 5.51 ( http://nmap.org ) at 2014-08-18 17:47 CDT
Nmap scan report for github.com (192.30.252.129)
Host is up (0.17s latency).
Not shown: 996 filtered ports
PORT     STATE SERVICE
22/tcp   open  ssh
| ssh-hostkey: 1024 ad:1c:08:a4:40:e3:6f:9c:f5:66:26:5d:4b:33:5d:8c (DSA)
|_2048 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48 (RSA)
80/tcp   open  http
443/tcp  open  https
9418/tcp open  git

Nmap done: 1 IP address (1 host up) scanned in 28.78 seconds
```

Another way is to check one's own `known_hosts` file, using this one-liner:

```
$ ssh-keygen -l -f /dev/stdin <<<`ssh-keyscan github.com 2>/dev/null` | awk '{print $2}'
16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48
```

Warning: AWS tracks usage of nmap and may flag it as abuse. On AWS hosts, the `ssh-keygen` method is recommended for host key verification.

Note: As of OpenSSH 6.8 the SSH fingerprint is now shown as a base64-encoded SHA256 checksum of the host key. So, instead of the fingerprint looking like `16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48`, it would look like `SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8`.

3.13.17 Refreshing gitfs Upon Push

By default, Salt updates the remote fileserver backends every 60 seconds. However, if it is desirable to refresh quicker than that, the *Reactor System* can be used to signal the master to update the fileserver on each push, provided that the git server is also a Salt minion. There are three steps to this process:

1. On the master, create a file **/srv/reactor/update_fileserver.sls**, with the following contents:

```
update_fileserver:
  runner.fileserver.update
```

2. Add the following reactor configuration to the master config file:

```
reactor:
  - 'salt/fileserver/gitfs/update':
    - /srv/reactor/update_fileserver.sls
```

3. On the git server, add a post-receive hook

   (a) If the user executing *git push* is the same as the minion user, use the following hook:

```
#!/usr/bin/env sh
salt-call event.fire_master update salt/fileserver/gitfs/update
```

   (a) To enable other git users to run the hook after a *push*, use sudo in the hook script:

```
#!/usr/bin/env sh
sudo -u root salt-call event.fire_master update salt/fileserver/gitfs/
↪update
```

2. If using sudo in the git hook (above), the policy must be changed to permit all users to fire the event. Add the following policy to the sudoers file on the git server.

```
Cmnd_Alias SALT_GIT_HOOK = /bin/salt-call event.fire_master update salt/
↪fileserver/gitfs/update
Defaults!SALT_GIT_HOOK !requiretty
ALL ALL=(root) NOPASSWD: SALT_GIT_HOOK
```

The `update` argument right after *`event.fire_master`* in this example can really be anything, as it represents the data being passed in the event, and the passed data is ignored by this reactor.

Similarly, the tag name `salt/fileserver/gitfs/update` can be replaced by anything, so long as the usage is consistent.

The `root` user name in the hook script and sudo policy should be changed to match the user under which the minion is running.

3.13.18 Using Git as an External Pillar Source

The git external pillar (a.k.a. git_pillar) has been rewritten for the 2015.8.0 release. This rewrite brings with it pygit2 support (allowing for access to authenticated repositories), as well as more granular support for per-remote configuration. This configuration schema is detailed *here*.

3.13.19 Why aren't my custom modules/states/etc. syncing to my Minions?

In versions 0.16.3 and older, when using the *git fileserver backend*, certain versions of GitPython may generate errors when fetching, which Salt fails to catch. While not fatal to the fetch process, these interrupt the fileserver update that takes place before custom types are synced, and thus interrupt the sync itself. Try disabling the git fileserver backend in the master config, restarting the master, and attempting the sync again.

This issue is worked around in Salt 0.16.4 and newer.

3.14 MinionFS Backend Walkthrough

New in version 2014.1.0.

Note: This walkthrough assumes basic knowledge of Salt and *cp.push*. To get up to speed, check out the *Salt Walkthrough*.

Sometimes it is desirable to deploy a file located on one minion to one or more other minions. This is supported in Salt, and can be accomplished in two parts:

1. Minion support for pushing files to the master (using *cp.push*)
2. The *minionfs* fileserver backend

This walkthrough will show how to use both of these features.

3.14.1 Enabling File Push

To set the master to accept files pushed from minions, the *file_recv* option in the master config file must be set to `True` (the default is `False`).

```
file_recv: True
```

Note: This change requires a restart of the salt-master service.

3.14.2 Pushing Files

Once this has been done, files can be pushed to the master using the *cp.push* function:

```
salt 'minion-id' cp.push /path/to/the/file
```

This command will store the file in a subdirectory named `minions` under the master's *cachedir*. On most masters, this path will be `/var/cache/salt/master/minions`. Within this directory will be one directory for each minion which has pushed a file to the master, and underneath that the full path to the file on the minion. So, for example, if a minion with an ID of `dev1` pushed a file `/var/log/myapp.log` to the master, it would be saved to `/var/cache/salt/master/minions/dev1/var/log/myapp.log`.

3.14.3 Serving Pushed Files Using MinionFS

While it is certainly possible to add `/var/cache/salt/master/minions` to the master's *file_roots* and serve these files, it may only be desirable to expose files pushed from certain minions. Adding `/var/cache/salt/master/minions/<minion-id>` for each minion that needs to be exposed can be cumbersome and prone to errors.

Enter *minionfs*. This fileserver backend will make files pushed using *cp.push* available to the Salt fileserver, and provides an easy mechanism to restrict which minions' pushed files are made available.

Simple Configuration

To use the *minionfs* backend, add `minionfs` to the list of backends in the *fileserver_backend* configuration option on the master:

```
file_recv: True

fileserver_backend:
  - roots
  - minionfs
```

Note: `minion` also works here. Prior to the 2018.3.0 release, *only* `minion` would work.

Also, as described earlier, `file_recv: True` is needed to enable the master to receive files pushed from minions. As always, changes to the master configuration require a restart of the `salt-master` service.

Files made available via *minionfs* are by default located at `salt://<minion-id>/path/to/file`. Think back to the earlier example, in which `dev1` pushed a file `/var/log/myapp.log` to the master. With *minionfs* enabled, this file would be addressable in Salt at `salt://dev1/var/log/myapp.log`.

If many minions have pushed to the master, this will result in many directories in the root of the Salt fileserver. For this reason, it is recommended to use the *minionfs_mountpoint* config option to organize these files underneath a subdirectory:

```
minionfs_mountpoint: salt://minionfs
```

Using the above mountpoint, the file in the example would be located at `salt://minionfs/dev1/var/log/myapp.log`.

Restricting Certain Minions' Files from Being Available Via MinionFS

A whitelist and blacklist can be used to restrict the minions whose pushed files are available via *minionfs*. These lists can be managed using the *minionfs_whitelist* and *minionfs_blacklist* config options. Click the links for both of them for a detailed explanation of how to use them.

A more complex configuration example, which uses both a whitelist and blacklist, can be found below:

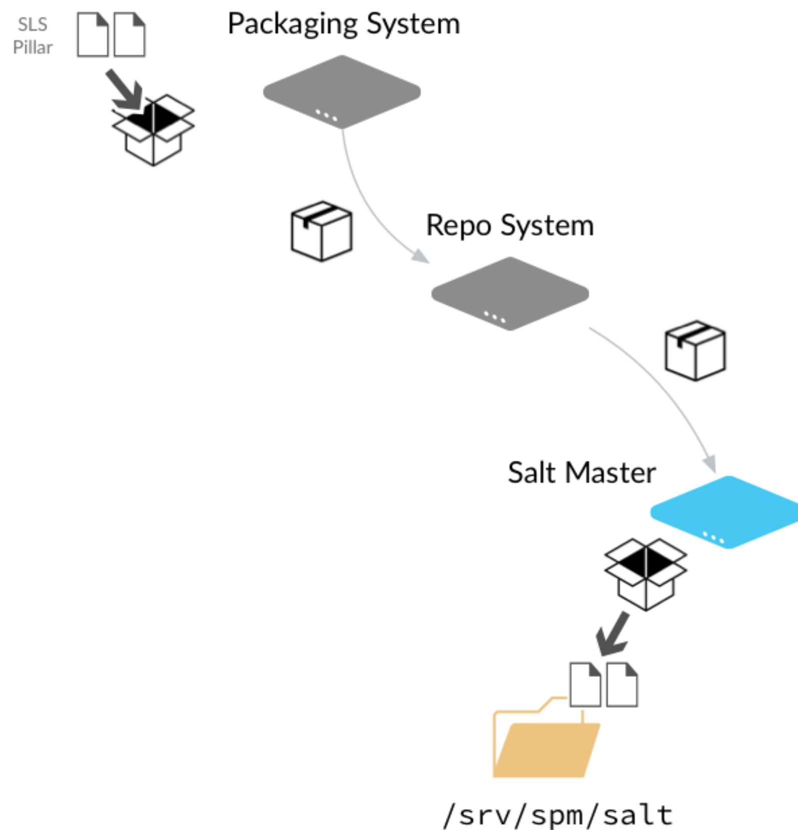
```
file_recv: True

fileserver_backend:
  - roots
  - minionfs

minionfs_mountpoint: salt://minionfs

minionfs_whitelist:
  - host04
  - web*
  - 'mail\d+\.domain\.tld'

minionfs_blacklist:
  - web21
```

Potential Concerns

- There is no access control in place to restrict which minions have access to files served up by *minionfs*. All minions will have access to these files.
- Unless the *minionfs_whitelist* and/or *minionfs_blacklist* config options are used, all minions which push files to the master will have their files made available via *minionfs*.

3.15 Salt Package Manager

The Salt Package Manager, or *SPM*, enables Salt formulas to be packaged to simplify distribution to Salt masters. The design of SPM was influenced by other existing packaging systems including RPM, Yum, and Pacman.



Note: The previous diagram shows each SPM component as a different system, but this is not required. You can build packages and host the SPM repo on a single Salt master if you'd like.

Packaging System

The packaging system is used to package the state, pillar, file templates, and other files used by your formula into a single file. After a formula package is created, it is copied to the Repository System where it is made available to Salt masters.

See *Building SPM Packages*

Repo System

The Repo system stores the SPM package and metadata files and makes them available to Salt masters via http(s), ftp, or file URLs. SPM repositories can be hosted on a Salt Master, a Salt Minion, or on another system.

See *Distributing SPM Packages*

Salt Master

SPM provides Salt master settings that let you configure the URL of one or more SPM repos. You can then quickly install packages that contain entire formulas to your Salt masters using SPM.

See *Installing SPM Packages*

Contents

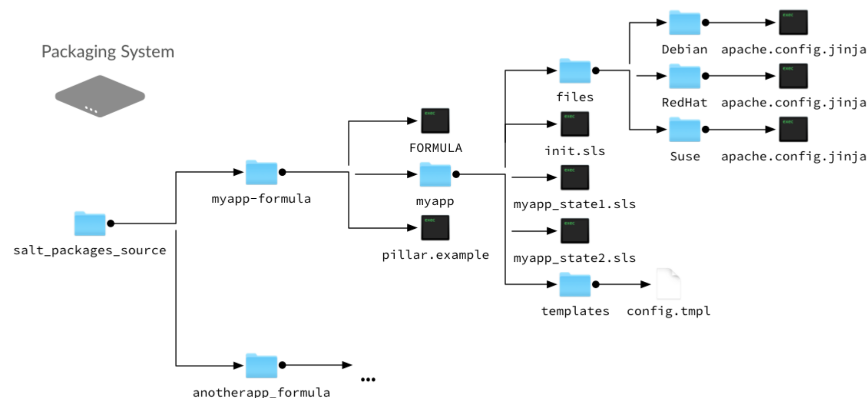
3.15.1 Building SPM Packages

The first step when using Salt Package Manager is to build packages for each of of the formulas that you want to distribute. Packages can be built on any system where you can install Salt.

Package Build Overview

To build a package, all state, pillar, jinja, and file templates used by your formula are assembled into a folder on the build system. These files can be cloned from a Git repository, such as those found at the saltstack-formulas organization on GitHub, or copied directly to the folder.

The following diagram demonstrates a typical formula layout on the build system:



In this example, all formula files are placed in a `myapp-formula` folder. This is the folder that is targeted by the `spm build` command when this package is built.

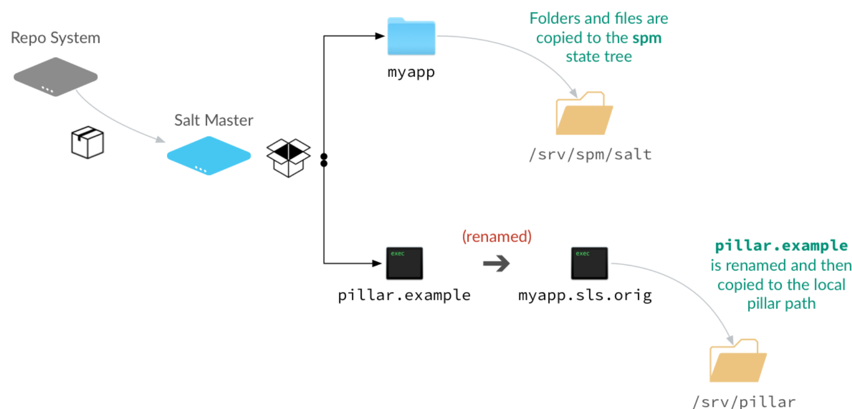
Within this folder, pillar data is placed in a `pillar.example` file at the root, and all state, jinja, and template files are placed within a subfolder that is named after the application being packaged. State files are typically contained within a subfolder, similar to how state files are organized in the state tree. Any non-pillar files in your package that are not contained in a subfolder are placed at the root of the spm state tree.

Additionally, a *FORMULA* file is created and placed in the root of the folder. This file contains package metadata that is used by SPM.

Package Installation Overview

When building packages, it is useful to know where files are installed on the Salt master. During installation, all files except `pillar.example` and `FORMULA` are copied directly to the spm state tree on the Salt master (located at `\srv\spm\salt`).

If a `pillar.example` file is present in the root, it is renamed to `<formula name>.sls.orig` and placed in the `pillar_path`.



Note: Even though the pillar data file is copied to the pillar root, you still need to manually assign this pillar data to systems using the pillar top file. This file can also be duplicated and renamed so the `.orig` version is left intact in case you need to restore it later.

Building an SPM Formula Package

1. Assemble formula files in a folder on the build system.
2. Create a *FORMULA* file and place it in the root of the package folder.
3. Run `spm build <folder name>`. The package is built and placed in the `/srv/spm_build` folder.

```
spm build /path/to/salt-packages-source/myapp-formula
```

4. Copy the `.spm` file to a folder on the *repository system*.

Types of Packages

SPM supports different types of packages. The function of each package is denoted by its name. For instance, packages which end in `-formula` are considered to be Salt States (the most common type of formula). Packages which end in `-conf` contain configuration which is to be placed in the `/etc/salt/` directory. Packages which do not contain one of these names are treated as if they have a `-formula` name.

formula

By default, most files from this type of package live in the `/srv/spm/salt/` directory. The exception is the `pillar.example` file, which will be renamed to `<package_name>.sls` and placed in the pillar directory (`/srv/spm/pillar/` by default).

reactor

By default, files from this type of package live in the `/srv/spm/reactor/` directory.

conf

The files in this type of package are configuration files for Salt, which normally live in the `/etc/salt/` directory. Configuration files for packages other than Salt can and should be handled with a Salt State (using a `formula` type of package).

Technical Information

Packages are built using BZ2-compressed tarballs. By default, the package database is stored using the `sqlite3` driver (see Loader Modules below).

Support for these are built into Python, and so no external dependencies are needed.

All other files belonging to SPM use YAML, for portability and ease of use and maintainability.

SPM-Specific Loader Modules

SPM was designed to behave like traditional package managers, which apply files to the filesystem and store package metadata in a local database. However, because modern infrastructures often extend beyond those use cases, certain parts of SPM have been broken out into their own set of modules.

Package Database

By default, the package database is stored using the `sqlite3` module. This module was chosen because support for SQLite3 is built into Python itself.

Please see the SPM Development Guide for information on creating new modules for package database management.

Package Files

By default, package files are installed using the `local` module. This module applies files to the local filesystem, on the machine that the package is installed on.

Please see the *SPM Development Guide* for information on creating new modules for package file management.

3.15.2 Distributing SPM Packages

SPM packages can be distributed to Salt masters over HTTP(S), FTP, or through the file system. The SPM repo can be hosted on any system where you can install Salt. Salt is installed so you can run the `spm create_repo` command when you update or add a package to the repo. SPM repos do not require the salt-master, salt-minion, or any other process running on the system.

Note: If you are hosting the SPM repo on a system where you can not or do not want to install Salt, you can run the `spm create_repo` command on the build system and then copy the packages and the generated `SPM-METADATA` file to the repo. You can also install SPM files *directly on a Salt master*, bypassing the repository completely.

Setting up a Package Repository

After packages are built, the generated SPM files are placed in the `srv/spm_build` folder.

Where you place the built SPM files on your repository server depends on how you plan to make them available to your Salt masters.

You can share the `srv/spm_build` folder on the network, or copy the files to your FTP or Web server.

Adding a Package to the repository

New packages are added by simply copying the SPM file to the repo folder, and then generating repo metadata.

Generate Repo Metadata

Each time you update or add an SPM package to your repository, issue an `spm create_repo` command:

```
spm create_repo /srv/spm_build
```

SPM generates the repository metadata for all of the packages in that directory and places it in an `SPM-METADATA` file at the folder root. This command is used even if repository metadata already exists in that directory.

3.15.3 Installing SPM Packages

SPM packages are installed to your Salt master, where they are available to Salt minions using all of Salt's package management functions.

Configuring Remote Repositories

Before SPM can use a repository, two things need to happen. First, the Salt master needs to know where the repository is through a configuration process. Then it needs to pull down the repository metadata.

Repository Configuration Files

Repositories are configured by adding each of them to the `/etc/salt/spm.repos.d/spm.repo` file on each Salt master. This file contains the name of the repository, and the link to the repository:

```
my_repo:
  url: https://spm.example.com/
```

For HTTP/HTTPS Basic authorization you can define credentials:

```
my_repo:
  url: https://spm.example.com/
  username: user
  password: pass
```

Beware of unauthorized access to this file, please set at least 0640 permissions for this configuration file:

The URL can use `http`, `https`, `ftp`, or `file`.

```
my_repo:
  url: file:///srv/spm_build
```

Updating Local Repository Metadata

After the repository is configured on the Salt master, repository metadata is downloaded using the `spm update_repo` command:

```
spm update_repo
```

Note: A file for each repo is placed in `/var/cache/salt/spm` on the Salt master after you run the *update_repo* command. If you add a repository and it does not seem to be showing up, check this path to verify that the repository was found.

Update File Roots

SPM packages are installed to the `srv/spm/salt` folder on your Salt master. This path needs to be added to the file roots on your Salt master manually.

```
file_roots:
  base:
    1. /srv/salt
    2. /srv/spm/salt
```

Restart the salt-master service after updating the `file_roots` setting.

Installing Packages

To install a package, use the `spm install` command:

```
spm install apache
```

Warning: Currently, SPM does not check to see if files are already in place before installing them. That means that existing files will be overwritten without warning.

Installing directly from an SPM file

You can also install SPM packages using a local SPM file using the `spm local install` command:

```
spm local install /srv/spm/apache-201506-1.spm
```

An SPM repository is not required when using *spm local install*.

Pillars

If an installed package includes Pillar data, be sure to target the installed pillar to the necessary systems using the pillar Top file.

Removing Packages

Packages may be removed after they are installed using the `spm remove` command.

```
spm remove apache
```

If files have been modified, they will not be removed. Empty directories will also be removed.

3.15.4 SPM Configuration

There are a number of options that are specific to SPM. They may be configured in the `master` configuration file, or in SPM's own `spm` configuration file (normally located at `/etc/salt/spm`). If configured in both places, the `spm` file takes precedence. In general, these values will not need to be changed from the defaults.

spm_logfile

Default: `/var/log/salt/spm`

Where SPM logs messages.

spm_repos_config

Default: `/etc/salt/spm.repos`

SPM repositories are configured with this file. There is also a directory which corresponds to it, which ends in `.d`. For instance, if the filename is `/etc/salt/spm.repos`, the directory will be `/etc/salt/spm.repos.d/`.

spm_cache_dir

Default: `/var/cache/salt/spm`

When SPM updates package repository metadata and downloads packaged, they will be placed in this directory. The package database, normally called `packages.db`, also lives in this directory.

spm_db

Default: `/var/cache/salt/spm/packages.db`

The location and name of the package database. This database stores the names of all of the SPM packages installed on the system, the files that belong to them, and the metadata for those files.

spm_build_dir

Default: `/srv/spm_build`

When packages are built, they will be placed in this directory.

spm_build_exclude

Default: ['.git']

When SPM builds a package, it normally adds all files in the formula directory to the package. Files listed here will be excluded from that package. This option requires a list to be specified.

```
spm_build_exclude:
  - .git
  - .svn
```

Types of Packages

SPM supports different types of formula packages. The function of each package is denoted by its name. For instance, packages which end in -formula are considered to be Salt States (the most common type of formula). Packages which end in -conf contain configuration which is to be placed in the /etc/salt/ directory. Packages which do not contain one of these names are treated as if they have a -formula name.

formula

By default, most files from this type of package live in the /srv/spm/salt/ directory. The exception is the pillar.example file, which will be renamed to <package_name>.sls and placed in the pillar directory (/srv/spm/pillar/ by default).

reactor

By default, files from this type of package live in the /srv/spm/reactor/ directory.

conf

The files in this type of package are configuration files for Salt, which normally live in the /etc/salt/ directory. Configuration files for packages other than Salt can and should be handled with a Salt State (using a formula type of package).

3.15.5 FORMULA File

In addition to the formula itself, a FORMULA file must exist which describes the package. An example of this file is:

```
name: apache
os: RedHat, Debian, Ubuntu, SUSE, FreeBSD
os_family: RedHat, Debian, Suse, FreeBSD
version: 201506
release: 2
summary: Formula for installing Apache
description: Formula for installing Apache
```

Required Fields

This file must contain at least the following fields:

name

The name of the package, as it will appear in the package filename, in the repository metadata, and the package database. Even if the source formula has `-formula` in its name, this name should probably not include that. For instance, when packaging the `apache-formula`, the name should be set to `apache`.

os

The value of the `os` grain that this formula supports. This is used to help users know which operating systems can support this package.

os_family

The value of the `os_family` grain that this formula supports. This is used to help users know which operating system families can support this package.

version

The version of the package. While it is up to the organization that manages this package, it is suggested that this version is specified in a `YYYYMM` format. For instance, if this version was released in June 2015, the package version should be `201506`. If multiple releases are made in a month, the `release` field should be used.

minimum_version

Minimum recommended version of Salt to use this formula. Not currently enforced.

release

This field refers primarily to a release of a version, but also to multiple versions within a month. In general, if a version has been made public, and immediate updates need to be made to it, this field should also be updated.

summary

A one-line description of the package.

description

A more detailed description of the package which can contain more than one line.

Optional Fields

The following fields may also be present.

top_level_dir

This field is optional, but highly recommended. If it is not specified, the package name will be used.

Formula repositories typically do not store `.sls` files in the root of the repository; instead they are stored in a subdirectory. For instance, an `apache-formula` repository would contain a directory called `apache`, which would contain an `init.sls`, plus a number of other related files. In this instance, the `top_level_dir` should be set to `apache`.

Files outside the `top_level_dir`, such as `README.rst`, `FORMULA`, and `LICENSE` will not be installed. The exceptions to this rule are files that are already treated specially, such as `pillar.example` and `_modules/`.

dependencies

A comma-separated list of packages that must be installed along with this package. When this package is installed, SPM will attempt to discover and install these packages as well. If it is unable to, then it will refuse to install this package.

This is useful for creating packages which tie together other packages. For instance, a package called wordpress-mariadb-apache would depend upon wordpress, mariadb, and apache.

optional

A comma-separated list of packages which are related to this package, but are neither required nor necessarily recommended. This list is displayed in an informational message when the package is installed to SPM.

recommended

A comma-separated list of optional packages that are recommended to be installed with the package. This list is displayed in an informational message when the package is installed to SPM.

files

A files section can be added, to specify a list of files to add to the SPM. Such a section might look like:

```
files:
  - _pillar
  - FORMULA
  - _runners
  - d|mymodule/index.rst
  - r|README.rst
```

When `files` are specified, then only those files will be added to the SPM, regardless of what other files exist in the directory. They will also be added in the order specified, which is useful if you have a need to lay down files in a specific order.

As can be seen in the example above, you may also tag files as being a specific type. This is done by pre-pending a filename with its type, followed by a pipe (`|`) character. The above example contains a document file and a readme. The available file types are:

- `c`: config file
- `d`: documentation file

- g: ghost file (i.e. the file contents are not included in the package payload)
- l: license file
- r: readme file
- s: SLS file
- m: Salt module

The first 5 of these types (c, d, g, l, r) will be placed in /usr/share/salt/spm/ by default. This can be changed by setting an spm_share_dir value in your /etc/salt/spm configuration file.

The last two types (s and m) are currently ignored, but they are reserved for future use.

Pre and Post States

It is possible to run Salt states before and after installing a package by using pre and post states. The following sections may be declared in a FORMULA:

- pre_local_state
- pre_tgt_state
- post_local_state
- post_tgt_state

Sections with pre in their name are evaluated before a package is installed and sections with post are evaluated after a package is installed. local states are evaluated before tgt states.

Each of these sections needs to be evaluated as text, rather than as YAML. Consider the following block:

```
pre_local_state: >
  echo test > /tmp/spmtest:
    cmd:
      - run
```

Note that this declaration uses > after pre_local_state. This is a YAML marker that marks the next multi-line block as text, including newlines. It is important to use this marker whenever declaring pre or post states, so that the text following it can be evaluated properly.

local States

local states are evaluated locally; this is analogous to issuing a state run using a salt-call --local command. These commands will be issued on the local machine running the spm command, whether that machine is a master or a minion.

local states do not require any special arguments, but they must still use the > marker to denote that the state is evaluated as text, not a data structure.

```
pre_local_state: >
  echo test > /tmp/spmtest:
    cmd:
      - run
```

tgt States

`tgt` states are issued against a remote target. This is analogous to issuing a state using the `salt` command. As such it requires that the machine that the `spm` command is running on is a master.

Because `tgt` states require that a target be specified, their code blocks are a little different. Consider the following state:

```
pre_tgt_state:
  tgt: '*'
  data: >
    echo test > /tmp/spmtest:
      cmd:
        - run
```

With `tgt` states, the state data is placed under a `data` section, inside the `*_tgt_state` code block. The target is of course specified as a `tgt` and you may also optionally specify a `tgt_type` (the default is `glob`).

You still need to use the > marker, but this time it follows the `data` line, rather than the `*_tgt_state` line.

Templating States

The reason that state data must be evaluated as text rather than a data structure is because that state data is first processed through the rendering engine, as it would be with a standard state run.

This means that you can use Jinja or any other supported renderer inside of Salt. All formula variables are available to the renderer, so you can reference `FORMULA` data inside your state if you need to:

```
pre_tgt_state:
  tgt: '*'
  data: >
     echo {{ name }} > /tmp/spmtest:
      cmd:
        - run
```

You may also declare your own variables inside the `FORMULA`. If SPM doesn't recognize them then it will ignore them, so there are no restrictions on variable names, outside of avoiding reserved words.

By default the renderer is set to `yaml_jinja`. You may change this by changing the `renderer` setting in the `FORMULA` itself.

Building a Package

Once a `FORMULA` file has been created, it is placed into the root of the formula that is to be turned into a package. The `spm build` command is used to turn that formula into a package:

```
spm build /path/to/saltstack-formulas/apache-formula
```

The resulting file will be placed in the build directory. By default this directory is located at `/srv/spm/`.

Loader Modules

When an execution module is placed in `<file_roots>/_modules/` on the master, it will automatically be synced to minions, the next time a sync operation takes place. Other modules are also propagated this way: state modules can be placed in `_states/`, and so on.

When SPM detects a file in a package which resides in one of these directories, that directory will be placed in `<file_roots>` instead of in the formula directory with the rest of the files.

Removing Packages

Packages may be removed once they are installed using the `spm remove` command.

```
spm remove apache
```

If files have been modified, they will not be removed. Empty directories will also be removed.

Technical Information

Packages are built using BZ2-compressed tarballs. By default, the package database is stored using the `sqlite3` driver (see Loader Modules below).

Support for these are built into Python, and so no external dependencies are needed.

All other files belonging to SPM use YAML, for portability and ease of use and maintainability.

SPM-Specific Loader Modules

SPM was designed to behave like traditional package managers, which apply files to the filesystem and store package metadata in a local database. However, because modern infrastructures often extend beyond those use cases, certain parts of SPM have been broken out into their own set of modules.

Package Database

By default, the package database is stored using the `sqlite3` module. This module was chosen because support for SQLite3 is built into Python itself.

Please see the SPM Development Guide for information on creating new modules for package database management.

Package Files

By default, package files are installed using the `local` module. This module applies files to the local filesystem, on the machine that the package is installed on.

Please see the *SPM Development Guide* for information on creating new modules for package file management.

Types of Packages

SPM supports different types of formula packages. The function of each package is denoted by its name. For instance, packages which end in `-formula` are considered to be Salt States (the most common type of formula). Packages which end in `-conf` contain configuration which is to be placed in the `/etc/salt/` directory. Packages which do not contain one of these names are treated as if they have a `-formula` name.

formula

By default, most files from this type of package live in the /srv/spm/salt/ directory. The exception is the pillar.example file, which will be renamed to <package_name>.sls and placed in the pillar directory (/srv/spm/pillar/ by default).

reactor

By default, files from this type of package live in the /srv/spm/reactor/ directory.

conf

The files in this type of package are configuration files for Salt, which normally live in the /etc/salt/ directory. Configuration files for packages other than Salt can and should be handled with a Salt State (using a formula type of package).

3.15.6 SPM Development Guide

This document discusses developing additional code for SPM.

SPM-Specific Loader Modules

SPM was designed to behave like traditional package managers, which apply files to the filesystem and store package metadata in a local database. However, because modern infrastructures often extend beyond those use cases, certain parts of SPM have been broken out into their own set of modules.

Each function that accepts arguments has a set of required and optional arguments. Take note that SPM will pass all arguments in, and therefore each function must accept each of those arguments. However, arguments that are marked as required are crucial to SPM's core functionality, while arguments that are marked as optional are provided as a benefit to the module, if it needs to use them.

Package Database

By default, the package database is stored using the sqlite3 module. This module was chosen because support for SQLite3 is built into Python itself.

Modules for managing the package database are stored in the salt/spm/pkgdb/ directory. A number of functions must exist to support database management.

init()

Get a database connection, and initialize the package database if necessary.

This function accepts no arguments. If a database is used which supports a connection object, then that connection object is returned. For instance, the sqlite3 module returns a connect() object from the sqlite3 library:

```
conn = sqlite3.connect(__opts__['spm_db'], isolation_level=None)
...
return conn
```

SPM itself will not use this connection object; it will be passed in as-is to the other functions in the module. Therefore, when you set up this object, make sure to do so in a way that is easily usable throughout the module.

info()

Return information for a package. This generally consists of the information that is stored in the `FORMULA` file in the package.

The arguments that are passed in, in order, are `package` (required) and `conn` (optional).

`package` is the name of the package, as specified in the `FORMULA`. `conn` is the connection object returned from `init()`.

list_files()

Return a list of files for an installed package. Only the filename should be returned, and no other information.

The arguments that are passed in, in order, are `package` (required) and `conn` (optional).

`package` is the name of the package, as specified in the `FORMULA`. `conn` is the connection object returned from `init()`.

register_pkg()

Register a package in the package database. Nothing is expected to be returned from this function.

The arguments that are passed in, in order, are `name` (required), `formula_def` (required), and `conn` (optional).

`name` is the name of the package, as specified in the `FORMULA`. `formula_def` is the contents of the `FORMULA` file, as a `dict`. `conn` is the connection object returned from `init()`.

register_file()

Register a file in the package database. Nothing is expected to be returned from this function.

The arguments that are passed in are `name` (required), `member` (required), `path` (required), `digest` (optional), and `conn` (optional).

`name` is the name of the package.

`member` is a `tarfile` object for the package file. It is included, because it contains most of the information for the file.

`path` is the location of the file on the local filesystem.

`digest` is the SHA1 checksum of the file.

`conn` is the connection object returned from `init()`.

unregister_pkg()

Unregister a package from the package database. This usually only involves removing the package's record from the database. Nothing is expected to be returned from this function.

The arguments that are passed in, in order, are `name` (required) and `conn` (optional).

name is the name of the package, as specified in the FORMULA. conn is the connection object returned from init().

unregister_file()

Unregister a package from the package database. This usually only involves removing the package's record from the database. Nothing is expected to be returned from this function.

The arguments that are passed in, in order, are name (required), pkg (optional) and conn (optional).

name is the path of the file, as it was installed on the filesystem.

pkg is the name of the package that the file belongs to.

conn is the connection object returned from init().

db_exists()

Check to see whether the package database already exists. This is the path to the package database file. This function will return True or False.

The only argument that is expected is db_, which is the package database file.

Package Files

By default, package files are installed using the local module. This module applies files to the local filesystem, on the machine that the package is installed on.

Modules for managing the package database are stored in the salt/spm/pkgfiles/ directory. A number of functions must exist to support file management.

init()

Initialize the installation location for the package files. Normally these will be directory paths, but other external destinations such as databases can be used. For this reason, this function will return a connection object, which can be a database object. However, in the default local module, this object is a dict containing the paths. This object will be passed into all other functions.

Three directories are used for the destinations: formula_path, pillar_path, and reactor_path.

formula_path is the location of most of the files that will be installed. The default is specific to the operating system, but is normally /srv/salt/.

pillar_path is the location that the pillar.example file will be installed to. The default is specific to the operating system, but is normally /srv/pillar/.

reactor_path is the location that reactor files will be installed to. The default is specific to the operating system, but is normally /srv/reactor/.

check_existing()

Check the filesystem for existing files. All files for the package will be checked, and if any are existing, then this function will normally state that SPM will refuse to install the package.

This function returns a list of the files that exist on the system.

The arguments that are passed into this function are, in order: package (required), pkg_files (required), formula_def (formula_def), and conn (optional).

package is the name of the package that is to be installed.

pkg_files is a list of the files to be checked.

formula_def is a copy of the information that is stored in the FORMULA file.

conn is the file connection object.

install_file()

Install a single file to the destination (normally on the filesystem). Nothing is expected to be returned from this function.

This function returns the final location that the file was installed to.

The arguments that are passed into this function are, in order, package (required), formula_tar (required), member (required), formula_def (required), and conn (optional).

package is the name of the package that is to be installed.

formula_tar is the tarfile object for the package. This is passed in so that the function can call formula_tar.extract() for the file.

member is the tarfile object which represents the individual file. This may be modified as necessary, before being passed into formula_tar.extract().

formula_def is a copy of the information from the FORMULA file.

conn is the file connection object.

remove_file()

Remove a single file from file system. Normally this will be little more than an os.remove(). Nothing is expected to be returned from this function.

The arguments that are passed into this function are, in order, path (required) and conn (optional).

path is the absolute path to the file to be removed.

conn is the file connection object.

hash_file()

Returns the hexdigest hash value of a file.

The arguments that are passed into this function are, in order, path (required), hashobj (required), and conn (optional).

path is the absolute path to the file.

hashobj is a reference to hashlib.sha1(), which is used to pull the hexdigest() for the file.

conn is the file connection object.

This function will not generally be more complex than:

```
def hash_file(path, hashobj, conn=None):
    with salt.utils.files.fopen(path, 'r') as f:
        hashobj.update(f.read())
        return hashobj.hexdigest()
```

path_exists()

Check to see whether the file already exists on the filesystem. Returns `True` or `False`.

This function expects a `path` argument, which is the absolute path to the file to be checked.

path_isdir()

Check to see whether the path specified is a directory. Returns `True` or `False`.

This function expects a `path` argument, which is the absolute path to be checked.

3.16 Storing Data in Other Databases

The SDB interface is designed to store and retrieve data that, unlike pillars and grains, is not necessarily minion-specific. The initial design goal was to allow passwords to be stored in a secure database, such as one managed by the keyring package, rather than as plain-text files. However, as a generic database interface, it could conceptually be used for a number of other purposes.

SDB was added to Salt in version 2014.7.0.

3.16.1 SDB Configuration

In order to use the SDB interface, a configuration profile must be set up. To be available for master commands, such as runners, it needs to be configured in the master configuration. For modules executed on a minion, it can be set either in the minion configuration file, or as a pillar. The configuration stanza includes the name/ID that the profile will be referred to as, a `driver` setting, and any other arguments that are necessary for the SDB module that will be used. For instance, a profile called `mykeyring`, which uses the `system` service in the `keyring` module would look like:

```
mykeyring:
  driver: keyring
  service: system
```

It is recommended to keep the name of the profile simple, as it is used in the SDB URI as well.

3.16.2 SDB URIs

SDB is designed to make small database queries (hence the name, SDB) using a compact URL. This allows users to reference a database value quickly inside a number of Salt configuration areas, without a lot of overhead. The basic format of an SDB URI is:

```
sdb://<profile>/<args>
```

The profile refers to the configuration profile defined in either the master or the minion configuration file. The args are specific to the module referred to in the profile, but will typically only need to refer to the key of a key/value pair inside the database. This is because the profile itself should define as many other parameters as possible.

For example, a profile might be set up to reference credentials for a specific OpenStack account. The profile might look like:

```
kevinopenstack:
  driver: keyring
  service: salt.cloud.openstack.kevin
```

And the URI used to reference the password might look like:

```
sdb://kevinopenstack/password
```

3.16.3 Getting, Setting and Deleting SDB Values

Once an SDB driver is configured, you can use the `sdb` execution module to get, set and delete values from it. There are two functions that may appear in most SDB modules: `get`, `set` and `delete`.

Getting a value requires only the SDB URI to be specified. To retrieve a value from the `kevinopenstack` profile above, you would use:

```
salt-call sdb.get sdb://kevinopenstack/password
```

Setting a value uses the same URI as would be used to retrieve it, followed by the value as another argument.

```
salt-call sdb.set 'sdb://myvault/secret/salt/saltstack' 'super awesome'
```

Deleting values (if supported by the driver) is done pretty much the same way as getting them. Provided that you have a profile called `mykvstore` that uses a driver allowing to delete values you would delete a value as shown below:

```
salt-call sdb.delete 'sdb://mykvstore/foobar'
```

The `sdb.get`, `sdb.set` and `sdb.delete` functions are also available in the runner system:

```
salt-run sdb.get 'sdb://myvault/secret/salt/saltstack'
salt-run sdb.set 'sdb://myvault/secret/salt/saltstack' 'super awesome'
salt-run sdb.delete 'sdb://mykvstore/foobar'
```

3.16.4 Using SDB URIs in Files

SDB URIs can be used in both configuration files, and files that are processed by the renderer system (jinja, mako, etc.). In a configuration file (such as `/etc/salt/master`, `/etc/salt/minion`, `/etc/salt/cloud`, etc.), make an entry as usual, and set the value to the SDB URI. For instance:

```
mykey: sdb://myetcd/mykey
```

To retrieve this value using a module, the module in question must use the `config.get` function to retrieve configuration values. This would look something like:

```
mykey = __salt__['config.get']('mykey')
```

Templating renderers use a similar construct. To get the `mykey` value from above in Jinja, you would use:

```
{{ salt['config.get']('mykey') }}
```

When retrieving data from configuration files using `config.get`, the SDB URI need only appear in the configuration file itself.

If you would like to retrieve a key directly from SDB, you would call the `sdb.get` function directly, using the SDB URI. For instance, in Jinja:

```
{{ salt['sdb.get']('sdb://myetcd/mykey') }}
```

When writing Salt modules, it is not recommended to call `sdb.get` directly, as it requires the user to provide values in SDB, using a specific URI. Use `config.get` instead.

3.16.5 Writing SDB Modules

There is currently one function that MUST exist in any SDB module (`get()`), one that SHOULD exist (`set_()`) and one that MAY exist (`delete()`). If using a (`set_()`) function, a `__func_alias__` dictionary MUST be declared in the module as well:

```
__func_alias__ = {
    'set_': 'set',
}
```

This is because `set` is a Python built-in, and therefore functions should not be created which are called `set()`. The `__func_alias__` functionality is provided via Salt's loader interfaces, and allows legally-named functions to be referred to using names that would otherwise be unwise to use.

The `get()` function is required, as it will be called via functions in other areas of the code which make use of the `sdb://` URI. For example, the `config.get` function in the `config` execution module uses this function.

The `set_()` function may be provided, but is not required, as some sources may be read-only, or may be otherwise unwise to access via a URI (for instance, because of SQL injection attacks).

The `delete()` function may be provided as well, but is not required, as many sources may be read-only or restrict such operations.

A simple example of an SDB module is `salt/sdb/keyring_db.py`, as it provides basic examples of most, if not all, of the types of functionality that are available not only for SDB modules, but for Salt modules in general.

3.17 Running the Salt Master/Minion as an Unprivileged User

While the default setup runs the master and minion as the root user, some may consider it an extra measure of security to run the master as a non-root user. Keep in mind that doing so does not change the master's capability to access minions as the user they are running as. Due to this many feel that running the master as a non-root user does not grant any real security advantage which is why the master has remained as root by default.

Note: Some of Salt's operations cannot execute correctly when the master is not running as root, specifically the pam external auth system, as this system needs root access to check authentication.

As of Salt 0.9.10 it is possible to run Salt as a non-root user. This can be done by setting the *`user`* parameter in the master configuration file. and restarting the `salt-master` service.

The minion has it's own `user` parameter as well, but running the minion as an unprivileged user will keep it from making changes to things like users, installed packages, etc. unless access controls (sudo, etc.) are setup on the minion to permit the non-root user to make the needed changes.

In order to allow Salt to successfully run as a non-root user, ownership, and permissions need to be set such that the desired user can read from and write to the following directories (and their subdirectories, where applicable):

- /etc/salt
- /var/cache/salt
- /var/log/salt
- /var/run/salt

Ownership can be easily changed with `chown`, like so:

```
# chown -R user /etc/salt /var/cache/salt /var/log/salt /var/run/salt
```

Warning: Running either the master or minion with the `root_dir` parameter specified will affect these paths, as will setting options like `pki_dir`, `cachedir`, `log_file`, and other options that normally live in the above directories.

3.18 Using cron with Salt

The Salt Minion can initiate its own *highstate* using the `salt-call` command.

```
$ salt-call state.apply
```

This will cause the minion to check in with the master and ensure it is in the correct ``state''.

3.19 Use cron to initiate a highstate

If you would like the Salt Minion to regularly check in with the master you can use cron to run the `salt-call` command:

```
0 0 * * * salt-call state.apply
```

The above cron entry will run a *highstate* every day at midnight.

Note: When executing Salt using cron, keep in mind that the default PATH for cron may not include the path for any scripts or commands used by Salt, and it may be necessary to set the PATH accordingly in the crontab:

```
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/opt/bin

0 0 * * * salt-call state.apply
```

3.20 Hardening Salt

This topic contains tips you can use to secure and harden your Salt environment. How you best secure and harden your Salt environment depends heavily on how you use Salt, where you use Salt, how your team is structured, where you get data from, and what kinds of access (internal and external) you require.

3.20.1 General hardening tips

- Restrict who can directly log into your Salt master system.
- Use SSH keys secured with a passphrase to gain access to the Salt master system.
- Track and secure SSH keys and any other login credentials you and your team need to gain access to the Salt master system.
- Use a hardened bastion server or a VPN to restrict direct access to the Salt master from the internet.
- Don't expose the Salt master any more than what is required.
- Harden the system as you would with any high-priority target.
- Keep the system patched and up-to-date.
- Use tight firewall rules.

3.20.2 Salt hardening tips

- Subscribe to salt-users or salt-announce so you know when new Salt releases are available. Keep your systems up-to-date with the latest patches.
- Use Salt's Client *ACL system* to avoid having to give out root access in order to run Salt commands.
- Use Salt's Client *ACL system* to restrict which users can run what commands.
- Use *external Pillar* to pull data into Salt from external sources so that non-sysadmins (other teams, junior admins, developers, etc) can provide configuration data without needing access to the Salt master.
- Make heavy use of SLS files that are version-controlled and go through a peer-review/code-review process before they're deployed and run in production. This is good advice even for ``one-off'' CLI commands because it helps mitigate typos and mistakes.
- Use salt-api, SSL, and restrict authentication with the *external auth* system if you need to expose your Salt master to external services.
- Make use of Salt's event system and *reactor* to allow minions to signal the Salt master without requiring direct access.
- Run the `salt-master` daemon as non-root.
- Disable which modules are loaded onto minions with the *`disable_modules`* setting. (for example, disable the `cmd` module if it makes sense in your environment.)
- Look through the fully-commented sample *master* and *minion* config files. There are many options for securing an installation.
- Run *masterless-mode* minions on particularly sensitive minions. There is also *Salt SSH* or the `modules.sudo` if you need to further restrict a minion.

3.21 Security disclosure policy

email security@saltstack.com

gpg key ID 4EA0793D

gpg key fingerprint `8ABE 4EFC F0F4 B24B FF2A AF90 D570 F2D3 4EA0 793D`

gpg public key:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG/MacGPG2 v2.0.22 (Darwin)

mQINBFO15mMBEADa3CfQwk5ED9wAQ8fFDku277CegG3U1hVGdcxqKNvucblwoKCb
hRK6u9ihgaO9V9duV2glwgjytiBI/z6lyWqdaD37YXG/gTL+9Md+qdSDeaOa/9eg
7y+g4P+FvU9HWUlujRVlofUn5Dj/IZgUywbxwEybutuzvvFVTzsn+DFVwTH34Qoh
QIuNzQCSEz3Lhh8zq9LqkNy91ZZQO1ZIUrypafspH6GBHHcE8msBFgYiNBnVcUFH
u0r4j1Rav+621EtD5GZsOt05+NJI8pkaC/dDKjURcuiV6bhmeSpNzLaXUhwx6f29
Vhag5JhVGGNQxlRTxNEM86HEFp+4zJQ8m/wRDrGX5IAHsdESdhP+ljDVlAAX/ttP
/Ucl2fgpTnDKVHOA00E515Q87ZHv6awJ3GL1veqi8zfsLaag7rw1TuuHyGLOPkDt
t5PAjsS9R3KI7pGnhqI6bTOi591odUdgzUhZChWUUX1VStiIDi2jCvyoOOLMOGS5
AEYXuWYP7KgujZCDRaTNqRDdgPd93Mh9JI8UmkzXDUgijdzVpzPjYgFaWtyK8lsc
Fizqe3/Yzf9RCVX/lmRbiEH+ql/zSxcWlBQd17PKaL+TisQFXcmQzccYgAxFbj2r
QHp5ABEu9YjFme2Jzun7Mv9V4qo3JF5dmnUk31yupZeAOGZkirIsaWC3hwARAQAB
tDBTYWx0U3RhY2sgU2VjdXJpdHkgVGVhbSA8c2VjdXJpdHlAc2FsdHN0YWNrLmNv
bT6JAj4EEwECACgFAlO15mMCGwMFCQeGH4AGCwkIBwMCBhUIAgkKCwQWAgMBAh4B
AheAAAoJENVw8tNOoHk9z/MP/2vzY27fmVxU5X8joiiturjlgEqQw41IYEmWv1Bw
4WVXYCHP1yu/1MC1uuvOmOd5BlI8YO2C2oyW7d1B0NorguPtz55b7jabCElekVCh
h/H4ZVThiwqgPpthRv/2npXjIm7SLSs/kuaXo6Qy2JpszwDVFw+xCRVL0tH9KJxz
HuNBeVq7abWD5fzIWkmGM9hicG/R2D0RIlco1Q0VNKy8klG+pOFOW886KnwkSPc7
JUYp1oUlHsSlhTmkLEG54cyVzrTP/XuZuyMTdtyTc3mfgW0adneAL6MARtC5UB/h
q+v9dqMf4iD3wY6ctu8KWE8Vo5MUEsNNO9EA2dUR88LwFZ3ZnnXdQkizgR/Aa515
dm17vlNkSoomYCo84eN7GOTfxWcq+iXYSWcKWT4X+h/ra+LmNndQWQBRebVUtbKE
ZDwKmiQz/5LY5EhlWcuU4lVmMSFpWXt5FR/PtzgTdZAo9QKkBjcv97LYbXvsPI69
El1BLAg+m+1UpE1L7zJT1il6PqVyEFAWBxW46wXCCkGssFsvz2yRp0PDX8A6u4yq
rTkt09uYht1is61joLDJ/kq3+6k8gJWkDOW+2NMrmf+/qcdYCMYXmrtOpg/wF27W
GMNAkbdyzgeX/MbUBCGCMdzhevRuivOI5bu4vT5s3KdshG+yhzV45bapKRd5VN+1
mZRquQINBFO15mMBEAC5UuLii9ZLz6qHfIJp35IOW9U8SOf7QFhzXR7NZ3DmJsd3
f6Nb/habQFIHjm3K9wbpj+FvaW2oWRlFVvYdzjUq6c82GUUjW1dnqgUvFwdmM835
1n0YQ2TonmyaF882RvsRZrbJ65uvy7SQxlouXaAYOdqwLsPxBEOyOnMPSktW5V2U
IWyxsNP3sADchWIGq9p5D3Y/loyIMsS1dj+TjoQZOKSj7CuRT98+8yhGAY8YBEXu
9r3I9o6mDkuPpAljuMc8r09Im6az2egtK/szKt4Hy1bpSSBZU4W/XR7XwQNywmb3
wxjmYT6Od3Mwj0jtzc3gQiH8hcEy3+BO+NNmyzFVyIwOLziwjmEcw62S57wYKUVn
HD2nglMsQa8Ve0e6ABBMEY7zGEGStva59rfgeh0jUMJiccGiUDTMs0tdkC6knYKb
u/fdRqNYFoNuDcSeLEw4DdCuP01l2W4yY+fiK6hAcL25amjzc+yYo9eaaqTn6RAT
bzdhHQZdpAMxY+vNT0+NhP1Zo5gYBMR65Zp/VhFsf67ijb03FUtdw9N8dHwiR2m8
vVA8kO/gCD6wS2p9RdXqrJ9JhnHYWjiVuXR+f755ZAndyQfRtowMdQIoiXuJEXYw
6XN+/BX81gJaynJYc0uw0MnxWQX+A5m8HqEsbIFUXBYXPgbwXTm7c4IHGgXXdwAR
AQABiQIlBBgBAgAPBQJTteZjAhsMBQkHhh+AAAoJENVw8tNOoHk91rcQAIhxLv4g
duF/J1Cyf6Wixz4rqslBQ7DgNztdIUMjCThg3eB6pvIzY5d3DNROmwU5JvGP1rEw
hNiJhgBDFaB0J/y28uSci+orhKDTHb/cn30IxfuAuqrv9dujvmlgM7JUswOtLZhs
5FYGa6v1RORRWhUx2PQsF6ORg22QAaagc7OlaO3BXBoiE/FWsnEQCUsc7GnnPqi7
um45OJl/pJntsBUKvivEU20fj7j1UpjmeWz56NcjXoKtEvGh99gM5W2nSMLE3aPw
vcKhS4yRyLjOe19NfYbtID8m8oshUDji0XjQ1z5NdGcf2V1YNGHU5xyK6zwyGxgV
xZqaWnbhDTu1UnYBna8BiUobkuqclb4T9k2WjbrUSmTwKixokCOirFDZvqISkgmN
r6/g3w2TRi1l/LtbUciF0FN2pd7rj5mWrOBPEFYJmrB6SQeswWNhr5RIsXrQd/Ho
zvNm0HnUNEe6w5YBfA6sXQy8B0Zs6pcgLogkFB15TuHIIIpxIsVRv5z8SlEnB7HQ
Io9hZT58yjhekJuzVQB9loU0C/W0lzci/pXTt6fd9puYQe1DG37pSifRG6kfHxrR
if6nRyrfdTlawqbqdkoqFDmEybAM9/hv3BqriGahGGH/hgplNQbYoXfNwYMYaHuB
```

```
aSkJvrOQW8bpuAzgVyd7TyNFv+t1kLlfaRYJ
=wBTJ
-----END PGP PUBLIC KEY BLOCK-----
```


The SaltStack Security Team is available at security@saltstack.com for security-related bug reports or questions.

We request the disclosure of any security-related bugs or issues be reported non-publicly until such time as the issue can be resolved and a security-fix release can be prepared. At that time we will release the fix and make a public announcement with upgrade instructions and download locations.

3.21.1 Security response procedure

SaltStack takes security and the trust of our customers and users very seriously. Our disclosure policy is intended to resolve security issues as quickly and safely as is possible.

1. A security report sent to security@saltstack.com is assigned to a team member. This person is the primary contact for questions and will coordinate the fix, release, and announcement.
2. The reported issue is reproduced and confirmed. A list of affected projects and releases is made.
3. Fixes are implemented for all affected projects and releases that are actively supported. Back-ports of the fix are made to any old releases that are actively supported.
4. Packagers are notified via the salt-packagers mailing list that an issue was reported and resolved, and that an announcement is incoming.
5. A new release is created and pushed to all affected repositories. The release documentation provides a full description of the issue, plus any upgrade instructions or other relevant details.
6. An announcement is made to the salt-users and salt-announce mailing lists. The announcement contains a description of the issue and a link to the full release documentation and download locations.

3.21.2 Receiving security announcements

The fastest place to receive security announcements is via the salt-announce mailing list. This list is low-traffic.

3.22 Salt Transport

One of fundamental features of Salt is remote execution. Salt has two basic ``channels'' for communicating with minions. Each channel requires a client (minion) and a server (master) implementation to work within Salt. These pairs of channels will work together to implement the specific message passing required by the channel interface.

3.22.1 Pub Channel

The pub channel, or publish channel, is how a master sends a job (payload) to a minion. This is a basic pub/sub paradigm, which has specific targeting semantics. All data which goes across the publish system should be encrypted such that only members of the Salt cluster can decrypt the publishes.

3.22.2 Req Channel

The req channel is how the minions send data to the master. This interface is primarily used for fetching files and returning job returns. The req channels have two basic interfaces when talking to the master. `send` is the basic method that guarantees the message is encrypted at least so that only minions attached to the same master can read it-- but no guarantee of minion-master confidentiality, whereas the `crypted_transfer_decode_dictentry` method does guarantee minion-master confidentiality.

Zeromq Transport

Note: Zeromq is the current default transport within Salt

Zeromq is a messaging library with bindings into many languages. Zeromq implements a socket interface for message passing, with specific semantics for the socket type.

Pub Channel

The pub channel is implemented using zeromq's pub/sub sockets. By default we don't use zeromq's filtering, which means that all publish jobs are sent to all minions and filtered minion side. Zeromq does have publisher side filtering which can be enabled in salt using `zmq_filtering`.

Req Channel

The req channel is implemented using zeromq's req/rep sockets. These sockets enforce a send/recv pattern, which forces salt to serialize messages through these socket pairs. This means that although the interface is asynchronous on the minion we cannot send a second message until we have received the reply of the first message.

TCP Transport

The tcp transport is an implementation of Salt's channels using raw tcp sockets. Since this isn't using a pre-defined messaging library we will describe the wire protocol, message semantics, etc. in this document.

The tcp transport is enabled by changing the *`transport`* setting to `tcp` on each Salt minion and Salt master.

```
transport: tcp
```

Warning: We currently recommend that when using Syndics that all Masters and Minions use the same transport. We're investigating a report of an error when using mixed transport types at very heavy loads.

Wire Protocol

This implementation over TCP focuses on flexibility over absolute efficiency. This means we are okay to spend a couple of bytes of wire space for flexibility in the future. That being said, the wire framing is quite efficient and looks like:

```
msgpack({'head': SOMEHEADER, 'body': SOMEBODY})
```

Since msgpack is an iterably parsed serialization, we can simply write the serialized payload to the wire. Within that payload we have two items ``head'' and ``body''. Head contains header information (such as ``message id''). The Body contains the actual message that we are sending. With this flexible wire protocol we can implement any message semantics that we'd like-- including multiplexed message passing on a single socket.

TLS Support

New in version 2016.11.1.

The TCP transport allows for the master/minion communication to be optionally wrapped in a TLS connection. Enabling this is simple, the master and minion need to be using the tcp connection, then the *ssl* option is enabled. The *ssl* option is passed as a dict and corresponds to the options passed to the Python *ssl.wrap_socket <https://docs.python.org/2/library/ssl.html#ssl.wrap_socket>* function.

A simple setup looks like this, on the Salt Master add the *ssl* option to the master configuration file:

```
ssl:
  keyfile: <path_to_keyfile>
  certfile: <path_to_certfile>
  ssl_version: PROTOCOL_TLSv1_2
```

The minimal *ssl* option in the minion configuration file looks like this:

```
ssl: True
# Versions below 2016.11.4:
ssl: {}
```

Specific options can be sent to the minion also, as defined in the Python *ssl.wrap_socket* function.

Note: While setting the ssl_version is not required, we recommend it. Some older versions of python do not support the latest TLS protocol and if this is the case for your version of python we strongly recommend upgrading your version of Python.

Crypto

The current implementation uses the same crypto as the `zeromq` transport.

Pub Channel

For the pub channel we send messages without ``message ids'' which the remote end interprets as a one-way send.

Note: As of today we send all publishes to all minions and rely on minion-side filtering.

Req Channel

For the req channel we send messages with a ``message id''. This ``message id'' allows us to multiplex messages across the socket.

The RAET Transport

Note: The RAET transport is in very early development, it is functional but no promises are yet made as to its reliability or security. As for reliability and security, the encryption used has been audited and our tests show that raet is reliable. With this said we are still conducting more security audits and pushing the reliability. This document outlines the encryption used in RAET

New in version 2014.7.0.

The Reliable Asynchronous Event Transport, or RAET, is an alternative transport medium developed specifically with Salt in mind. It has been developed to allow queuing to happen up on the application layer and comes with socket layer encryption. It also abstracts a great deal of control over the socket layer and makes it easy to bubble up errors and exceptions.

RAET also offers very powerful message routing capabilities, allowing for messages to be routed between processes on a single machine all the way up to processes on multiple machines. Messages can also be restricted, allowing processes to be sent messages of specific types from specific sources allowing for trust to be established.

Using RAET in Salt

Using RAET in Salt is easy, the main difference is that the core dependencies change, instead of needing pycrypto, M2Crypto, ZeroMQ, and PYZMQ, the packages libsodium, libnacl, ioflo, and raet are required. Encryption is handled very cleanly by libnacl, while the queueing and flow control is handled by ioflo. Distribution packages are forthcoming, but libsodium can be easily installed from source, or many distributions do ship packages for it. The libnacl and ioflo packages can be easily installed from pypi, distribution packages are in the works.

Once the new deps are installed the 2014.7 release or higher of Salt needs to be installed.

Once installed, modify the configuration files for the minion and master to set the transport to raet:

`/etc/salt/master`:

```
transport: raet
```

`/etc/salt/minion`:

```
transport: raet
```

Now start salt as it would normally be started, the minion will connect to the master and share long term keys, which can then in turn be managed via salt-key. Remote execution and salt states will function in the same way as with Salt over ZeroMQ.

Limitations

The 2014.7 release of RAET is not complete! The Syndic and Multi Master have not been completed yet and these are slated for completion in the 2015.5.0 release.

Also, Salt-Raet allows for more control over the client but these hooks have not been implemented yet, thereforre the client still uses the same system as the ZeroMQ client. This means that the extra reliability that RAET exposes has not yet been implemented in the CLI client.

Why?

Customer and User Request

Why make an alternative transport for Salt? There are many reasons, but the primary motivation came from customer requests, many large companies came with requests to run Salt over an alternative transport, the reasoning was varied, from performance and scaling improvements to licensing concerns. These customers have partnered with SaltStack to make RAET a reality.

More Capabilities

RAET has been designed to allow salt to have greater communication capabilities. It has been designed to allow for development into features which out ZeroMQ topologies can't match.

Many of the proposed features are still under development and will be announced as they enter proof of concept phases, but these features include *salt-fuse* - a filesystem over salt, *salt-vt* - a parallel api driven shell over the salt transport and many others.

RAET Reliability

RAET is reliable, hence the name (Reliable Asynchronous Event Transport).

The concern posed by some over RAET reliability is based on the fact that RAET uses UDP instead of TCP and UDP does not have built in reliability.

RAET itself implements the needed reliability layers that are not natively present in UDP, this allows RAET to dynamically optimize packet delivery in a way that keeps it both reliable and asynchronous.

RAET and ZeroMQ

When using RAET, ZeroMQ is not required. RAET is a complete networking replacement. It is noteworthy that RAET is not a ZeroMQ replacement in a general sense, the ZeroMQ constructs are not reproduced in RAET, but they are instead implemented in such a way that is specific to Salt's needs.

RAET is primarily an async communication layer over truly async connections, defaulting to UDP. ZeroMQ is over TCP and abstracts async constructs within the socket layer.

Salt is not dropping ZeroMQ support and has no immediate plans to do so.

Encryption

RAET uses Dan Bernstein's NACL encryption libraries and CurveCP handshake. The libnacl python binding binds to both libsodium and tweetnacl to execute the underlying cryptography. This allows us to completely rely on an externally developed cryptography system.

Programming Intro

Intro to RAET Programming

Note: This page is still under construction

The first thing to cover is that RAET does not present a socket api, it presents, and queueing api, all messages in RAET are made available to via queues. This is the single most differentiating factor with RAET vs other networking libraries, instead of making a socket, a stack is created. Instead of calling send() or recv(), messages are placed on the stack to be sent and messages that are received appear on the stack.

Different kinds of stacks are also available, currently two stacks exist, the UDP stack, and the UXD stack. The UDP stack is used to communicate over udp sockets, and the UXD stack is used to communicate over Unix Domain Sockets.

The UDP stack runs a context for communicating over networks, while the UXD stack has contexts for communicating between processes.

UDP Stack Messages

To create a UDP stack in RAET, simply create the stack, manage the queues, and process messages:

```
from salt.transport.road.raet import stacking
from salt.transport.road.raet import estating

udp_stack = stacking.StackUdp(ha=('127.0.0.1', 7870))
r_estate = estating.Estate(stack=stack, name='foo', ha=('192.168.42.42', 7870))
msg = {'hello': 'world'}
udp_stack.transmit(msg, udp_stack.estates[r_estate.name])
udp_stack.serviceAll()
```

3.23 Master Tops System

In 0.10.4 the *external_nodes* system was upgraded to allow for modular subsystems to be used to generate the top file data for a *highstate* run on the master.

The old *external_nodes* option has been removed. The master tops system provides a pluggable and extendable replacement for it, allowing for multiple different subsystems to provide top file data.

Using the new *master_tops* option is simple:

```
master_tops:
  ext_nodes: cobbler-external-nodes
```

for `Cobbler` or:

```
master_tops:
  reclass:
    inventory_base_uri: /etc/reclass
    classes_uri: roles
```

for `Reclass`.

```
master_tops:
  varstack: /path/to/the/config/file/varstack.yaml
```

for *Varstack*.

It's also possible to create custom master_tops modules. Simply place them into `salt://_tops` in the Salt fileserver and use the *saltutil.sync_tops* runner to sync them. If this runner function is not available, they can manually be placed into `extmods/tops`, relative to the master cachedir (in most cases the full path will be `/var/cache/salt/master/extmods/tops`).

Custom tops modules are written like any other execution module, see the source for the two modules above for examples of fully functional ones. Below is a bare-bones example:

/etc/salt/master:

```
master_tops:
  customtop: True
```

customtop.py: (custom master_tops module)

```
import logging
import sys
# Define the module's virtual name
__virtualname__ = 'customtop'

log = logging.getLogger(__name__)


def __virtual__():
    return __virtualname__


def top(**kwargs):
    log.debug('Calling top in customtop')
    return {'base': ['test']}
```

salt minion state.show_top should then display something like:

```
$ salt minion state.show_top

minion
    ----------
    base:
      - test
```

Note: If a master_tops module returns *top file* data for a given minion, it will be added to the states configured in the top file. It will *not* replace it altogether. The 2018.3.0 release adds additional functionality allowing a minion to treat master_tops as the single source of truth, irrespective of the top file.

3.24 Returners

By default the return values of the commands sent to the Salt minions are returned to the Salt master, however anything at all can be done with the results data.

By using a Salt returner, results data can be redirected to external data-stores for analysis and archival.

Returners pull their configuration values from the Salt minions. Returners are only configured once, which is generally at load time.

The returner interface allows the return data to be sent to any system that can receive data. This means that return data can be sent to a Redis server, a MongoDB server, a MySQL server, or any system.

See also:

Full list of builtin returners

3.24.1 Using Returners

All Salt commands will return the command data back to the master. Specifying returners will ensure that the data is _also_ sent to the specified returner interfaces.

Specifying what returners to use is done when the command is invoked:

```
salt '*' test.ping --return redis_return
```

This command will ensure that the redis_return returner is used.

It is also possible to specify multiple returners:

```
salt '*' test.ping --return mongo_return,redis_return,cassandra_return
```

In this scenario all three returners will be called and the data from the test.ping command will be sent out to the three named returners.

3.24.2 Writing a Returner

Returners are Salt modules that allow the redirection of results data to targets other than the Salt Master.

Returners Are Easy To Write!

Writing a Salt returner is straightforward.

A returner is a Python module containing at minimum a `returner` function. Other optional functions can be included to add support for *`master_job_cache`*, *Storing Job Results in an External System*, and *Event Returners*.

`returner` The `returner` function must accept a single argument. The argument contains return data from the called minion function. If the minion function `test.ping` is called, the value of the argument will be a dictionary. Run the following command from a Salt master to get a sample of the dictionary:

```
salt-call --local --metadata test.ping --out=pprint
```

```
import redis
import salt.utils.json

def returner(ret):
    '''
    Return information to a redis server
    '''
    # Get a redis connection
    serv = redis.Redis(
        host='redis-serv.example.com',
        port=6379,
        db='0')
    serv.sadd("%(id)s:jobs" % ret, ret['jid'])
    serv.set("%(jid)s:%(id)s" % ret, salt.utils.json.dumps(ret['return']))
```

```
    serv.sadd('jobs', ret['jid'])
    serv.sadd(ret['jid'], ret['id'])
```

The above example of a returner set to send the data to a Redis server serializes the data as JSON and sets it in redis.

Using Custom Returner Modules

Place custom returners in a `_returners/` directory within the *file_roots* specified by the master config file.

Custom returners are distributed when any of the following are called:

- `state.apply`
- *saltutil.sync_returners*
- *saltutil.sync_all*

Any custom returners which have been synced to a minion that are named the same as one of Salt's default set of returners will take the place of the default returner with the same name.

Naming the Returner

Note that a returner's default name is its filename (i.e. `foo.py` becomes returner `foo`), but that its name can be overridden by using a *__virtual__ function*. A good example of this can be found in the redis returner, which is named `redis_return.py` but is loaded as simply `redis`:

```
try:
    import redis
    HAS_REDIS = True
except ImportError:
    HAS_REDIS = False

__virtualname__ = 'redis'

def __virtual__():
    if not HAS_REDIS:
        return False
    return __virtualname__
```

Master Job Cache Support

master_job_cache, *Storing Job Results in an External System*, and *Event Returners*. Salt's *master_job_cache* allows returners to be used as a pluggable replacement for the *Default Job Cache*. In order to do so, a returner must implement the following functions:

Note: The code samples contained in this section were taken from the cassandra_cql returner.

prep_jid Ensures that job ids (jid) don't collide, unless passed_jid is provided.

`nochache` is an optional boolean that indicates if return data should be cached. `passed_jid` is a caller provided jid which should be returned unconditionally.

```
def prep_jid(nocache, passed_jid=None):  # pylint: disable=unused-argument
    '''
    Do any work necessary to prepare a JID, including sending a custom id
    '''
    return passed_jid if passed_jid is not None else salt.utils.jid.gen_jid()
```

save_load Save job information. The `jid` is generated by `prep_jid` and should be considered a unique identifier for the job. The jid, for example, could be used as the primary/unique key in a database. The `load` is what is returned to a Salt master by a minion. `minions` is a list of minions that the job was run against. The following code example stores the load as a JSON string in the salt.jids table.

```
import salt.utils.json

def save_load(jid, load, minions=None):
    '''
    Save the load to the specified jid id
    '''
    query = '''INSERT INTO salt.jids (
                 jid, load
               ) VALUES (
                 '{0}', '{1}'
               );'''.format(jid, salt.utils.json.dumps(load))

    # cassandra_cql.cql_query may raise a CommandExecutionError
    try:
        __salt__['cassandra_cql.cql_query'](query)
    except CommandExecutionError:
        log.critical('Could not save load in jids table.')
        raise
    except Exception as e:
        log.critical(
            'Unexpected error while inserting into jids: {0}'.format(e)
        )
        raise
```

get_load must accept a job id (jid) and return the job load stored by `save_load`, or an empty dictionary when not found.

```
def get_load(jid):
    '''
    Return the load data that marks a specified jid
    '''
    query = '''SELECT load FROM salt.jids WHERE jid = '{0}';'''.format(jid)

    ret = {}

    # cassandra_cql.cql_query may raise a CommandExecutionError
    try:
        data = __salt__['cassandra_cql.cql_query'](query)
        if data:
            load = data[0].get('load')
            if load:
                ret = json.loads(load)
    except CommandExecutionError:
        log.critical('Could not get load from jids table.')
        raise
    except Exception as e:
        log.critical('''Unexpected error while getting load from
```

```
        jids: {0}'''.format(str(e)))
        raise

    return ret
```

External Job Cache Support

Salt's *Storing Job Results in an External System* extends the `master_job_cache`. External Job Cache support requires the following functions in addition to what is required for Master Job Cache support:

get_jid Return a dictionary containing the information (load) returned by each minion when the specified job id was executed.

Sample:

```
{
    "local": {
        "master_minion": {
            "fun_args": [],
            "jid": "20150330121011408195",
            "return": true,
            "retcode": 0,
            "success": true,
            "cmd": "_return",
            "_stamp": "2015-03-30T12:10:12.708663",
            "fun": "test.ping",
            "id": "master_minion"
        }
    }
}
```

get_fun Return a dictionary of minions that called a given Salt function as their last function call.

Sample:

```
{
    "local": {
        "minion1": "test.ping",
        "minion3": "test.ping",
        "minion2": "test.ping"
    }
}
```

get_jids Return a list of all job ids.

Sample:

```
{
    "local": [
        "20150330121011408195",
        "20150330195922139916"
    ]
}
```

get_minions Returns a list of minions

Sample:

```
{
    "local": [
        "minion3",
        "minion2",
        "minion1",
        "master_minion"
    ]
}
```

Please refer to one or more of the existing returners (i.e. mysql, cassandra_cql) if you need further clarification.

Event Support

An `event_return` function must be added to the returner module to allow events to be logged from a master via the returner. A list of events are passed to the function by the master.

The following example was taken from the MySQL returner. In this example, each event is inserted into the salt_events table keyed on the event tag. The tag contains the jid and therefore is guaranteed to be unique.

```
import salt.utils.json

def event_return(events):
 '''
 Return event to mysql server

 Requires that configuration be enabled via 'event_return'
 option in master config.
 '''
 with _get_serv(events, commit=True) as cur:
     for event in events:
         tag = event.get('tag', '')
         data = event.get('data', '')
         sql = '''INSERT INTO `salt_events` (`tag`, `data`, `master_id` )
                  VALUES (%s, %s, %s)'''
         cur.execute(sql, (tag, salt.utils.json.dumps(data), __opts__['id']))
```

Testing the Returner

The `returner`, `prep_jid`, `save_load`, `get_load`, and `event_return` functions can be tested by configuring the *`master_job_cache`* and *Event Returners* in the master config file and submitting a job to `test.ping` each minion from the master.

Once you have successfully exercised the Master Job Cache functions, test the External Job Cache functions using the `ret` execution module.

```
salt-call ret.get_jids cassandra_cql --output=json
salt-call ret.get_fun cassandra_cql test.ping --output=json
salt-call ret.get_minions cassandra_cql --output=json
salt-call ret.get_jid cassandra_cql 20150330121011408195 --output=json
```

3.24.3 Event Returners

For maximum visibility into the history of events across a Salt infrastructure, all events seen by a salt master may be logged to one or more returners.

To enable event logging, set the `event_return` configuration option in the master config to the returner(s) which should be designated as the handler for event returns.

Note: Not all returners support event returns. Verify a returner has an `event_return()` function before using.

Note: On larger installations, many hundreds of events may be generated on a busy master every second. Be certain to closely monitor the storage of a given returner as Salt can easily overwhelm an underpowered server with thousands of returns.

3.24.4 Full List of Returners

returner modules

carbon_return	Take data from salt and ``return'' it into a carbon receiver
cassandra_cql_return	Return data to a cassandra server
cassandra_return	Return data to a Cassandra ColumnFamily
couchbase_return	Simple returner for Couchbase.
couchdb_return	Simple returner for CouchDB.
django_return	A returner that will inform a Django system that returns are available using Django's signal system.
elasticsearch_return	Return data to an elasticsearch server for indexing.
etcd_return	Return data to an etcd server or cluster
highstate_return	Return the results of a highstate (or any other state function that returns data in a compatible format) via an HTML email or HTML file.
hipchat_return	Return salt data via hipchat.
influxdb_return	Return data to an influxdb server.
kafka_return	Return data to a Kafka topic
librato_return	Salt returner to return highstate stats to Librato
local	The local returner is used to test the returner interface, it just prints the
local_cache	Return data to local job cache
mattermost_returner	Return salt data via mattermost
memcache_return	Return data to a memcache server
mongo_future_return	Return data to a mongodb server
mongo_return	Return data to a mongodb server
multi_returner	Read/Write multiple returners
mysql	Return data to a mysql server
nagios_return	Return salt data to Nagios
odbc	Return data to an ODBC compliant server.
pgjsonb	Return data to a PostgreSQL server with json data stored in Pg's jsonb data type
postgres	Return data to a postgresql server
postgres_local_cache	Use a postgresql server for the master job cache.
pushover_returner	Return salt data via pushover (http://www.pushover.net)
rawfile_json	Take data from salt and ``return'' it into a raw file containing the json, with one line per event.

Continued on next page

Table 3.2 -- continued from previous page

redis_return	Return data to a redis server
sentry_return	Salt returner that reports execution results back to sentry.
slack_returner	Return salt data via slack
sms_return	Return data by SMS.
smtp_return	Return salt data via email
splunk	Send json response data to Splunk via the HTTP Event Collector
sqlite3_return	Insert minion return data into a sqlite3 database
syslog_return	Return data to the host operating system's syslog facility
telegram_return	Return salt data via Telegram.
xmpp_return	Return salt data via xmpp
zabbix_return	Return salt data to Zabbix

salt.returners.carbon_return

Take data from salt and ``return'' it into a carbon receiver

Add the following configuration to the minion configuration file:

```
carbon.host: <server ip address>
carbon.port: 2003
```

Errors when trying to convert data to numbers may be ignored by setting `carbon.skip_on_error` to *True*:

```
carbon.skip_on_error: True
```

By default, data will be sent to carbon using the plaintext protocol. To use the pickle protocol, set `carbon.mode` to `pickle`:

```
carbon.mode: pickle
```

You can also specify the pattern used for the metric base path (except for virt modules metrics):
carbon.metric_base_pattern: carbon.[minion_id].[module].[function]

These tokens can used : [module]: salt module [function]: salt function [minion_id]: minion id

Default is : carbon.metric_base_pattern: [module].[function].[minion_id]

Carbon settings may also be configured as:

```
carbon:
  host: <server IP or hostname>
  port: <carbon port>
  skip_on_error: True
  mode: (pickle|text)
  metric_base_pattern: <pattern> | [module].[function].[minion_id]
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.carbon:
  host: <server IP or hostname>
  port: <carbon port>
  skip_on_error: True
  mode: (pickle|text)
```

To use the carbon returner, append `--return carbon' to the salt command.

```
salt '*' test.ping --return carbon
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return carbon --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return carbon --return_kwargs '{"skip_on_error": False}'
```

salt.returners.carbon_return.**event_return**(*events*)
: Return event data to remote carbon server

  Provide a list of events to be stored in carbon

salt.returners.carbon_return.**prep_jid**(*nocache=False, passed_jid=None*)
: Do any work necessary to prepare a JID, including sending a custom id

salt.returners.carbon_return.**returner**(*ret*)
: Return data to a remote carbon server using the text metric protocol

  Each metric will look like:

  ```
  [module].[function].[minion_id].[metric path [...]].[metric name]
  ```

salt.returners.cassandra_cql_return

Return data to a cassandra server

New in version 2015.5.0.

maintainer Corin Kochenower<ckochenower@saltstack.com>

maturity new as of 2015.2

depends salt.modules.cassandra_cql

depends DataStax Python Driver for Apache Cassandra https://github.com/datastax/python-driver pip install cassandra-driver

platform all

configuration To enable this returner, the minion will need the DataStax Python Driver for Apache Cassandra (https://github.com/datastax/python-driver) installed and the following values configured in the minion or master config. The list of cluster IPs must include at least one cassandra node IP address. No assumption or default will be used for the cluster IPs. The cluster IPs will be tried in the order listed. The port, username, and password values shown below will be the assumed defaults if you do not provide values.:

```
cassandra:
  cluster:
    - 192.168.50.11
    - 192.168.50.12
    - 192.168.50.13
```

```
port: 9042
username: salt
password: salt
```

Use the following cassandra database schema:

```
CREATE KEYSPACE IF NOT EXISTS salt
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' :
↪1};

CREATE USER IF NOT EXISTS salt WITH PASSWORD 'salt' NOSUPERUSER;

GRANT ALL ON KEYSPACE salt TO salt;

USE salt;

CREATE TABLE IF NOT EXISTS salt.salt_returns (
    jid text,
    minion_id text,
    fun text,
    alter_time timestamp,
    full_ret text,
    return text,
    success boolean,
    PRIMARY KEY (jid, minion_id, fun)
) WITH CLUSTERING ORDER BY (minion_id ASC, fun ASC);
CREATE INDEX IF NOT EXISTS salt_returns_minion_id ON salt.salt_returns
↪(minion_id);
CREATE INDEX IF NOT EXISTS salt_returns_fun ON salt.salt_returns (fun);

CREATE TABLE IF NOT EXISTS salt.jids (
    jid text PRIMARY KEY,
    load text
);

CREATE TABLE IF NOT EXISTS salt.minions (
    minion_id text PRIMARY KEY,
    last_fun text
);
CREATE INDEX IF NOT EXISTS minions_last_fun ON salt.minions (last_fun);

CREATE TABLE IF NOT EXISTS salt.salt_events (
    id timeuuid,
    tag text,
    alter_time timestamp,
    data text,
    master_id text,
    PRIMARY KEY (id, tag)
) WITH CLUSTERING ORDER BY (tag ASC);
CREATE INDEX tag ON salt.salt_events (tag);
```

Required python modules: cassandra-driver

To use the cassandra returner, append `--return cassandra_cql' to the salt command. ex:

```
salt '*' test.ping --return_cql cassandra
```

Note: if your Cassandra instance has not been tuned much you may benefit from altering some timeouts in *cassan-*

dra.yaml like so:

```
# How long the coordinator should wait for read operations to complete
read_request_timeout_in_ms: 5000
# How long the coordinator should wait for seq or index scans to complete
range_request_timeout_in_ms: 20000
# How long the coordinator should wait for writes to complete
write_request_timeout_in_ms: 20000
# How long the coordinator should wait for counter writes to complete
counter_write_request_timeout_in_ms: 10000
# How long a coordinator should continue to retry a CAS operation
# that contends with other proposals for the same row
cas_contention_timeout_in_ms: 5000
# How long the coordinator should wait for truncates to complete
# (This can be much longer, because unless auto_snapshot is disabled
# we need to flush first so we can snapshot before removing the data.)
truncate_request_timeout_in_ms: 60000
# The default timeout for other, miscellaneous operations
request_timeout_in_ms: 20000
```

As always, your mileage may vary and your Cassandra cluster may have different needs. SaltStack has seen situations where these timeouts can resolve some stacktraces that appear to come from the Datastax Python driver.

`salt.returners.cassandra_cql_return.`**`event_return`**`(`*events*`)`
: Return event to one of potentially many clustered cassandra nodes

  Requires that configuration be enabled via `event_return' option in master config.

  Cassandra does not support an auto-increment feature due to the highly inefficient nature of creating a monotonically increasing number across all nodes in a distributed database. Each event will be assigned a uuid by the connecting client.

`salt.returners.cassandra_cql_return.`**`get_fun`**`(`*fun*`)`
: Return a dict of the last function called for all minions

`salt.returners.cassandra_cql_return.`**`get_jid`**`(`*jid*`)`
: Return the information returned when the specified job id was executed

`salt.returners.cassandra_cql_return.`**`get_jids`**`()`
: Return a list of all job ids

`salt.returners.cassandra_cql_return.`**`get_load`**`(`*jid*`)`
: Return the load data that marks a specified jid

`salt.returners.cassandra_cql_return.`**`get_minions`**`()`
: Return a list of minions

`salt.returners.cassandra_cql_return.`**`prep_jid`**`(`*nocache, passed_jid=None*`)`
: Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.cassandra_cql_return.`**`returner`**`(`*ret*`)`
: Return data to one of potentially many clustered cassandra nodes

`salt.returners.cassandra_cql_return.`**`save_load`**`(`*jid, load, minions=None*`)`
: Save the load to the specified jid id

salt.returners.cassandra_return

Return data to a Cassandra ColumnFamily

Here's an example Keyspace / ColumnFamily setup that works with this returner:

```
create keyspace salt;
use salt;
create column family returns
  with key_validation_class='UTF8Type'
  and comparator='UTF8Type'
  and default_validation_class='UTF8Type';
```

Required python modules: pycassa

To use the cassandra returner, append `--return cassandra' to the salt command. ex:

salt `*' test.ping --return cassandra

salt.returners.cassandra_return.**prep_jid**(*nocache=False*, *passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.cassandra_return.**returner**(*ret*)
Return data to a Cassandra ColumnFamily

salt.returners.couchbase_return

Simple returner for Couchbase. Optional configuration settings are listed below, along with sane defaults.

```
couchbase.host:   'salt'
couchbase.port:   8091
couchbase.bucket: 'salt'
couchbase.ttl: 24
couchbase.password: 'password'
couchbase.skip_verify_views: False
```

To use the couchbase returner, append `--return couchbase' to the salt command. ex:

```
salt '*' test.ping --return couchbase
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return couchbase --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return couchbase --return_kwargs '{"bucket": "another-salt"}'
```

All of the return data will be stored in documents as follows:

JID

load: load obj tgt_minions: list of minions targeted nocache: should we not cache the return data

JID/MINION_ID

return: return_data full_ret: full load of job return

salt.returners.couchbase_return.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.couchbase_return.**get_jids**()
Return a list of all job ids

salt.returners.couchbase_return.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.couchbase_return.**prep_jid**(*nocache=False*, *passed_jid=None*)
Return a job id and prepare the job id directory This is the function responsible for making sure jids don't collide (unless its passed a jid) So do what you have to do to make sure that stays the case

salt.returners.couchbase_return.**returner**(*load*)
Return data to couchbase bucket

salt.returners.couchbase_return.**save_load**(*jid*, *clear_load*, *minion=None*)
Save the load to the specified jid

salt.returners.couchbase_return.**save_minions**(*jid*, *minions*, *syndic_id=None*)
Save/update the minion list for a given jid. The syndic_id argument is included for API compatibility only.

salt.returners.couchdb_return

Simple returner for CouchDB. Optional configuration settings are listed below, along with sane defaults:

```
couchdb.db: 'salt'
couchdb.url: 'http://salt:5984/'
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.couchdb.db: 'salt'
alternative.couchdb.url: 'http://salt:5984/'
```

To use the couchdb returner, append `--return couchdb` to the salt command. Example:

```
salt '*' test.ping --return couchdb
```

To use the alternative configuration, append `--return_config alternative` to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return couchdb --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return couchdb --return_kwargs '{"db": "another-salt"}'
```

On concurrent database access

As this returner creates a couchdb document with the salt job id as document id and as only one document with a given id can exist in a given couchdb database, it is advised for most setups that every minion be configured to write to it own database (the value of `couchdb.db` may be suffixed with the minion id), otherwise multi-minion targeting can lead to losing output:

- the first returning minion is able to create a document in the database
- other minions fail with {'error': 'HTTP Error 409: Conflict'}

salt.returners.couchdb_return.**ensure_views**()
: This function makes sure that all the views that should exist in the design document do exist.

salt.returners.couchdb_return.**get_fun**(*fun*)
: Return a dict with key being minion and value being the job details of the last run of function `fun'.

salt.returners.couchdb_return.**get_jid**(*jid*)
: Get the document with a given JID.

salt.returners.couchdb_return.**get_jids**()
: List all the jobs that we have..

salt.returners.couchdb_return.**get_minions**()
: Return a list of minion identifiers from a request of the view.

salt.returners.couchdb_return.**get_valid_salt_views**()
: Returns a dict object of views that should be part of the salt design document.

salt.returners.couchdb_return.**prep_jid**(*nocache=False, passed_jid=None*)
: Do any work necessary to prepare a JID, including sending a custom id

salt.returners.couchdb_return.**returner**(*ret*)
: Take in the return and shove it into the couchdb database.

salt.returners.couchdb_return.**set_salt_view**()
: Helper function that sets the salt design document. Uses get_valid_salt_views and some hardcoded values.

salt.returners.django_return

A returner that will inform a Django system that returns are available using Django's signal system.

https://docs.djangoproject.com/en/dev/topics/signals/

It is up to the Django developer to register necessary handlers with the signals provided by this returner and process returns as necessary.

The easiest way to use signals is to import them from this returner directly and then use a decorator to register them.

An example Django module that registers a function called `returner_callback' with this module's `returner' function:

```
import salt.returners.django_return
from django.dispatch import receiver

@receiver(salt.returners.django_return, sender=returner)
def returner_callback(sender, ret):
    print('I received {0} from {1}'.format(ret, sender))
```

salt.returners.django_return.**prep_jid**(*nocache=False, passed_jid=None*)
: Do any work necessary to prepare a JID, including sending a custom ID

salt.returners.django_return.**returner**(*ret*)
: Signal a Django server that a return is available

salt.returners.django_return.**save_load**(*jid, load, minions=None*)
: Save the load to the specified jid

salt.returners.elasticsearch_return

Return data to an elasticsearch server for indexing.

maintainer Jurnell Cockhren <jurnell.cockhren@sophicware.com>, Arnold Bechtoldt <mail@arnoldbechtoldt.com>

maturity New

depends elasticsearch-py

platform all

To enable this returner the elasticsearch python client must be installed on the desired minions (all or some subset).

Please see documentation of `elasticsearch execution module` for a valid connection configuration.

> **Warning:** The index that you wish to store documents will be created by Elasticsearch automatically if doesn't exist yet. It is highly recommended to create predefined index templates with appropriate mapping(s) that will be used by Elasticsearch upon index creation. Otherwise you will have problems as described in #20826.

To use the returner per salt call:

```
salt '*' test.ping --return elasticsearch
```

In order to have the returner apply to all minions:

```
ext_job_cache: elasticsearch
```

Minion configuration:

debug_returner_payload': False Output the payload being posted to the log file in debug mode

doc_type: `default' Document type to use for normal return messages

functions_blacklist Optional list of functions that should not be returned to elasticsearch

index_date: False Use a dated index (e.g. <index>-2016.11.29)

master_event_index: `salt-master-event-cache' Index to use when returning master events

master_event_doc_type: `efault' Document type to use got master events

master_job_cache_index: `salt-master-job-cache' Index to use for master job cache

master_job_cache_doc_type: `default' Document type to use for master job cache

number_of_shards: 1 Number of shards to use for the indexes

number_of_replicas: 0 Number of replicas to use for the indexes

NOTE: The following options are valid for `state.apply', `state.sls' and `state.highstate' functions only.

states_count: False Count the number of states which succeeded or failed and return it in top-level item called `counts'. States reporting None (i.e. changes would be made but it ran in test mode) are counted as successes.

states_order_output: False Prefix the state UID (e.g. file_|-yum_configured_|-/etc/yum.conf_|-managed) with a zero-padded version of the `__run_num__' value to allow for easier sorting. Also store the state function (i.e. file.managed) into a new key `_func'. Change the index to be `<index>-ordered' (e.g. salt-state_apply-ordered).

states_single_index: **False** Store results for state.apply, state.sls and state.highstate in the salt-state_apply index (or -ordered/-<date>) indexes if enabled

```
elasticsearch:
    hosts:
      - "10.10.10.10:9200"
      - "10.10.10.11:9200"
      - "10.10.10.12:9200"
    index_date: True
    number_of_shards: 5
    number_of_replicas: 1
    debug_returner_payload: True
    states_count: True
    states_order_output: True
    states_single_index: True
    functions_blacklist:
      - test.ping
      - saltutil.find_job
```

`salt.returners.elasticsearch_return.`**`event_return`**(*events*)

Return events to Elasticsearch

Requires that the *event_return* configuration be set in master config.

`salt.returners.elasticsearch_return.`**`get_load`**(*jid*)

Return the load data that marks a specified jid

New in version 2015.8.1.

`salt.returners.elasticsearch_return.`**`prep_jid`**(*nocache=False*, *passed_jid=None*)

Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.elasticsearch_return.`**`returner`**(*ret*)

Process the return from Salt

`salt.returners.elasticsearch_return.`**`save_load`**(*jid*, *load*, *minions=None*)

Save the load to the specified jid id

New in version 2015.8.1.

salt.returners.etcd_return

Return data to an etcd server or cluster

depends

- python-etcd

In order to return to an etcd server, a profile should be created in the master configuration file:

```
my_etcd_config:
  etcd.host: 127.0.0.1
  etcd.port: 4001
```

It is technically possible to configure etcd without using a profile, but this is not considered to be a best practice, especially when multiple etcd servers or clusters are available.

```
etcd.host: 127.0.0.1
etcd.port: 4001
```

Additionally, two more options must be specified in the top-level configuration in order to use the etcd returner:

```
etcd.returner: my_etcd_config
etcd.returner_root: /salt/return
```

The `etcd.returner` option specifies which configuration profile to use. The `etcd.returner_root` option specifies the path inside etcd to use as the root of the returner system.

Once the etcd options are configured, the returner may be used:

CLI Example:

salt `*' test.ping --return etcd

A username and password can be set:

```
etcd.username: larry  # Optional; requires etcd.password to be set
etcd.password: 123pass  # Optional; requires etcd.username to be set
```

You can also set a TTL (time to live) value for the returner:

```
etcd.ttl: 5
```

Authentication with username and password, and ttl, currently requires the `master` branch of `python-etcd`.

You may also specify different roles for read and write operations. First, create the profiles as specified above. Then add:

```
etcd.returner_read_profile: my_etcd_read
etcd.returner_write_profile: my_etcd_write
```

`salt.returners.etcd_return.`**`get_fun`**`()`
Return a dict of the last function called for all minions

`salt.returners.etcd_return.`**`get_jid`**`(`*jid*`)`
Return the information returned when the specified job id was executed

`salt.returners.etcd_return.`**`get_jids`**`()`
Return a list of all job ids

`salt.returners.etcd_return.`**`get_load`**`(`*jid*`)`
Return the load data that marks a specified jid

`salt.returners.etcd_return.`**`get_minions`**`()`
Return a list of minions

`salt.returners.etcd_return.`**`prep_jid`**`(`*nocache=False, passed_jid=None*`)`
Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.etcd_return.`**`returner`**`(`*ret*`)`
Return data to an etcd server or cluster

`salt.returners.etcd_return.`**`save_load`**`(`*jid, load, minions=None*`)`
Save the load to the specified jid

salt.returners.highstate_return module

Return the results of a highstate (or any other state function that returns data in a compatible format) via an HTML email or HTML file.

New in version 2017.7.0.

Similar results can be achieved by using the smtp returner with a custom template, except an attempt at writing such a template for the complex data structure returned by highstate function had proven to be a challenge, not to mention that the smtp module doesn't support sending HTML mail at the moment.

The main goal of this returner was to produce an easy to read email similar to the output of highstate outputter used by the CLI.

This returner could be very useful during scheduled executions, but could also be useful for communicating the results of a manual execution.

Returner configuration is controlled in a standard fashion either via highstate group or an alternatively named group.

```
salt '*' state.highstate --return highstate
```

To use the alternative configuration, append `--return_config config-name'

```
salt '*' state.highstate --return highstate --return_config simple
```

Here is an example of what the configuration might look like:

```
simple.highstate:
  report_failures: True
  report_changes: True
  report_everything: False
  failure_function: pillar.items
  success_function: pillar.items
  report_format: html
  report_delivery: smtp
  smtp_success_subject: 'success minion {id} on host {host}'
  smtp_failure_subject: 'failure minion {id} on host {host}'
  smtp_server: smtp.example.com
  smtp_recipients: saltusers@example.com, devops@example.com
  smtp_sender: salt@example.com
```

The *report_failures*, *report_changes*, and *report_everything* flags provide filtering of the results. If you want an email to be sent every time, then *report_everything* is your choice. If you want to be notified only when changes were successfully made use *report_changes*. And *report_failures* will generate an email if there were failures.

The configuration allows you to run a salt module function in case of success (*success_function*) or failure (*failure_function*).

Any salt function, including ones defined in the _module folder of your salt repo, could be used here and its output will be displayed under the `extra' heading of the email.

Supported values for *report_format* are html, json, and yaml. The latter two are typically used for debugging purposes, but could be used for applying a template at some later stage.

The values for *report_delivery* are smtp or file. In case of file delivery the only other applicable option is *file_output*.

In case of smtp delivery, smtp_* options demonstrated by the example above could be used to customize the email.

As you might have noticed, the success and failure subjects contain {id} and {host} values. Any other grain name could be used. As opposed to using {{grains['id']}}, which will be rendered by the master and contain master's values at the time of pillar generation, these will contain minion values at the time of execution.

`salt.returners.highstate_return.`**`returner`**(*ret*)

Check highstate return information and possibly fire off an email or save a file.

salt.returners.hipchat_return

Return salt data via hipchat.

New in version 2015.5.0.

The following fields can be set in the minion conf file:

```
hipchat.room_id (required)
hipchat.api_key (required)
hipchat.api_version (required)
hipchat.api_url (optional)
hipchat.from_name (required)
hipchat.color (optional)
hipchat.notify (optional)
hipchat.profile (optional)
hipchat.url (optional)
```

Note: When using Hipchat's API v2, `api_key` needs to be assigned to the room with the ``Label'' set to what you would have been set in the hipchat.from_name field. The v2 API disregards the `from_name` in the data sent for the room notification and uses the Label assigned through the Hipchat control panel.

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
hipchat.room_id
hipchat.api_key
hipchat.api_version
hipchat.api_url
hipchat.from_name
```

Hipchat settings may also be configured as:

```
hipchat:
  room_id: RoomName
  api_url: https://hipchat.myteam.con
  api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  api_version: v1
  from_name: user@email.com

alternative.hipchat:
  room_id: RoomName
  api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  api_version: v1
  from_name: user@email.com

hipchat_profile:
  hipchat.api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  hipchat.api_version: v1
  hipchat.from_name: user@email.com

hipchat:
  profile: hipchat_profile
  room_id: RoomName

alternative.hipchat:
  profile: hipchat_profile
```

```
  room_id: RoomName

hipchat:
  room_id: RoomName
  api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  api_version: v1
  api_url: api.hipchat.com
  from_name: user@email.com
```

To use the HipChat returner, append `--return hipchat' to the salt command.

```
salt '*' test.ping --return hipchat
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return hipchat --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return hipchat --return_kwargs '{"room_id": "another-room"}'
```

salt.returners.hipchat_return.**event_return**(*events*)
Return event data to hipchat

salt.returners.hipchat_return.**returner**(*ret*)
Send an hipchat message with the return data from a job

salt.returners.influxdb_return

Return data to an influxdb server.

New in version 2015.8.0.

To enable this returner the minion will need the python client for influxdb installed and the following values configured in the minion or master config, these are the defaults:

```
influxdb.db: 'salt'
influxdb.user: 'salt'
influxdb.password: 'salt'
influxdb.host: 'localhost'
influxdb.port: 8086
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.influxdb.db: 'salt'
alternative.influxdb.user: 'salt'
alternative.influxdb.password: 'salt'
alternative.influxdb.host: 'localhost'
alternative.influxdb.port: 6379
```

To use the influxdb returner, append `--return influxdb' to the salt command.

```
salt '*' test.ping --return influxdb
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

```
salt '*' test.ping --return influxdb --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return influxdb --return_kwargs '{"db": "another-salt"}'
```

salt.returners.influxdb_return.**get_fun**(*fun*)
Return a dict of the last function called for all minions

salt.returners.influxdb_return.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.influxdb_return.**get_jids**()
Return a list of all job ids

salt.returners.influxdb_return.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.influxdb_return.**get_minions**()
Return a list of minions

salt.returners.influxdb_return.**prep_jid**(*nocache=False*, *passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.influxdb_return.**returner**(*ret*)
Return data to a influxdb data store

salt.returners.influxdb_return.**save_load**(*jid*, *load*, *minions=None*)
Save the load to the specified jid

salt.returners.kafka_return

Return data to a Kafka topic

maintainer Christer Edwards (christer.edwards@gmail.com)

maturity 0.1

depends kafka-python

platform all

To enable this returner install kafka-python and enable the following settings in the minion config:

returner.kafka.hostnames:

- ``server1''
- ``server2''
- ``server3''

returner.kafka.topic: `topic'

To use the kafka returner, append `--return kafka' to the Salt command, eg;

salt `*' test.ping --return kafka

`salt.returners.kafka_return.`**`returner`**`(`*ret*`)`
Return information to a Kafka server

salt.returners.librato_return

Salt returner to return highstate stats to Librato

To enable this returner the minion will need the Librato client importable on the Python path and the following values configured in the minion or master config.

The Librato python client can be found at: https://github.com/librato/python-librato

```
librato.email: example@librato.com
librato.api_token: abc12345def
```

This return supports multi-dimension metrics for Librato. To enable support for more metrics, the tags JSON object can be modified to include other tags.

Adding EC2 Tags example: If ec2_tags:region were desired within the tags for multi-dimension. The tags could be modified to include the ec2 tags. Multiple dimensions are added simply by adding more tags to the submission.

```
pillar_data = __salt__['pillar.raw']()
q.add(metric.name, value, tags={'Name': ret['id'],'Region': pillar_data['ec2_tags'][
↪'Name']]})
```

`salt.returners.librato_return.`**`returner`**`(`*ret*`)`
Parse the return data and return metrics to Librato.

salt.returners.local

The local returner is used to test the returner interface, it just prints the return data to the console to verify that it is being passed properly

To use the local returner, append `--return local' to the salt command. ex:

```
salt '*' test.ping --return local
```

`salt.returners.local.`**`event_return`**`(`*event*`)`
Print event return data to the terminal to verify functionality

`salt.returners.local.`**`returner`**`(`*ret*`)`
Print the return data to the terminal to verify functionality

salt.returners.local_cache

Return data to local job cache

`salt.returners.local_cache.`**`clean_old_jobs`**`()`
Clean out the old jobs from the job cache

`salt.returners.local_cache.`**`get_endtime`**`(`*jid*`)`
Retrieve the stored endtime for a given job

Returns False if no endtime is present

`salt.returners.local_cache.`**`get_jid`**`(`*jid*`)`
Return the information returned when the specified job id was executed

salt.returners.local_cache.**get_jids**()
Return a dict mapping all job ids to job information

salt.returners.local_cache.**get_jids_filter**(*count*, *filter_find_job=True*)
Return a list of all jobs information filtered by the given criteria. :param int count: show not more than the count of most recent jobs :param bool filter_find_jobs: filter out `saltutil.find_job' jobs

salt.returners.local_cache.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.local_cache.**load_reg**()
Load the register from msgpack files

salt.returners.local_cache.**prep_jid**(*nocache=False*, *passed_jid=None*, *recurse_count=0*)
Return a job id and prepare the job id directory.

This is the function responsible for making sure jids don't collide (unless it is passed a jid). So do what you have to do to make sure that stays the case

salt.returners.local_cache.**returner**(*load*)
Return data to the local job cache

salt.returners.local_cache.**save_load**(*jid*, *clear_load*, *minions=None*, *recurse_count=0*)
Save the load to the specified jid

minions argument is to provide a pre-computed list of matched minions for the job, for cases when this function can't compute that list itself (such as for salt-ssh)

salt.returners.local_cache.**save_minions**(*jid*, *minions*, *syndic_id=None*)
Save/update the serialized list of minions for a given job

salt.returners.local_cache.**save_reg**(*data*)
Save the register to msgpack files

salt.returners.local_cache.**update_endtime**(*jid*, *time*)
Update (or store) the end time for a given job

Endtime is stored as a plain text string

salt.returners.mattermost_returner module

Return salt data via mattermost

New in version 2017.7.0.

The following fields can be set in the minion conf file:

```
mattermost.hook (required)
mattermost.username (optional)
mattermost.channel (optional)
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
mattermost.channel
mattermost.hook
mattermost.username
```

mattermost settings may also be configured as:

```
mattermost:
  channel: RoomName
  hook: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  username: user
```

To use the mattermost returner, append `--return mattermost' to the salt command.

```
salt '*' test.ping --return mattermost
```

To override individual configuration items, append --return_kwargs `{`key:': `value'}' to the salt command.

```
salt '*' test.ping --return mattermost --return_kwargs '{'channel': '#random'}'
```

salt.returners.mattermost_returner.**event_return**(*events*)
Send the events to a mattermost room.

Parameters **events** -- List of events

Returns Boolean if messages were sent successfully.

salt.returners.mattermost_returner.**post_message**(*channel*, *message*, *username*, *api_url*, *hook*)
Send a message to a mattermost room.

Parameters

- **channel** -- The room name.
- **message** -- The message to send to the mattermost room.
- **username** -- Specify who the message is from.
- **hook** -- The mattermost hook, if not specified in the configuration.

Returns Boolean if message was sent successfully.

salt.returners.mattermost_returner.**returner**(*ret*)
Send an mattermost message with the data

salt.returners.memcache_return

Return data to a memcache server

To enable this returner the minion will need the python client for memcache installed and the following values configured in the minion or master config, these are the defaults.

```
memcache.host: 'localhost'
memcache.port: '11211'
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location.

```
alternative.memcache.host: 'localhost'
alternative.memcache.port: '11211'
```

python2-memcache uses `localhost' and `11211' as syntax on connection.

To use the memcache returner, append `--return memcache' to the salt command.

```
salt '*' test.ping --return memcache
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return memcache --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return memcache --return_kwargs '{"host": "hostname.domain.com"}'
```

salt.returners.memcache_return.**get_fun**(*fun*)
Return a dict of the last function called for all minions

salt.returners.memcache_return.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.memcache_return.**get_jids**()
Return a list of all job ids

salt.returners.memcache_return.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.memcache_return.**get_minions**()
Return a list of minions

salt.returners.memcache_return.**prep_jid**(*nocache=False*, *passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.memcache_return.**returner**(*ret*)
Return data to a memcache data store

salt.returners.memcache_return.**save_load**(*jid*, *load*, *minions=None*)
Save the load to the specified jid

salt.returners.mongo_future_return

Return data to a mongodb server

Required python modules: pymongo

This returner will send data from the minions to a MongoDB server. To configure the settings for your MongoDB server, add the following lines to the minion config files:

```
mongo.db: <database name>
mongo.host: <server ip address>
mongo.user: <MongoDB username>
mongo.password: <MongoDB user password>
mongo.port: 27017
```

You can also ask for indexes creation on the most common used fields, which should greatly improve performance. Indexes are not created by default.

```
mongo.indexes: true
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.mongo.db: <database name>
alternative.mongo.host: <server ip address>
alternative.mongo.user: <MongoDB username>
alternative.mongo.password: <MongoDB user password>
alternative.mongo.port: 27017
```

This mongo returner is being developed to replace the default mongodb returner in the future and should not be considered API stable yet.

To use the mongo returner, append `--return mongo' to the salt command.

```
salt '*' test.ping --return mongo
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return mongo --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return mongo --return_kwargs '{"db": "another-salt"}'
```

salt.returners.mongo_future_return.**event_return**(*events*)
Return events to Mongodb server

salt.returners.mongo_future_return.**get_fun**(*fun*)
Return the most recent jobs that have executed the named function

salt.returners.mongo_future_return.**get_jid**(*jid*)
Return the return information associated with a jid

salt.returners.mongo_future_return.**get_jids**()
Return a list of job ids

salt.returners.mongo_future_return.**get_load**(*jid*)
Return the load associated with a given job id

salt.returners.mongo_future_return.**get_minions**()
Return a list of minions

salt.returners.mongo_future_return.**prep_jid**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.mongo_future_return.**returner**(*ret*)
Return data to a mongodb server

salt.returners.mongo_future_return.**save_load**(*jid, load, minions=None*)
Save the load for a given job id

salt.returners.mongo_return

Return data to a mongodb server

Required python modules: pymongo

This returner will send data from the minions to a MongoDB server. To configure the settings for your MongoDB server, add the following lines to the minion config files.

```
mongo.db: <database name>
mongo.host: <server ip address>
mongo.user: <MongoDB username>
mongo.password: <MongoDB user password>
mongo.port: 27017
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location.

```
alternative.mongo.db: <database name>
alternative.mongo.host: <server ip address>
alternative.mongo.user: <MongoDB username>
alternative.mongo.password: <MongoDB user password>
alternative.mongo.port: 27017
```

To use the mongo returner, append `--return mongo' to the salt command.

```
salt '*' test.ping --return mongo_return
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return mongo_return --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return mongo --return_kwargs '{"db": "another-salt"}'
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return mongo --return_kwargs '{"db": "another-salt"}'
```

salt.returners.mongo_return.**get_fun**(*fun*)
: Return the most recent jobs that have executed the named function

salt.returners.mongo_return.**get_jid**(*jid*)
: Return the return information associated with a jid

salt.returners.mongo_return.**prep_jid**(*nocache=False, passed_jid=None*)
: Do any work necessary to prepare a JID, including sending a custom id

salt.returners.mongo_return.**returner**(*ret*)
: Return data to a mongodb server

salt.returners.multi_returner

Read/Write multiple returners

salt.returners.multi_returner.**clean_old_jobs**()
: Clean out the old jobs from all returners (if you have it)

salt.returners.multi_returner.**get_jid**(*jid*)
Merge the return data from all returners

salt.returners.multi_returner.**get_jids**()
Return all job data from all returners

salt.returners.multi_returner.**get_load**(*jid*)
Merge the load data from all returners

salt.returners.multi_returner.**prep_jid**(*nocache=False*, *passed_jid=None*)
Call both with prep_jid on all returners in multi_returner

TODO: finish this, what do do when you get different jids from 2 returners... since our jids are time based, this make this problem hard, because they aren't unique, meaning that we have to make sure that no one else got the jid and if they did we spin to get a new one, which means ``locking'' the jid in 2 returners is non-trivial

salt.returners.multi_returner.**returner**(*load*)
Write return to all returners in multi_returner

salt.returners.multi_returner.**save_load**(*jid*, *clear_load*, *minions=None*)
Write load to all returners in multi_returner

salt.returners.mysql

Return data to a mysql server

maintainer Dave Boucha <dave@saltstack.com>, Seth House <shouse@saltstack.com>

maturity mature

depends python-mysqldb

platform all

To enable this returner, the minion will need the python client for mysql installed and the following values configured in the minion or master config. These are the defaults:

```
mysql.host: 'salt'
mysql.user: 'salt'
mysql.pass: 'salt'
mysql.db: 'salt'
mysql.port: 3306
```

SSL is optional. The defaults are set to None. If you do not want to use SSL, either exclude these options or set them to None.

```
mysql.ssl_ca: None
mysql.ssl_cert: None
mysql.ssl_key: None
```

Alternative configuration values can be used by prefacing the configuration with *alternative.*. Any values not found in the alternative configuration will be pulled from the default location. As stated above, SSL configuration is optional. The following ssl options are simply for illustration purposes:

```
alternative.mysql.host: 'salt'
alternative.mysql.user: 'salt'
alternative.mysql.pass: 'salt'
alternative.mysql.db: 'salt'
alternative.mysql.port: 3306
alternative.mysql.ssl_ca: '/etc/pki/mysql/certs/localhost.pem'
```

```
alternative.mysql.ssl_cert: '/etc/pki/mysql/certs/localhost.crt'
alternative.mysql.ssl_key: '/etc/pki/mysql/certs/localhost.key'
```

Should you wish the returner data to be cleaned out every so often, set *keep_jobs* to the number of hours for the jobs to live in the tables. Setting it to *0* or leaving it unset will cause the data to stay in the tables.

Should you wish to archive jobs in a different table for later processing, set *archive_jobs* to True. Salt will create 3 archive tables

- *jids_archive*
- *salt_returns_archive*
- *salt_events_archive*

and move the contents of *jids*, *salt_returns*, and *salt_events* that are more than *keep_jobs* hours old to these tables.

Use the following mysql database schema:

```
CREATE DATABASE  `salt`
  DEFAULT CHARACTER SET utf8
  DEFAULT COLLATE utf8_general_ci;

USE `salt`;

--
-- Table structure for table `jids`
--

DROP TABLE IF EXISTS `jids`;
CREATE TABLE `jids` (
  `jid` varchar(255) NOT NULL,
  `load` mediumtext NOT NULL,
  UNIQUE KEY `jid` (`jid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
CREATE INDEX jid ON jids(jid) USING BTREE;

--
-- Table structure for table `salt_returns`
--

DROP TABLE IF EXISTS `salt_returns`;
CREATE TABLE `salt_returns` (
  `fun` varchar(50) NOT NULL,
  `jid` varchar(255) NOT NULL,
  `return` mediumtext NOT NULL,
  `id` varchar(255) NOT NULL,
  `success` varchar(10) NOT NULL,
  `full_ret` mediumtext NOT NULL,
  `alter_time` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  KEY `id` (`id`),
  KEY `jid` (`jid`),
  KEY `fun` (`fun`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

--
-- Table structure for table `salt_events`
--

DROP TABLE IF EXISTS `salt_events`;
```

```
CREATE TABLE `salt_events` (
`id` BIGINT NOT NULL AUTO_INCREMENT,
`tag` varchar(255) NOT NULL,
`data` mediumtext NOT NULL,
`alter_time` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
`master_id` varchar(255) NOT NULL,
PRIMARY KEY (`id`),
KEY `tag` (`tag`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Required python modules: MySQLdb

To use the mysql returner, append `--return mysql' to the salt command.

```
salt '*' test.ping --return mysql
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return mysql --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return mysql --return_kwargs '{"db": "another-salt"}'
```

salt.returners.mysql.**clean_old_jobs**()
Called in the master's event loop every loop_interval. Archives and/or deletes the events and job details from the database. :return:

salt.returners.mysql.**event_return**(*events*)
Return event to mysql server

Requires that configuration be enabled via `event_return' option in master config.

salt.returners.mysql.**get_fun**(*fun*)
Return a dict of the last function called for all minions

salt.returners.mysql.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.mysql.**get_jids**()
Return a list of all job ids

salt.returners.mysql.**get_jids_filter**(*count, filter_find_job=True*)
Return a list of all job ids :param int count: show not more than the count of most recent jobs :param bool filter_find_jobs: filter out `saltutil.find_job' jobs

salt.returners.mysql.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.mysql.**get_minions**()
Return a list of minions

salt.returners.mysql.**prep_jid**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.mysql.**returner**(*ret*)
Return data to a mysql server

`salt.returners.mysql.save_load(jid, load, minions=None)`
Save the load to the specified jid id

salt.returners.nagios_return

Return salt data to Nagios

The following fields can be set in the minion conf file:

```
nagios.url (required)
nagios.token (required)
nagios.service (optional)
nagios.check_type (optional)
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
nagios.url
nagios.token
nagios.service
```

Nagios settings may also be configured as:

```
  nagios:
      url: http://localhost/nrdp
      token: r4nd0mt0k3n
      service: service-check

  alternative.nagios:
      url: http://localhost/nrdp
      token: r4nd0mt0k3n
      service: another-service-check

To use the Nagios returner, append '--return nagios' to the salt command. ex:

.. code-block:: bash

  salt '*' test.ping --return nagios

To use the alternative configuration, append '--return_config alternative' to the salt
→command. ex:

  salt '*' test.ping --return nagios --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return nagios --return_kwargs '{"service": "service-name"}'
```

`salt.returners.nagios_return.returner(ret)`
Send a message to Nagios with the data

salt.returners.odbc

Return data to an ODBC compliant server. This driver was developed with Microsoft SQL Server in mind, but theoretically could be used to return data to any compliant ODBC database as long as there is a working ODBC driver for it on your minion platform.

maintainer

3. (a) Oldham (cr@saltstack.com)

maturity New

depends unixodbc, pyodbc, freetds (for SQL Server)

platform all

To enable this returner the minion will need

On Linux:

unixodbc (http://www.unixodbc.org) pyodbc (*pip install pyodbc*) The FreeTDS ODBC driver for SQL Server (http://www.freetds.org) or another compatible ODBC driver

On Windows:

TBD

unixODBC and FreeTDS need to be configured via /etc/odbcinst.ini and /etc/odbc.ini.

/etc/odbcinst.ini:

```
[TDS]
Description=TDS
Driver=/usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
```

(Note the above Driver line needs to point to the location of the FreeTDS shared library. This example is for Ubuntu 14.04.)

/etc/odbc.ini:

```
[TS]
Description = "Salt Returner"
Driver=TDS
Server = <your server ip or fqdn>
Port = 1433
Database = salt
Trace = No
```

Also you need the following values configured in the minion or master config. Configure as you see fit:

```
returner.odbc.dsn: 'TS'
returner.odbc.user: 'salt'
returner.odbc.passwd: 'salt'
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.returner.odbc.dsn: 'TS'
alternative.returner.odbc.user: 'salt'
alternative.returner.odbc.passwd: 'salt'
```

Running the following commands against Microsoft SQL Server in the desired database as the appropriate user should create the database tables correctly. Replace with equivalent SQL for other ODBC-compliant servers

```
  --
  -- Table structure for table 'jids'
  --

  if OBJECT_ID('dbo.jids', 'U') is not null
      DROP TABLE dbo.jids

  CREATE TABLE dbo.jids (
     jid   varchar(255) PRIMARY KEY,
     load  varchar(MAX) NOT NULL
   );

  --
  -- Table structure for table 'salt_returns'
  --
  IF OBJECT_ID('dbo.salt_returns', 'U') IS NOT NULL
      DROP TABLE dbo.salt_returns;

  CREATE TABLE dbo.salt_returns (
     added     datetime not null default (getdate()),
     fun       varchar(100) NOT NULL,
     jid       varchar(255) NOT NULL,
     retval    varchar(MAX) NOT NULL,
     id        varchar(255) NOT NULL,
     success   bit default(0) NOT NULL,
     full_ret  varchar(MAX)
   );

  CREATE INDEX salt_returns_added on dbo.salt_returns(added);
  CREATE INDEX salt_returns_id on dbo.salt_returns(id);
  CREATE INDEX salt_returns_jid on dbo.salt_returns(jid);
  CREATE INDEX salt_returns_fun on dbo.salt_returns(fun);

To use this returner, append '--return odbc' to the salt command.

.. code-block:: bash

  salt '*' status.diskusage --return odbc

To use the alternative configuration, append '--return_config alternative' to the salt
 ↪command.

.. versionadded:: 2015.5.0

.. code-block:: bash

  salt '*' test.ping --return odbc --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return odbc --return_kwargs '{"dsn": "dsn-name"}'
```

salt.returners.odbc.**get_fun**(*fun*)
  Return a dict of the last function called for all minions

`salt.returners.odbc.`**`get_jid`**(*jid*)
Return the information returned when the specified job id was executed

`salt.returners.odbc.`**`get_jids`**()
Return a list of all job ids

`salt.returners.odbc.`**`get_load`**(*jid*)
Return the load data that marks a specified jid

`salt.returners.odbc.`**`get_minions`**()
Return a list of minions

`salt.returners.odbc.`**`prep_jid`**(*nocache=False*, *passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.odbc.`**`returner`**(*ret*)
Return data to an odbc server

`salt.returners.odbc.`**`save_load`**(*jid*, *load*, *minions=None*)
Save the load to the specified jid id

salt.returners.pgjsonb

Return data to a PostgreSQL server with json data stored in Pg's jsonb data type

maintainer Dave Boucha <dave@saltstack.com>, Seth House <shouse@saltstack.com>, C. R. Oldham <cr@saltstack.com>

maturity Stable

depends python-psycopg2

platform all

Note: There are three PostgreSQL returners. Any can function as an external *master job cache*. but each has different features. SaltStack recommends `returners.pgjsonb` if you are working with a version of PostgreSQL that has the appropriate native binary JSON types. Otherwise, review `returners.postgres` and `returners.postgres_local_cache` to see which module best suits your particular needs.

To enable this returner, the minion will need the python client for PostgreSQL installed and the following values configured in the minion or master config. These are the defaults:

```
returner.pgjsonb.host: 'salt'
returner.pgjsonb.user: 'salt'
returner.pgjsonb.pass: 'salt'
returner.pgjsonb.db: 'salt'
returner.pgjsonb.port: 5432
```

SSL is optional. The defaults are set to None. If you do not want to use SSL, either exclude these options or set them to None.

```
returner.pgjsonb.sslmode: None
returner.pgjsonb.sslcert: None
returner.pgjsonb.sslkey: None
returner.pgjsonb.sslrootcert: None
returner.pgjsonb.sslcrl: None
```

New in version 2017.5.0.

Alternative configuration values can be used by prefacing the configuration with *alternative.*. Any values not found in the alternative configuration will be pulled from the default location. As stated above, SSL configuration is optional. The following ssl options are simply for illustration purposes:

```
alternative.pgjsonb.host: 'salt'
alternative.pgjsonb.user: 'salt'
alternative.pgjsonb.pass: 'salt'
alternative.pgjsonb.db: 'salt'
alternative.pgjsonb.port: 5432
alternative.pgjsonb.ssl_ca: '/etc/pki/mysql/certs/localhost.pem'
alternative.pgjsonb.ssl_cert: '/etc/pki/mysql/certs/localhost.crt'
alternative.pgjsonb.ssl_key: '/etc/pki/mysql/certs/localhost.key'
```

Use the following Pg database schema:

```
CREATE DATABASE  salt
  WITH ENCODING 'utf-8';

--
-- Table structure for table `jids`
--
DROP TABLE IF EXISTS jids;
CREATE TABLE jids (
   jid varchar(255) NOT NULL primary key,
   load jsonb NOT NULL
);
CREATE INDEX idx_jids_jsonb on jids
       USING gin (load)
       WITH (fastupdate=on);

--
-- Table structure for table `salt_returns`
--

DROP TABLE IF EXISTS salt_returns;
CREATE TABLE salt_returns (
  fun varchar(50) NOT NULL,
  jid varchar(255) NOT NULL,
  return jsonb NOT NULL,
  id varchar(255) NOT NULL,
  success varchar(10) NOT NULL,
  full_ret jsonb NOT NULL,
  alter_time TIMESTAMP WITH TIME ZONE DEFAULT NOW());

CREATE INDEX idx_salt_returns_id ON salt_returns (id);
CREATE INDEX idx_salt_returns_jid ON salt_returns (jid);
CREATE INDEX idx_salt_returns_fun ON salt_returns (fun);
CREATE INDEX idx_salt_returns_return ON salt_returns
    USING gin (return) with (fastupdate=on);
CREATE INDEX idx_salt_returns_full_ret ON salt_returns
    USING gin (full_ret) with (fastupdate=on);

--
-- Table structure for table `salt_events`
--

DROP TABLE IF EXISTS salt_events;
```

```
DROP SEQUENCE IF EXISTS seq_salt_events_id;
CREATE SEQUENCE seq_salt_events_id;
CREATE TABLE salt_events (
    id BIGINT NOT NULL UNIQUE DEFAULT nextval('seq_salt_events_id'),
    tag varchar(255) NOT NULL,
    data jsonb NOT NULL,
    alter_time TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    master_id varchar(255) NOT NULL);

CREATE INDEX idx_salt_events_tag on
    salt_events (tag);
CREATE INDEX idx_salt_events_data ON salt_events
    USING gin (data) with (fastupdate=on);
```

Required python modules: Psycopg2

To use this returner, append `--return pgjsonb' to the salt command.

```
salt '*' test.ping --return pgjsonb
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return pgjsonb --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return pgjsonb --return_kwargs '{"db": "another-salt"}'
```

salt.returners.pgjsonb.**event_return**(*events*)
Return event to Pg server

Requires that configuration be enabled via `event_return' option in master config.

salt.returners.pgjsonb.**get_fun**(*fun*)
Return a dict of the last function called for all minions

salt.returners.pgjsonb.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.pgjsonb.**get_jids**()
Return a list of all job ids

salt.returners.pgjsonb.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.pgjsonb.**get_minions**()
Return a list of minions

salt.returners.pgjsonb.**prep_jid**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.pgjsonb.**returner**(*ret*)
Return data to a Pg server

salt.returners.pgjsonb.**save_load**(*jid, load, minions=None*)
Save the load to the specified jid

salt.returners.postgres

Return data to a postgresql server

Note: There are three PostgreSQL returners. Any can function as an external *master job cache*. but each has different features. SaltStack recommends `returners.pgjsonb` if you are working with a version of PostgreSQL that has the appropriate native binary JSON types. Otherwise, review `returners.postgres` and `returners.postgres_local_cache` to see which module best suits your particular needs.

---

maintainer None

maturity New

depends psycopg2

platform all

To enable this returner the minion will need the psycopg2 installed and the following values configured in the minion or master config:

```
returner.postgres.host: 'salt'
returner.postgres.user: 'salt'
returner.postgres.passwd: 'salt'
returner.postgres.db: 'salt'
returner.postgres.port: 5432
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.returner.postgres.host: 'salt'
alternative.returner.postgres.user: 'salt'
alternative.returner.postgres.passwd: 'salt'
alternative.returner.postgres.db: 'salt'
alternative.returner.postgres.port: 5432
```

Running the following commands as the postgres user should create the database correctly:

```
psql << EOF
CREATE ROLE salt WITH PASSWORD 'salt';
CREATE DATABASE salt WITH OWNER salt;
EOF

psql -h localhost -U salt << EOF
--
-- Table structure for table 'jids'
--

DROP TABLE IF EXISTS jids;
CREATE TABLE jids (
  jid   varchar(20) PRIMARY KEY,
  load  text NOT NULL
);

--
-- Table structure for table 'salt_returns'
--

DROP TABLE IF EXISTS salt_returns;
```

```
CREATE TABLE salt_returns (
  fun       varchar(50) NOT NULL,
  jid       varchar(255) NOT NULL,
  return    text NOT NULL,
  full_ret  text,
  id        varchar(255) NOT NULL,
  success   varchar(10) NOT NULL,
  alter_time   TIMESTAMP WITH TIME ZONE DEFAULT now()
);

CREATE INDEX idx_salt_returns_id ON salt_returns (id);
CREATE INDEX idx_salt_returns_jid ON salt_returns (jid);
CREATE INDEX idx_salt_returns_fun ON salt_returns (fun);
CREATE INDEX idx_salt_returns_updated ON salt_returns (alter_time);

--
-- Table structure for table `salt_events`
--

DROP TABLE IF EXISTS salt_events;
DROP SEQUENCE IF EXISTS seq_salt_events_id;
CREATE SEQUENCE seq_salt_events_id;
CREATE TABLE salt_events (
    id BIGINT NOT NULL UNIQUE DEFAULT nextval('seq_salt_events_id'),
    tag varchar(255) NOT NULL,
    data text NOT NULL,
    alter_time TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    master_id varchar(255) NOT NULL
);

CREATE INDEX idx_salt_events_tag on salt_events (tag);

EOF
```

Required python modules: psycopg2

To use the postgres returner, append `--return postgres' to the salt command.

```
salt '*' test.ping --return postgres
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return postgres --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return postgres --return_kwargs '{"db": "another-salt"}'
```

salt.returners.postgres.**event_return**(*events*)

Return event to Pg server

Requires that configuration be enabled via `event_return' option in master config.

salt.returners.postgres.**get_fun**(*fun*)

Return a dict of the last function called for all minions

salt.returners.postgres.**get_jid**(*jid*)
Return the information returned when the specified job id was executed

salt.returners.postgres.**get_jids**()
Return a list of all job ids

salt.returners.postgres.**get_load**(*jid*)
Return the load data that marks a specified jid

salt.returners.postgres.**get_minions**()
Return a list of minions

salt.returners.postgres.**prep_jid**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.postgres.**returner**(*ret*)
Return data to a postgres server

salt.returners.postgres.**save_load**(*jid, load, minions=None*)
Save the load to the specified jid id

salt.returners.postgres_local_cache

Use a postgresql server for the master job cache. This helps the job cache to cope with scale.

Note: There are three PostgreSQL returners. Any can function as an external *master job cache*. but each has different features. SaltStack recommends *returners.pgjsonb* if you are working with a version of PostgreSQL that has the appropriate native binary JSON types. Otherwise, review *returners.postgres* and *returners.postgres_local_cache* to see which module best suits your particular needs.

maintainer gjredelinghuys@gmail.com

maturity Stable

depends psycopg2

platform all

To enable this returner the minion will need the psycopg2 installed and the following values configured in the master config:

```
master_job_cache: postgres_local_cache
master_job_cache.postgres.host: 'salt'
master_job_cache.postgres.user: 'salt'
master_job_cache.postgres.passwd: 'salt'
master_job_cache.postgres.db: 'salt'
master_job_cache.postgres.port: 5432
```

Running the following command as the postgres user should create the database correctly:

```
psql << EOF
CREATE ROLE salt WITH PASSWORD 'salt';
CREATE DATABASE salt WITH OWNER salt;
EOF
```

In case the postgres database is a remote host, you'll need this command also:

```
ALTER ROLE salt WITH LOGIN;
```

and then:

```
psql -h localhost -U salt << EOF
--
-- Table structure for table 'jids'
--

DROP TABLE IF EXISTS jids;
CREATE TABLE jids (
  jid   varchar(20) PRIMARY KEY,
  started TIMESTAMP WITH TIME ZONE DEFAULT now(),
  tgt_type text NOT NULL,
  cmd text NOT NULL,
  tgt text NOT NULL,
  kwargs text NOT NULL,
  ret text NOT NULL,
  username text NOT NULL,
  arg text NOT NULL,
  fun text NOT NULL
);

--
-- Table structure for table 'salt_returns'
--
-- note that 'success' must not have NOT NULL constraint, since
-- some functions don't provide it.

DROP TABLE IF EXISTS salt_returns;
CREATE TABLE salt_returns (
  added     TIMESTAMP WITH TIME ZONE DEFAULT now(),
  fun       text NOT NULL,
  jid       varchar(20) NOT NULL,
  return    text NOT NULL,
  id        text NOT NULL,
  success   boolean
);
CREATE INDEX ON salt_returns (added);
CREATE INDEX ON salt_returns (id);
CREATE INDEX ON salt_returns (jid);
CREATE INDEX ON salt_returns (fun);

DROP TABLE IF EXISTS salt_events;
CREATE TABLE salt_events (
  id SERIAL,
  tag text NOT NULL,
  data text NOT NULL,
  alter_time TIMESTAMP WITH TIME ZONE DEFAULT now(),
  master_id text NOT NULL
);
CREATE INDEX ON salt_events (tag);
CREATE INDEX ON salt_events (data);
CREATE INDEX ON salt_events (id);
CREATE INDEX ON salt_events (master_id);
EOF
```

Required python modules: psycopg2

salt.returners.postgres_local_cache.clean_old_jobs()
Clean out the old jobs from the job cache

salt.returners.postgres_local_cache.event_return(*events*)
Return event to a postgres server

Require that configuration be enabled via `event_return' option in master config.

salt.returners.postgres_local_cache.get_jid(*jid*)
Return the information returned when the specified job id was executed

salt.returners.postgres_local_cache.get_jids()
Return a list of all job ids For master job cache this also formats the output and returns a string

salt.returners.postgres_local_cache.get_load(*jid*)
Return the load data that marks a specified jid

salt.returners.postgres_local_cache.prep_jid(*nocache=False*, *passed_jid=None*)
Return a job id and prepare the job id directory This is the function responsible for making sure jids don't collide (unless its passed a jid). So do what you have to do to make sure that stays the case

salt.returners.postgres_local_cache.returner(*load*)
Return data to a postgres server

salt.returners.postgres_local_cache.save_load(*jid*, *clear_load*, *minions=None*)
Save the load to the specified jid id

salt.returners.pushover_returner

Return salt data via pushover (http://www.pushover.net)

New in version 2016.3.0.

The following fields can be set in the minion conf file:

```
pushover.user (required)
pushover.token (required)
pushover.title (optional)
pushover.device (optional)
pushover.priority (optional)
pushover.expire (optional)
pushover.retry (optional)
pushover.profile (optional)
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.pushover.user
alternative.pushover.token
alternative.pushover.title
alternative.pushover.device
alternative.pushover.priority
alternative.pushover.expire
alternative.pushover.retry
```

PushOver settings may also be configured as:

```
  pushover:
      user: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      token: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
      title: Salt Returner
      device: phone
      priority: -1
      expire: 3600
      retry: 5

  alternative.pushover:
      user: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      token: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      title: Salt Returner
      device: phone
      priority: 1
      expire: 4800
      retry: 2

  pushover_profile:
      pushover.token: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  pushover:
      user: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      profile: pushover_profile

  alternative.pushover:
      user: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      profile: pushover_profile

To use the PushOver returner, append '--return pushover' to the salt command. ex:

.. code-block:: bash

  salt '*' test.ping --return pushover

To use the alternative configuration, append '--return_config alternative' to the salt
↪command. ex:

  salt '*' test.ping --return pushover --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

```
salt '*' test.ping --return pushover --return_kwargs '{"title": "Salt is awesome!"}'
```

salt.returners.pushover_returner.**returner**(*ret*)
Send an PushOver message with the data

salt.returners.rawfile_json

Take data from salt and ``return'' it into a raw file containing the json, with one line per event.

Add the following to the minion or master configuration file.

```
rawfile_json.filename: <path_to_output_file>
```

Default is `/var/log/salt/events`.

Common use is to log all events on the master. This can generate a lot of noise, so you may wish to configure batch processing and/or configure the *event_return_whitelist* or *event_return_blacklist* to restrict the events that are written.

`salt.returners.rawfile_json.`**`event_return`**(*events*)
Write event data (return data and non-return data) to file on the master.

`salt.returners.rawfile_json.`**`returner`**(*ret*)
Write the return data to a file on the minion.

salt.returners.redis_return

Return data to a redis server

To enable this returner the minion will need the python client for redis installed and the following values configured in the minion or master config, these are the defaults:

```
redis.db: '0'
redis.host: 'salt'
redis.port: 6379
```

New in version 2018.3.1: Alternatively a UNIX socket can be specified by *unix_socket_path*:

```
redis.db: '0'
redis.unix_socket_path: /var/run/redis/redis.sock
```

Cluster Mode Example:

```
redis.db: '0'
redis.cluster_mode: true
redis.cluster.skip_full_coverage_check: true
redis.cluster.startup_nodes:
  - host: redis-member-1
    port: 6379
  - host: redis-member-2
    port: 6379
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.redis.db: '0'
alternative.redis.host: 'salt'
alternative.redis.port: 6379
```

To use the redis returner, append `--return redis' to the salt command.

```
salt '*' test.ping --return redis
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return redis --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return redis --return_kwargs '{"db": "another-salt"}'
```

Redis Cluster Mode Options:

cluster_mode: **`False`** Whether cluster_mode is enabled or not

cluster.startup_nodes: A list of host, port dictionaries pointing to cluster members. At least one is required but multiple nodes are better

```
cache.redis.cluster.startup_nodes
  - host: redis-member-1
    port: 6379
  - host: redis-member-2
    port: 6379
```

cluster.skip_full_coverage_check: **False** Some cluster providers restrict certain redis commands such as CONFIG for enhanced security. Set this option to true to skip checks that required advanced privileges.

Note: Most cloud hosted redis clusters will require this to be set to `True`

`salt.returners.redis_return.`**`clean_old_jobs`**`()`
Clean out minions's return data for old jobs.

Normally, hset `ret:<jid>' are saved with a TTL, and will eventually get cleaned by redis.But for jobs with some very late minion return, the corresponding hset's TTL will be refreshed to a too late timestamp, we'll do manually cleaning here.

`salt.returners.redis_return.`**`get_fun`**`(`*fun*`)`
Return a dict of the last function called for all minions

`salt.returners.redis_return.`**`get_jid`**`(`*jid*`)`
Return the information returned when the specified job id was executed

`salt.returners.redis_return.`**`get_jids`**`()`
Return a dict mapping all job ids to job information

`salt.returners.redis_return.`**`get_load`**`(`*jid*`)`
Return the load data that marks a specified jid

`salt.returners.redis_return.`**`get_minions`**`()`
Return a list of minions

`salt.returners.redis_return.`**`prep_jid`**`(`*nocache=False, passed_jid=None*`)`
Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.redis_return.`**`returner`**`(`*ret*`)`
Return data to a redis data store

`salt.returners.redis_return.`**`save_load`**`(`*jid, load, minions=None*`)`
Save the load to the specified jid

salt.returners.sentry_return

Salt returner that reports execution results back to sentry. The returner will inspect the payload to identify errors and flag them as such.

Pillar needs something like:

```
raven:
  servers:
    - http://192.168.1.1
    - https://sentry.example.com
  public_key: deadbeefdeadbeefdeadbeefdeadbeef
  secret_key: beefdeadbeefdeadbeefdeadbeefdead
```

```
  project: 1
  tags:
    - os
    - master
    - saltversion
    - cpuarch
```

or using a dsn:

```
raven:
  dsn: https://aaaa:bbbb@app.getsentry.com/12345
  tags:
    - os
    - master
    - saltversion
    - cpuarch
```

https://pypi.python.org/pypi/raven must be installed.

The pillar can be hidden on sentry return by setting hide_pillar: true.

The tags list (optional) specifies grains items that will be used as sentry tags, allowing tagging of events in the sentry ui.

To report only errors to sentry, set report_errors_only: true.

`salt.returners.sentry_return.`**`prep_jid`**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.sentry_return.`**`returner`**(*ret*)
Log outcome to sentry. The returner tries to identify errors and report them as such. All other messages will be reported at info level. Failed states will be appended as separate list for convenience.

salt.returners.slack_returner

Return salt data via slack

New in version 2015.5.0.

The following fields can be set in the minion conf file:

```
slack.channel (required)
slack.api_key (required)
slack.username (required)
slack.as_user (required to see the profile picture of your bot)
slack.profile (optional)
slack.changes(optional, only show changes and failed states)
slack.yaml_format(optional, format the json in yaml format)
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
slack.channel
slack.api_key
slack.username
slack.as_user
```

Slack settings may also be configured as:

```
slack:
    channel: RoomName
    api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    username: user
    as_user: true

alternative.slack:
    room_id: RoomName
    api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    from_name: user@email.com

slack_profile:
    slack.api_key: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    slack.from_name: user@email.com

slack:
    profile: slack_profile
    channel: RoomName

alternative.slack:
    profile: slack_profile
    channel: RoomName
```

To use the Slack returner, append `--return slack' to the salt command.

```
salt '*' test.ping --return slack
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

```
salt '*' test.ping --return slack --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return slack --return_kwargs '{"channel": "#random"}'
```

salt.returners.slack_returner.**returner**(*ret*)
    Send an slack message with the data

salt.returners.sms_return

Return data by SMS.

New in version 2015.5.0.

maintainer Damian Myerscough

maturity new

depends twilio

platform all

To enable this returner the minion will need the python twilio library installed and the following values configured in the minion or master config:

```
twilio.sid: 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
twilio.token: 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
twilio.to: '+1415XXXXXXX'
twilio.from: '+1650XXXXXXX'
```

To use the sms returner, append `--return sms' to the salt command.

```
salt '*' test.ping --return sms
```

`salt.returners.sms_return.`**`returner`**(*ret*)
Return a response in an SMS message

salt.returners.smtp_return

Return salt data via email

The following fields can be set in the minion conf file. Fields are optional unless noted otherwise.

- `from` (required) The name/address of the email sender.
- **`to` (required) The names/addresses of the email recipients**; comma-delimited. For example: `you@example.com,someoneelse@example.com`.
- `host` (required) The SMTP server hostname or address.
- `port` The SMTP server port; defaults to `25`.
- **`username` The username used to authenticate to the server. If specified a** password is also required. It is recommended but not required to also use TLS with this option.
- `password` The password used to authenticate to the server.
- `tls` Whether to secure the connection using TLS; defaults to `False`
- `subject` The email subject line.
- **`fields` Which fields from the returned data to include in the subject line** of the email; comma-delimited. For example: `id,fun`. Please note, *the subject line is not encrypted*.
- **`gpgowner` A user's `~/.gpg` directory. This must contain a gpg** public key matching the address the mail is sent to. If left unset, no encryption will be used. Requires **`python-gnupg`** to be installed.
- `template` The path to a file to be used as a template for the email body.
- **`renderer` A Salt renderer, or render-pipe, to use to render the email** template. Default `jinja`.

Below is an example of the above settings in a Salt Minion configuration file:

```
smtp.from: me@example.net
smtp.to: you@example.com
smtp.host: localhost
smtp.port: 1025
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location. For example:

```
alternative.smtp.username: saltdev
alternative.smtp.password: saltdev
alternative.smtp.tls: True
```

To use the SMTP returner, append `--return smtp' to the `salt` command.

```
salt '*' test.ping --return smtp
```

To use the alternative configuration, append `--return_config alternative' to the `salt` command.

New in version 2015.5.0.

```
salt '*' test.ping --return smtp --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the `salt` command.

New in version 2016.3.0.

```
salt '*' test.ping --return smtp --return_kwargs '{"to": "user@domain.com"}'
```

An easy way to test the SMTP returner is to use the development SMTP server built into Python. The command below will start a single-threaded SMTP server that prints any email it receives to the console.

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

New in version 2016.11.0.

It is possible to send emails with selected Salt events by configuring `event_return` option for Salt Master. For example:

```
event_return: smtp

event_return_whitelist:
  - salt/key

smtp.from: me@example.net
smtp.to: you@example.com
smtp.host: localhost
smtp.subject: 'Salt Master {{act}}ed key from Minion ID: {{id}}'
smtp.template: /srv/salt/templates/email.j2
```

Also you need to create additional file `/srv/salt/templates/email.j2` with email body template:

```
act: {{act}}
id: {{id}}
result: {{result}}
```

This configuration enables Salt Master to send an email when accepting or rejecting minions keys.

`salt.returners.smtp_return.`**`event_return`**`(`*events*`)`
Return event data via SMTP

`salt.returners.smtp_return.`**`prep_jid`**`(`*nocache=False, passed_jid=None*`)`
Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.smtp_return.`**`returner`**`(`*ret*`)`
Send an email with the data

salt.returners.splunk module

Send json response data to Splunk via the HTTP Event Collector Requires the following config values to be specified in config or pillar:

```
splunk_http_forwarder:
  token: <splunk_http_forwarder_token>
  indexer: <hostname/IP of Splunk indexer>
  sourcetype: <Destination sourcetype for data>
  index: <Destination index for data>
```

Run a test by using `salt-call test.ping --return splunk`

Written by Scott Pack (github.com/scottjpack)

`salt.returners.splunk.`**`returner`**(*ret*)
    Send a message to Splunk via the HTTP Event Collector

salt.returners.sqlite3

Insert minion return data into a sqlite3 database

maintainer Mickey Malone <mickey.malone@gmail.com>

maturity New

depends None

platform All

Sqlite3 is a serverless database that lives in a single file. In order to use this returner the database file must exist, have the appropriate schema defined, and be accessible to the user whom the minion process is running as. This returner requires the following values configured in the master or minion config:

```
sqlite3.database: /usr/lib/salt/salt.db
sqlite3.timeout: 5.0
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.sqlite3.database: /usr/lib/salt/salt.db
alternative.sqlite3.timeout: 5.0
```

Use the commands to create the sqlite3 database and tables:

```
sqlite3 /usr/lib/salt/salt.db << EOF
--
-- Table structure for table 'jids'
--

CREATE TABLE jids (
  jid TEXT PRIMARY KEY,
  load TEXT NOT NULL
  );

--
-- Table structure for table 'salt_returns'
--

CREATE TABLE salt_returns (
  fun TEXT KEY,
  jid TEXT KEY,
  id TEXT KEY,
  fun_args TEXT,
```

```
  date TEXT NOT NULL,
  full_ret TEXT NOT NULL,
  success TEXT NOT NULL
  );
EOF
```

To use the sqlite returner, append `--return sqlite3' to the salt command.

```
salt '*' test.ping --return sqlite3
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return sqlite3 --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return sqlite3 --return_kwargs '{"db": "/var/lib/salt/another-
↪salt.db"}'
```

salt.returners.sqlite3_return.**get_fun**(*fun*)
Return a dict of the last function called for all minions

salt.returners.sqlite3_return.**get_jid**(*jid*)
Return the information returned from a specified jid

salt.returners.sqlite3_return.**get_jids**()
Return a list of all job ids

salt.returners.sqlite3_return.**get_load**(*jid*)
Return the load from a specified jid

salt.returners.sqlite3_return.**get_minions**()
Return a list of minions

salt.returners.sqlite3_return.**prep_jid**(*nocache=False, passed_jid=None*)
Do any work necessary to prepare a JID, including sending a custom id

salt.returners.sqlite3_return.**returner**(*ret*)
Insert minion return data into the sqlite3 database

salt.returners.sqlite3_return.**save_load**(*jid, load, minions=None*)
Save the load to the specified jid

salt.returners.syslog_return

Return data to the host operating system's syslog facility

To use the syslog returner, append `--return syslog' to the salt command.

```
salt '*' test.ping --return syslog
```

The following fields can be set in the minion conf file:

```
syslog.level (optional, Default: LOG_INFO)
syslog.facility (optional, Default: LOG_USER)
syslog.tag (optional, Default: salt-minion)
syslog.options (list, optional, Default: [])
```

Available levels, facilities, and options can be found in the `syslog` docs for your python version.

Note: The default tag comes from `sys.argv[0]` which is usually ``salt-minion'' but could be different based on the specific environment.

Configuration example:

```
syslog.level: 'LOG_ERR'
syslog.facility: 'LOG_DAEMON'
syslog.tag: 'mysalt'
syslog.options:
  - LOG_PID
```

Of course you can also nest the options:

```
syslog:
  level: 'LOG_ERR'
  facility: 'LOG_DAEMON'
  tag: 'mysalt'
  options:
    - LOG_PID
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
alternative.syslog.level: 'LOG_WARN'
alternative.syslog.facility: 'LOG_NEWS'
```

To use the alternative configuration, append `--return_config alternative` to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return syslog --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return syslog --return_kwargs '{"level": "LOG_DEBUG"}'
```

Note: Syslog server implementations may have limits on the maximum record size received by the client. This may lead to job return data being truncated in the syslog server's logs. For example, for rsyslog on RHEL-based systems, the default maximum record size is approximately 2KB (which return data can easily exceed). This is configurable in rsyslog.conf via the $MaxMessageSize config parameter. Please consult your syslog implmentation's documentation to determine how to adjust this limit.

`salt.returners.syslog_return.`**`prep_jid`**(*nocache=False, passed_jid=None*)

Do any work necessary to prepare a JID, including sending a custom id

`salt.returners.syslog_return.`**`returner`**`(`*ret*`)`
Return data to the local syslog

salt.returners.telegram_return

Return salt data via Telegram.

The following fields can be set in the minion conf file:

```
telegram.chat_id (required)
telegram.token (required)
```

Telegram settings may also be configured as:

```
telegram:
  chat_id: 000000000
  token: 000000000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

To use the Telegram return, append `--return telegram' to the salt command.

```
salt '*' test.ping --return telegram
```

`salt.returners.telegram_return.`**`returner`**`(`*ret*`)`
Send a Telegram message with the data.

Parameters **`ret`** -- The data to be sent.

Returns Boolean if message was sent successfully.

salt.returners.xmpp_return

Return salt data via xmpp

depends sleekxmpp >= 1.3.1

The following fields can be set in the minion conf file:

```
xmpp.jid (required)
xmpp.password (required)
xmpp.recipient (required)
xmpp.profile (optional)
```

Alternative configuration values can be used by prefacing the configuration. Any values not found in the alternative configuration will be pulled from the default location:

```
xmpp.jid
xmpp.password
xmpp.recipient
xmpp.profile
```

XMPP settings may also be configured as:

```
xmpp:
    jid: user@xmpp.domain.com/resource
    password: password
    recipient: user@xmpp.example.com
```

```
alternative.xmpp:
    jid: user@xmpp.domain.com/resource
    password: password
    recipient: someone@xmpp.example.com

xmpp_profile:
    xmpp.jid: user@xmpp.domain.com/resource
    xmpp.password: password

xmpp:
    profile: xmpp_profile
    recipient: user@xmpp.example.com

alternative.xmpp:
    profile: xmpp_profile
    recipient: someone-else@xmpp.example.com
```

To use the XMPP returner, append `--return xmpp' to the salt command.

```
salt '*' test.ping --return xmpp
```

To use the alternative configuration, append `--return_config alternative' to the salt command.

New in version 2015.5.0.

```
salt '*' test.ping --return xmpp --return_config alternative
```

To override individual configuration items, append --return_kwargs `{``key:'': ``value''}' to the salt command.

New in version 2016.3.0.

```
salt '*' test.ping --return xmpp --return_kwargs '{"recipient": "someone-else@xmpp.
↪example.com"}'
```

`salt.returners.xmpp_return.`**`returner`**`(`*ret*`)`
Send an xmpp message with the data

salt.returners.zabbix_return module

Return salt data to Zabbix

The following Type: ``Zabbix trapper'' with ``Type of information'' Text items are required:

```
Key: salt.trap.info
Key: salt.trap.average
Key: salt.trap.warning
Key: salt.trap.high
Key: salt.trap.disaster
```

To use the Zabbix returner, append `--return zabbix' to the salt command. ex:

```
salt '*' test.ping --return zabbix
```

`salt.returners.zabbix_return.`**`returner`**`(`*ret*`)`

`salt.returners.zabbix_return.`**`zabbix_send`**`(`*key, host, output*`)`

`salt.returners.zabbix_return.`**`zbx`**`()`

3.25 Renderers

The Salt state system operates by gathering information from common data types such as lists, dictionaries, and strings that would be familiar to any developer.

SLS files are translated from whatever data templating format they are written in back into Python data types to be consumed by Salt.

By default SLS files are rendered as Jinja templates and then parsed as YAML documents. But since the only thing the state system cares about is raw data, the SLS files can be any structured format that can be dreamed up.

Currently there is support for `Jinja + YAML`, `Mako + YAML`, `Wempy + YAML`, `Jinja + json`, `Mako + json` and `Wempy + json`.

Renderers can be written to support any template type. This means that the Salt states could be managed by XML files, HTML files, Puppet files, or any format that can be translated into the Pythonic data structure used by the state system.

3.25.1 Multiple Renderers

A default renderer is selected in the master configuration file by providing a value to the `renderer` key.

When evaluating an SLS, more than one renderer can be used.

When rendering SLS files, Salt checks for the presence of a Salt-specific shebang line.

The shebang line directly calls the name of the renderer as it is specified within Salt. One of the most common reasons to use multiple renderers is to use the Python or `py` renderer.

Below, the first line is a shebang that references the `py` renderer.

```
#!py

def run():
    '''
    Install the python-mako package
    '''
    return {'include': ['python'],
            'python-mako': {'pkg': ['installed']}}
```

3.25.2 Composing Renderers

A renderer can be composed from other renderers by connecting them in a series of pipes(|).

In fact, the default `Jinja + YAML` renderer is implemented by connecting a YAML renderer to a Jinja renderer. Such renderer configuration is specified as: `jinja | yaml`.

Other renderer combinations are possible:

`yaml` i.e, just YAML, no templating.

`mako | yaml` pass the input to the `mako` renderer, whose output is then fed into the `yaml` renderer.

`jinja | mako | yaml` This one allows you to use both jinja and mako templating syntax in the input and then parse the final rendered output as YAML.

The following is a contrived example SLS file using the `jinja | mako | yaml` renderer:

```
#!jinja|mako|yaml

An_Example:
  cmd.run:
    - name: |
        echo "Using Salt ${grains['saltversion']}" \
             "from path {{grains['saltpath']}}."
    - cwd: /

<%doc> ${...} is Mako's notation, and so is this comment. </%doc>
{#     Similarly, {{...}} is Jinja's notation, and so is this comment. #}
```

For backward compatibility, `jinja | yaml` can also be written as `yaml_jinja`, and similarly, the `yaml_mako`, `yaml_wempy`, `json_jinja`, `json_mako`, and `json_wempy` renderers are all supported.

Keep in mind that not all renderers can be used alone or with any other renderers. For example, the template renderers shouldn't be used alone as their outputs are just strings, which still need to be parsed by another renderer to turn them into highstate data structures.

For example, it doesn't make sense to specify `yaml | jinja` because the output of the YAML renderer is a highstate data structure (a dict in Python), which cannot be used as the input to a template renderer. Therefore, when combining renderers, you should know what each renderer accepts as input and what it returns as output.

3.25.3 Writing Renderers

A custom renderer must be a Python module placed in the renderers directory and the module implement the `render` function.

The `render` function will be passed the path of the SLS file as an argument.

The purpose of the `render` function is to parse the passed file and to return the Python data structure derived from the file.

Custom renderers must be placed in a `_renderers` directory within the *`file_roots`* specified by the master config file.

Custom renderers are distributed when any of the following are run:

- `state.apply`
- *`saltutil.sync_renderers`*
- *`saltutil.sync_all`*

Any custom renderers which have been synced to a minion, that are named the same as one of Salt's default set of renderers, will take the place of the default renderer with the same name.

3.25.4 Examples

The best place to find examples of renderers is in the Salt source code.

Documentation for renderers included with Salt can be found here:

https://github.com/saltstack/salt/blob/develop/salt/renderers

Here is a simple YAML renderer example:

```
import salt.utils.yaml
from salt.utils.yamlloader import SaltYamlSafeLoader
from salt.ext import six

def render(yaml_data, saltenv='', sls='', **kws):
    if not isinstance(yaml_data, six.string_types):
        yaml_data = yaml_data.read()
    data = salt.utils.yaml.safe_load(yaml_data)
    return data if data else {}
```

3.25.5 Full List of Renderers

renderer modules

cheetah	Cheetah Renderer for Salt
dson	DSON Renderer for Salt
genshi	Genshi Renderer for Salt
gpg	Renderer that will decrypt GPG ciphers
hjson	Hjson Renderer for Salt
jinja	Jinja loading utils to enable a more powerful backend for jinja templates
json	JSON Renderer for Salt
json5	JSON5 Renderer for Salt
mako	Mako Renderer for Salt
msgpack	
pass	Pass Renderer for Salt
py	Pure python state renderer
pydsl	A Python-based DSL
pyobjects	Python renderer that includes a Pythonic Object based interface
stateconf	A flexible renderer that takes a templating engine and a data format
wempy	
yaml	YAML Renderer for Salt
yamlex	

salt.renderers.cheetah

Cheetah Renderer for Salt

salt.renderers.cheetah.**render**(*cheetah_data, saltenv=u'base', sls=u'`, method=u'xml', **kws*)

Render a Cheetah template.

Return type A Python data structure

salt.renderers.dson

DSON Renderer for Salt

This renderer is intended for demonstration purposes. Information on the DSON spec can be found here.

This renderer requires Dogeon (installable via pip)

salt.renderers.dson.**render**(*dson_input*, *saltenv=u'base'*, *sls=u''*, ***kwargs*)
Accepts DSON data as a string or as a file object and runs it through the JSON parser.

Return type A Python data structure

salt.renderers.genshi

Genshi Renderer for Salt

salt.renderers.genshi.**render**(*genshi_data*, *saltenv=u'base'*, *sls=u''*, *method=u'xml'*, ***kws*)
Render a Genshi template. A method should be passed in as part of the kwargs. If no method is passed in, xml is assumed. Valid methods are:

Note that the `text` method will call `NewTextTemplate`. If `oldtext` is desired, it must be called explicitly

Return type A Python data structure

salt.renderers.gpg

Renderer that will decrypt GPG ciphers

Any key in the SLS file can be a GPG cipher, and this renderer will decrypt it before passing it off to Salt. This allows you to safely store secrets in source control, in such a way that only your Salt master can decrypt them and distribute them only to the minions that need them.

The typical use-case would be to use ciphers in your pillar data, and keep a secret key on your master. You can put the public key in source control so that developers can add new secrets quickly and easily.

This renderer requires the gpg binary. No python libraries are required as of the 2015.8.0 release.

Setup

To set things up, first generate a keypair. On the master, run the following:

```
# mkdir -p /etc/salt/gpgkeys
# chmod 0700 /etc/salt/gpgkeys
# gpg --gen-key --homedir /etc/salt/gpgkeys
```

Do not supply a password for the keypair, and use a name that makes sense for your application. Be sure to back up the `gpgkeys` directory someplace safe!

Note: Unfortunately, there are some scenarios - for example, on virtual machines which don't have real hardware - where insufficient entropy causes key generation to be extremely slow. In these cases, there are usually means of increasing the system entropy. On virtualised Linux systems, this can often be achieved by installing the `rng-tools` package.

Export the Public Key

```
# gpg --homedir /etc/salt/gpgkeys --armor --export <KEY-NAME> > exported_pubkey.gpg
```

Import the Public Key

To encrypt secrets, copy the public key to your local machine and run:

```
$ gpg --import exported_pubkey.gpg
```

To generate a cipher from a secret:

```
$ echo -n "supersecret" | gpg --armor --batch --trust-model always --encrypt -r <KEY-
↪name>
```

To apply the renderer on a file-by-file basis add the following line to the top of any pillar with gpg data in it:

```
#!yaml|gpg
```

Now with your renderer configured, you can include your ciphers in your pillar data like so:

```
#!yaml|gpg

a-secret: |
  -----BEGIN PGP MESSAGE-----
  Version: GnuPG v1

  hQEMAweRHKaPCfNeAQf9GLTN16hCfXAbPwU6BbBK0unOc7i9/etGuVc5CyU9Q6um
  QuetdvQVLFO/HkrC4lgeNQdM6D9E8PKonMlgJPyUvC8ggxhj0/IPFEKmrsnv2k6+
  cnEfmVexS7o/U1VOVjoyUeliMCJlAz/30RXaME49Cpi6No2+vKD8a4q4nZN1UZcG
  RhkhC0S22zNxOXQ38TBkmtJcqxnqT6YWKTUsjVubW3bVC+u2HGqJHu79wmwuN8tz
  m4wBkfCAd8Eyo2jEnWQcM4TcXiF01XPL4z4g1/9AAxh+Q4d8RIRP4fbw7ct4nCJv
  Gr9v2DTF7HNigIMl4ivMIn9fp+EZurJNiQskLgNbktJGAeEKYkqX5iCuB1b693hJ
  FKlwHiJt5yA8X2dDtfk8/Ph1Jx2TwGS+lGjlZaNqp3R1xuAZzXzZMLyZDe5+i3RJ
  skqmFTbOiA===Eqsm
  -----END PGP MESSAGE-----
```

Encrypted CLI Pillar Data

New in version 2016.3.0.

Functions like *state.highstate* and *state.sls* allow for pillar data to be passed on the CLI.

```
salt myminion state.highstate pillar="{'mypillar': 'foo'}"
```

Starting with the 2016.3.0 release of Salt, it is now possible for this pillar data to be GPG-encrypted, and to use the GPG renderer to decrypt it.

Replacing Newlines

To pass encrypted pillar data on the CLI, the ciphertext must have its newlines replaced with a literal backslash-n (`\n`), as newlines are not supported within Salt CLI arguments. There are a number of ways to do this:

With awk or Perl:

```
# awk
ciphertext=`echo -n "supersecret" | gpg --armor --batch --trust-model always --encrypt
↪-r user@domain.com | awk '{printf "%s\\n",$0} END {print ""}'`
```

```
# Perl
ciphertext=`echo -n "supersecret" | gpg --armor --batch --trust-model always --encrypt␣
↪-r user@domain.com | perl -pe 's/\n/\\n/g'`
```

With Python:

```
import subprocess

secret, stderr = subprocess.Popen(
    ['gpg', '--armor', '--batch', '--trust-model', 'always', '--encrypt',
     '-r', 'user@domain.com'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE).communicate(input='supersecret')

if secret:
    print(secret.replace('\n', r'\n'))
else:
    raise ValueError('No ciphertext found: {0}'.format(stderr))
```

```
ciphertext=`python /path/to/script.py`
```

The ciphertext can be included in the CLI pillar data like so:

```
salt myminion state.sls secretstuff pillar_enc=gpg pillar="{secret_pillar: '$ciphertext
↪'}"
```

The `pillar_enc=gpg` argument tells Salt that there is GPG-encrypted pillar data, so that the CLI pillar data is passed through the GPG renderer, which will iterate recursively though the CLI pillar dictionary to decrypt any encrypted values.

Encrypting the Entire CLI Pillar Dictionary

If several values need to be encrypted, it may be more convenient to encrypt the entire CLI pillar dictionary. Again, this can be done in several ways:

With awk or Perl:

```
# awk
ciphertext=`echo -n "{'secret_a': 'CorrectHorseBatteryStaple', 'secret_b': 'GPG is fun!
↪'}" | gpg --armor --batch --trust-model always --encrypt -r user@domain.com | awk '
↪{printf "%s\\n",$0} END {print ""}'`
# Perl
ciphertext=`echo -n "{'secret_a': 'CorrectHorseBatteryStaple', 'secret_b': 'GPG is fun!
↪'}" | gpg --armor --batch --trust-model always --encrypt -r user@domain.com | perl -
↪pe 's/\n/\\n/g'`
```

With Python:

```
import subprocess

pillar_data = {'secret_a': 'CorrectHorseBatteryStaple',
               'secret_b': 'GPG is fun!'}

secret, stderr = subprocess.Popen(
    ['gpg', '--armor', '--batch', '--trust-model', 'always', '--encrypt',
```

```
     '-r', 'user@domain.com'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE).communicate(input=repr(pillar_data))

if secret:
    print(secret.replace('\n', r'\n'))
else:
    raise ValueError('No ciphertext found: {0}'.format(stderr))
```

```
ciphertext=`python /path/to/script.py`
```

With the entire pillar dictionary now encrypted, it can be included in the CLI pillar data like so:

```
salt myminion state.sls secretstuff pillar_enc=gpg pillar="$ciphertext"
```

`salt.renderers.gpg.`**`render`**(*gpg_data, saltenv=u'base', sls=u'', argline=u'', **kwargs*)
Create a gpg object given a gpg_keydir, and then use it to try to decrypt the data to be rendered.

salt.renderers.hjson

Hjson Renderer for Salt http://laktak.github.io/hjson/

`salt.renderers.hjson.`**`render`**(*hjson_data, saltenv=u'base', sls=u'', **kws*)
Accepts HJSON as a string or as a file object and runs it through the HJSON parser.

Return type A Python data structure

salt.renderers.jinja

Jinja loading utils to enable a more powerful backend for jinja templates

For Jinja usage information see *Understanding Jinja*.

`salt.renderers.jinja.`**`render`**(*template_file, saltenv=u'base', sls=u'', argline=u'', context=None, tmplpath=None, **kws*)
Render the template_file, passing the functions and grains into the Jinja rendering system.

Return type *string*

class `salt.utils.jinja.`**`SerializerExtension`**(*environment*)
Yaml and Json manipulation.

Format filters

Allows jsonifying or yamlifying any data structure. For example, this dataset:

```
data = {
    'foo': True,
    'bar': 42,
    'baz': [1, 2, 3],
    'qux': 2.0
}
```

```
yaml = {{ data|yaml }}
json = {{ data|json }}
```

```
python = {{ data|python }}
xml  = {{ {'root_node': data}|xml }}
```

will be rendered as:

```
yaml = {bar: 42, baz: [1, 2, 3], foo: true, qux: 2.0}
json = {"baz": [1, 2, 3], "foo": true, "bar": 42, "qux": 2.0}
python = {'bar': 42, 'baz': [1, 2, 3], 'foo': True, 'qux': 2.0}
xml = """<<?xml version="1.0" ?>
         <root_node bar="42" foo="True" qux="2.0">
          <baz>1</baz>
          <baz>2</baz>
          <baz>3</baz>
         </root_node>"""
```

The yaml filter takes an optional flow_style parameter to control the default-flow-style parameter of the YAML dumper.

```
{{ data|yaml(False) }}
```

will be rendered as:

```
bar: 42
baz:
  - 1
  - 2
  - 3
foo: true
qux: 2.0
```

Load filters

Strings and variables can be deserialized with **load_yaml** and **load_json** tags and filters. It allows one to manipulate data directly in templates, easily:

```
{%- set yaml_src = "{foo: it works}"|load_yaml %}
{%- set json_src = "{'bar': 'for real'}"|load_json %}
Dude, {{ yaml_src.foo }} {{ json_src.bar }}!
```

will be rendered as:

```
Dude, it works for real!
```

Load tags

Salt implements `load_yaml` and `load_json` tags. They work like the import tag, except that the document is also deserialized.

Syntaxes are `{% load_yaml as [VARIABLE] %}[YOUR DATA]{% endload %}` and `{% load_json as [VARIABLE] %}[YOUR DATA]{% endload %}`

For example:

```
{% load_yaml as yaml_src %}
    foo: it works
{% endload %}
{% load_json as json_src %}
    {
        "bar": "for real"
```

```
    }
{% endload %}
Dude, {{ yaml_src.foo }} {{ json_src.bar }}!
```

will be rendered as:

```
Dude, it works for real!
```

Import tags

External files can be imported and made available as a Jinja variable.

```
{% import_yaml "myfile.yml" as myfile %}
{% import_json "defaults.json" as defaults %}
{% import_text "completeworksofshakespeare.txt" as poems %}
```

Catalog

`import_*` and `load_*` tags will automatically expose their target variable to import. This feature makes catalog of data to handle.

for example:

```
# doc1.sls
{% load_yaml as var1 %}
    foo: it works
{% endload %}
{% load_yaml as var2 %}
    bar: for real
{% endload %}
```

```
# doc2.sls
{% from "doc1.sls" import var1, var2 as local2 %}
{{ var1.foo }} {{ local2.bar }}
```

** Escape Filters **

New in version 2017.7.0.

Allows escaping of strings so they can be interpreted literally by another function.

For example:

```
regex_escape = {{ 'https://example.com?foo=bar%20baz' | regex_escape }}
```

will be rendered as:

```
regex_escape = https\:\/\/example\.com\?foo\=bar\%20baz
```

** Set Theory Filters **

New in version 2017.7.0.

Performs set math using Jinja filters.

For example:

```
unique = {{ ['foo', 'foo', 'bar'] | unique }}
```

will be rendered as:

```
unique = ['foo', 'bar']
```

salt.renderers.json

JSON Renderer for Salt

salt.renderers.json.**render**(*json_data*, *saltenv=u'base'*, *sls=u''*, ***kws*)
Accepts JSON as a string or as a file object and runs it through the JSON parser.

Return type A Python data structure

salt.renderers.json5

JSON5 Renderer for Salt

New in version 2016.3.0.

JSON5 is an unofficial extension to JSON. See http://json5.org/ for more information.

This renderer requires the json5 python bindings, installable via pip.

salt.renderers.json5.**render**(*json_data*, *saltenv=u'base'*, *sls=u''*, ***kws*)
Accepts JSON as a string or as a file object and runs it through the JSON parser.

Return type A Python data structure

salt.renderers.mako

Mako Renderer for Salt

salt.renderers.mako.**render**(*template_file*, *saltenv=u'base'*, *sls=u''*, *context=None*, *tmplpath=None*, ***kws*)
Render the template_file, passing the functions and grains into the Mako rendering system.

Return type *string*

salt.renderers.msgpack

salt.renderers.msgpack.**render**(*msgpack_data*, *saltenv=u'base'*, *sls=u''*, ***kws*)
Accepts a message pack string or a file object, renders said data back to a python dict.

Return type A Python data structure

salt.renderers.pass module

Pass Renderer for Salt

[pass](https://www.passwordstore.org/)

New in version 2017.7.0.

Setup

Note: `<user>` needs to be replaced with the user salt-master will be running as

1. Have private gpg loaded into *user*`s gpg keyring. Example:

```
load_private_gpg_key:
  cmd.run:
    - name: gpg --import <location_of_private_gpg_key>
    - unless: gpg --list-keys '<gpg_name>'
```

2. Said private key's public key should have been used when encrypting pass entries that are of interest for pillar data.

3. Fetch and keep local pass git repo up-to-date

```
update_pass:
  git.latest:
    - force_reset: True
    - name: <git_repo>
    - target: /<user>/.password-store
    - identity: <location_of_ssh_private_key>
    - require:
      - cmd: load_private_gpg_key
```

4. Install pass binary

```
pass:
  pkg.installed
```

`salt.renderers.pass.`**`render`**(*pass_info*, *saltenv=u'base'*, *sls=u'`*, *argline=u'`*, ***kwargs*)
    Fetch secret from pass based on pass_path

salt.renderers.py

Pure python state renderer

To use this renderer, the SLS file should contain a function called `run` which returns highstate data.

The highstate data is a dictionary containing identifiers as keys, and execution dictionaries as values. For example the following state declaration in YAML:

```
common_packages:
  pkg.installed:
   - pkgs:
      - curl
      - vim
```

tranlastes to:

```
{'common_packages': {'pkg.installed': [{'pkgs': ['curl', 'vim']}]}}
```

In this module, a few objects are defined for you, giving access to Salt's execution functions, grains, pillar, etc. They are:

- `__salt__` - *Execution functions* (i.e. `__salt__['test.echo']('foo')`)
- `__grains__` - *Grains* (i.e. `__grains__['os']`)
- `__pillar__` - *Pillar data* (i.e. `__pillar__['foo']`)
- `__opts__` - Minion configuration options
- `__env__` - The effective salt fileserver environment (i.e. `base`). Also referred to as a ``saltenv''. `__env__` should not be modified in a pure python SLS file. To use a different environment, the environment should be set when executing the state. This can be done in a couple different ways:
  - Using the `saltenv` argument on the salt CLI (i.e. `salt '*' state.sls foo.bar.baz saltenv=env_name`).
  - By adding a `saltenv` argument to an individual state within the SLS file. In other words, adding a line like this to the state's data structure: `{'saltenv': 'env_name'}`
- `__sls__` - The SLS path of the file. For example, if the root of the base environment is `/srv/salt`, and the SLS file is `/srv/salt/foo/bar/baz.sls`, then `__sls__` in that file will be `foo.bar.baz`.

The global context `data` (same as context `{{ data }}` for states written with Jinja + YAML). The following YAML + Jinja state declaration:

```
{% if data['id'] == 'mysql1' %}
highstate_run:
  local.state.apply:
    - tgt: mysql1
{% endif %}
```

translates to:

```
if data['id'] == 'mysql1':
    return {'highstate_run': {'local.state.apply': [{'tgt': 'mysql1'}]}}
```

Full Example

```
#!py

def run():
    config = {}

    if __grains__['os'] == 'Ubuntu':
        user = 'ubuntu'
        group = 'ubuntu'
        home = '/home/{0}'.format(user)
    else:
        user = 'root'
        group = 'root'
        home = '/root/'

    config['s3cmd'] = {
        'pkg': [
            'installed',
            {'name': 's3cmd'},
        ],
    }

    config[home + '/.s3cfg'] = {
```

```
        'file.managed': [
            {'source': 'salt://s3cfg/templates/s3cfg'},
            {'template': 'jinja'},
            {'user': user},
            {'group': group},
            {'mode': 600},
            {'context': {
                'aws_key': __pillar__['AWS_ACCESS_KEY_ID'],
                'aws_secret_key': __pillar__['AWS_SECRET_ACCESS_KEY'],
                },
            },
        ],
    }

    return config
```

`salt.renderers.py.render`(*template*, *saltenv=u'base'*, *sls=u''*, *tmplpath=None*, ***kws*)
Render the python module's components

Return type *string*

salt.renderers.pydsl

A Python-based DSL

maintainer Jack Kuan <kjkuan@gmail.com>

maturity new

platform all

The *pydsl* renderer allows one to author salt formulas (.sls files) in pure Python using a DSL that's easy to write and easy to read. Here's an example:

```
#!pydsl

apache = state('apache')
apache.pkg.installed()
apache.service.running()
state('/var/www/index.html') \
    .file('managed',
          source='salt://webserver/index.html') \
    .require(pkg='apache')
```

Notice that any Python code is allow in the file as it's really a Python module, so you have the full power of Python at your disposal. In this module, a few objects are defined for you, including the usual (with `__` added) `__salt__` dictionary, `__grains__`, `__pillar__`, `__opts__`, `__env__`, and `__sls__`, plus a few more:

`__file__`

local file system path to the sls module.

`__pydsl__`

Salt PyDSL object, useful for configuring DSL behavior per sls rendering.

`include`

Salt PyDSL function for creating *Include declaration*`s.

`extend`

Salt PyDSL function for creating *Extend declaration*`s.

state

Salt PyDSL function for creating *ID declaration*`s.

A state *ID declaration* is created with a `state(id)` function call. Subsequent `state(id)` call with the same id returns the same object. This singleton access pattern applies to all declaration objects created with the DSL.

```
state('example')
assert state('example') is state('example')
assert state('example').cmd is state('example').cmd
assert state('example').cmd.running is state('example').cmd.running
```

The *id* argument is optional. If omitted, an UUID will be generated and used as the *id*.

`state(id)` returns an object under which you can create a *State declaration* object by accessing an attribute named after *any* state module available in Salt.

```
state('example').cmd
state('example').file
state('example').pkg
...
```

Then, a *Function declaration* object can be created from a *State declaration* object by one of the following two ways:

1. by calling a method named after the state function on the *State declaration* object.

```
state('example').file.managed(...)
```

2. by directly calling the attribute named for the *State declaration*, and supplying the state function name as the first argument.

```
state('example').file('managed', ...)
```

With either way of creating a *Function declaration* object, any *Function arg declaration*`s can be passed as keyword arguments to the call. Subsequent calls of a *Function declaration* will update the arg declarations.

```
state('example').file('managed', source='salt://webserver/index.html')
state('example').file.managed(source='salt://webserver/index.html')
```

As a shortcut, the special *name* argument can also be passed as the first or second positional argument depending on the first or second way of calling the *State declaration* object. In the following two examples *ls -la* is the *name* argument.

```
state('example').cmd.run('ls -la', cwd='/')
state('example').cmd('run', 'ls -la', cwd='/')
```

Finally, a *Requisite declaration* object with its *Requisite reference*`s can be created by invoking one of the requisite methods (see *State Requisites*) on either a *Function declaration* object or a *State declaration* object. The return value of a requisite call is also a *Function declaration* object, so you can chain several requisite calls together.

Arguments to a requisite call can be a list of *State declaration* objects and/or a set of keyword arguments whose names are state modules and values are IDs of *ID declaration*`s or names of *Name declaration*`s.

```
apache2 = state('apache2')
apache2.pkg.installed()
state('libapache2-mod-wsgi').pkg.installed()

# you can call requisites on function declaration
```

```
apache2.service.running() \
               .require(apache2.pkg,
                        pkg='libapache2-mod-wsgi') \
               .watch(file='/etc/apache2/httpd.conf')

# or you can call requisites on state declaration.
# this actually creates an anonymous function declaration object
# to add the requisites.
apache2.service.require(state('libapache2-mod-wsgi').pkg,
                        pkg='apache2') \
               .watch(file='/etc/apache2/httpd.conf')

# we still need to set the name of the function declaration.
apache2.service.running()
```

Include declaration objects can be created with the `include` function, while *Extend declaration* objects can be created with the `extend` function, whose arguments are just *Function declaration* objects.

```
include('edit.vim', 'http.server')
extend(state('apache2').service.watch(file='/etc/httpd/httpd.conf')
```

The `include` function, by default, causes the included sls file to be rendered as soon as the `include` function is called. It returns a list of rendered module objects; sls files not rendered with the pydsl renderer return `None`s. This behavior creates no *Include declaration*`s in the resulting high state data structure.

```
import types

# including multiple sls returns a list.
_, mod = include('a-non-pydsl-sls', 'a-pydsl-sls')

assert _ is None
assert isinstance(slsmods[1], types.ModuleType)

# including a single sls returns a single object
mod = include('a-pydsl-sls')

# myfunc is a function that calls state(...) to create more states.
mod.myfunc(1, 2, "three")
```

Notice how you can define a reusable function in your pydsl sls module and then call it via the module returned by `include`.

It's still possible to do late includes by passing the `delayed=True` keyword argument to `include`.

```
include('edit.vim', 'http.server', delayed=True)
```

Above will just create a *Include declaration* in the rendered result, and such call always returns `None`.

Special integration with the *cmd* state

Taking advantage of rendering a Python module, PyDSL allows you to declare a state that calls a pre-defined Python function when the state is executed.

```
greeting = "hello world"
def helper(something, *args, **kws):
    print greeting                # hello world
```

```
    print something, args, kws    # test123 ['a', 'b', 'c'] {'x': 1, 'y': 2}

state().cmd.call(helper, "test123", 'a', 'b', 'c', x=1, y=2)
```

The *cmd.call* state function takes care of calling our `helper` function with the arguments we specified in the states, and translates the return value of our function into a structure expected by the state system. See `salt.states.cmd.call()` for more information.

Implicit ordering of states

Salt states are explicitly ordered via *Requisite declaration*`s. However, with *pydsl* it's possible to let the renderer track the order of creation for *Function declaration* objects, and implicitly add `require` requisites for your states to enforce the ordering. This feature is enabled by setting the `ordered` option on `__pydsl__`.

Note: this feature is only available if your minions are using Python >= 2.7.

```
include('some.sls.file')

A = state('A').cmd.run(cwd='/var/tmp')
extend(A)

__pydsl__.set(ordered=True)

for i in range(10):
    i = six.text_type(i)
    state(i).cmd.run('echo '+i, cwd='/')
state('1').cmd.run('echo one')
state('2').cmd.run(name='echo two')
```

Notice that the `ordered` option needs to be set after any `extend` calls. This is to prevent *pydsl* from tracking the creation of a state function that's passed to an `extend` call.

Above example should create states from `0` to `9` that will output `0`, `one`, `two`, `3`, … `9`, in that order.

It's important to know that *pydsl* tracks the *creations* of *Function declaration* objects, and automatically adds a `require` requisite to a *Function declaration* object that requires the last *Function declaration* object created before it in the sls file.

This means later calls (perhaps to update the function's *Function arg declaration*) to a previously created function declaration will not change the order.

Render time state execution

When Salt processes a salt formula file, the file is rendered to salt's high state data representation by a renderer before the states can be executed. In the case of the *pydsl* renderer, the .sls file is executed as a python module as it is being rendered which makes it easy to execute a state at render time. In *pydsl*, executing one or more states at render time can be done by calling a configured *ID declaration* object.

```
#!pydsl

s = state() # save for later invocation

# configure it
```

```
s.cmd.run('echo at render time', cwd='/')
s.file.managed('target.txt', source='salt://source.txt')

s() # execute the two states now
```

Once an *ID declaration* is called at render time it is detached from the sls module as if it was never defined.

Note: If *implicit ordering* is enabled (i.e., via `__pydsl__.set(ordered=True)`) then the *first* invocation of a *ID declaration* object must be done before a new *Function declaration* is created.

Integration with the stateconf renderer

The `salt.renderers.stateconf` renderer offers a few interesting features that can be leveraged by the *pydsl* renderer. In particular, when using with the *pydsl* renderer, we are interested in *stateconf*`s sls namespacing feature (via dot-prefixed id declarations), as well as, the automatic *start* and *goal* states generation.

Now you can use *pydsl* with *stateconf* like this:

```
#!pydsl|stateconf -ps

include('xxx', 'yyy')

# ensure that states in xxx run BEFORE states in this file.
extend(state('.start').stateconf.require(stateconf='xxx::goal'))

# ensure that states in yyy run AFTER states in this file.
extend(state('.goal').stateconf.require_in(stateconf='yyy::start'))

__pydsl__.set(ordered=True)

...
```

`-s` enables the generation of a stateconf *start* state, and `-p` lets us pipe high state data rendered by *pydsl* to *stateconf*. This example shows that by `require`-ing or `require_in`-ing the included sls' *start* or *goal* states, it's possible to ensure that the included sls files can be made to execute before or after a state in the including sls file.

Importing custom Python modules

To use a custom Python module inside a PyDSL state, place the module somewhere that it can be loaded by the Salt loader, such as *_modules* in the */srv/salt* directory.

Then, copy it to any minions as necessary by using *saltutil.sync_modules*.

To import into a PyDSL SLS, one must bypass the Python importer and insert it manually by getting a reference from Python's *sys.modules* dictionary.

For example:

```
#!pydsl|stateconf -ps

def main():
    my_mod = sys.modules['salt.loaded.ext.module.my_mod']
```

salt.renderers.pyobjects

Python renderer that includes a Pythonic Object based interface

maintainer Evan Borgstrom <evan@borgstrom.ca>

Let's take a look at how you use pyobjects in a state file. Here's a quick example that ensures the `/tmp` directory is in the correct state.

```
#!pyobjects

File.managed("/tmp", user='root', group='root', mode='1777')
```

Nice and Pythonic!

By using the ``shebang'' syntax to switch to the pyobjects renderer we can now write our state data using an object based interface that should feel at home to python developers. You can import any module and do anything that you'd like (with caution, importing sqlalchemy, django or other large frameworks has not been tested yet). Using the pyobjects renderer is exactly the same as using the built-in Python renderer with the exception that pyobjects provides you with an object based interface for generating state data.

Creating state data

Pyobjects takes care of creating an object for each of the available states on the minion. Each state is represented by an object that is the CamelCase version of its name (i.e. `File`, `Service`, `User`, etc), and these objects expose all of their available state functions (i.e. `File.managed`, `Service.running`, etc).

The name of the state is split based upon underscores (_), then each part is capitalized and finally the parts are joined back together.

Some examples:

- `postgres_user` becomes `PostgresUser`
- `ssh_known_hosts` becomes `SshKnownHosts`

Context Managers and requisites

How about something a little more complex. Here we're going to get into the core of how to use pyobjects to write states.

```
#!pyobjects

with Pkg.installed("nginx"):
    Service.running("nginx", enable=True)

    with Service("nginx", "watch_in"):
        File.managed("/etc/nginx/conf.d/mysite.conf",
                     owner='root', group='root', mode='0444',
                     source='salt://nginx/mysite.conf')
```

The objects that are returned from each of the magic method calls are setup to be used a Python context managers (`with`) and when you use them as such all declarations made within the scope will **automatically** use the enclosing state as a requisite!

The above could have also been written use direct requisite statements as.

```
#!pyobjects

Pkg.installed("nginx")
Service.running("nginx", enable=True, require=Pkg("nginx"))
File.managed("/etc/nginx/conf.d/mysite.conf",
             owner='root', group='root', mode='0444',
             source='salt://nginx/mysite.conf',
             watch_in=Service("nginx"))
```

You can use the direct requisite statement for referencing states that are generated outside of the current file.

```
#!pyobjects

# some-other-package is defined in some other state file
Pkg.installed("nginx", require=Pkg("some-other-package"))
```

The last thing that direct requisites provide is the ability to select which of the SaltStack requisites you want to use (require, require_in, watch, watch_in, use & use_in) when using the requisite as a context manager.

```
#!pyobjects

with Service("my-service", "watch_in"):
    ...
```

The above example would cause all declarations inside the scope of the context manager to automatically have their `watch_in` set to `Service("my-service")`.

Including and Extending

To include other states use the `include()` function. It takes one name per state to include.

To extend another state use the `extend()` function on the name when creating a state.

```
#!pyobjects

include('http', 'ssh')

Service.running(extend('apache'),
                watch=[File('/etc/httpd/extra/httpd-vhosts.conf')])
```

Importing from other state files

Like any Python project that grows you will likely reach a point where you want to create reusability in your state tree and share objects between state files, Map Data (described below) is a perfect example of this.

To facilitate this Python's `import` statement has been augmented to allow for a special case when working with a Salt state tree. If you specify a Salt url (`salt://...`) as the target for importing from then the pyobjects renderer will take care of fetching the file for you, parsing it with all of the pyobjects features available and then place the requested objects in the global scope of the template being rendered.

This works for all types of import statements; `import X`, `from X import Y`, and `from X import Y as Z`.

```
#!pyobjects

import salt://myfile.sls
```

```
from salt://something/data.sls import Object
from salt://something/data.sls import Object as Other
```

See the Map Data section for a more practical use.

Caveats:

- Imported objects are ALWAYS put into the global scope of your template, regardless of where your import statement is.

Salt object

In the spirit of the object interface for creating state data pyobjects also provides a simple object interface to the `__salt__` object.

A function named `salt` exists in scope for your sls files and will dispatch its attributes to the `__salt__` dictionary.

The following lines are functionally equivalent:

```
#!pyobjects

ret = salt.cmd.run(bar)
ret = __salt__['cmd.run'](bar)
```

Pillar, grain, mine & config data

Pyobjects provides shortcut functions for calling `pillar.get`, `grains.get`, `mine.get` & `config.get` on the `__salt__` object. This helps maintain the readability of your state files.

Each type of data can be access by a function of the same name: `pillar()`, `grains()`, `mine()` and `config()`.

The following pairs of lines are functionally equivalent:

```
#!pyobjects

value = pillar('foo:bar:baz', 'qux')
value = __salt__['pillar.get']('foo:bar:baz', 'qux')

value = grains('pkg:apache')
value = __salt__['grains.get']('pkg:apache')

value = mine('os:Fedora', 'network.interfaces', 'grain')
value = __salt__['mine.get']('os:Fedora', 'network.interfaces', 'grain')

value = config('foo:bar:baz', 'qux')
value = __salt__['config.get']('foo:bar:baz', 'qux')
```

Map Data

When building complex states or formulas you often need a way of building up a map of data based on grain data. The most common use of this is tracking the package and service name differences between distributions.

To build map data using pyobjects we provide a class named Map that you use to build your own classes with inner classes for each set of values for the different grain matches.

```python
#!pyobjects

class Samba(Map):
    merge = 'samba:lookup'
    # NOTE: priority is new to 2017.7.0
    priority = ('os_family', 'os')

    class Ubuntu:
        __grain__ = 'os'
        service = 'smbd'

    class Debian:
        server = 'samba'
        client = 'samba-client'
        service = 'samba'

    class RHEL:
        __match__ = 'RedHat'
        server = 'samba'
        client = 'samba'
        service = 'smb'
```

Note: By default, the `os_family` grain will be used as the target for matching. This can be overridden by specifying a `__grain__` attribute.

If a `__match__` attribute is defined for a given class, then that value will be matched against the targeted grain, otherwise the class name's value will be be matched.

Given the above example, the following is true:

1. Minions with an `os_family` of **Debian** will be assigned the attributes defined in the **Debian** class.
2. Minions with an `os` grain of **Ubuntu** will be assigned the attributes defined in the **Ubuntu** class.
3. Minions with an `os_family` grain of **RedHat** will be assigned the attributes defined in the **RHEL** class.

That said, sometimes a minion may match more than one class. For instance, in the above example, Ubuntu minions will match both the **Debian** and **Ubuntu** classes, since Ubuntu has an `os_family` grain of **Debian** an an `os` grain of **Ubuntu**. As of the 2017.7.0 release, the order is dictated by the order of declaration, with classes defined later overriding earlier ones. Additionally, 2017.7.0 adds support for explicitly defining the ordering using an optional attribute called `priority`.

Given the above example, `os_family` matches will be processed first, with `os` matches processed after. This would have the effect of assigning `smbd` as the `service` attribute on Ubuntu minions. If the `priority` item was not defined, or if the order of the items in the `priority` tuple were reversed, Ubuntu minions would have a `service` attribute of `samba`, since `os_family` matches would have been processed second.

To use this new data you can import it into your state file and then access your attributes. To access the data in the map you simply access the attribute name on the base class that is extending Map. Assuming the above Map was in the file `samba/map.sls`, you could do the following.

```python
#!pyobjects

from salt://samba/map.sls import Samba

with Pkg.installed("samba", names=[Samba.server, Samba.client]):
    Service.running("samba", name=Samba.service)
```

class salt.renderers.pyobjects.**PyobjectsModule**(*name*, *attrs*)
This provides a wrapper for bare imports.

salt.renderers.pyobjects.**load_states**()
This loads our states into the salt __context__

salt.renderers.stateconf

maintainer Jack Kuan <kjkuan@gmail.com>

maturity new

platform all

This module provides a custom renderer that processes a salt file with a specified templating engine (e.g. Jinja) and a chosen data renderer (e.g. YAML), extracts arguments for any `stateconf.set` state, and provides the extracted arguments (including Salt-specific args, such as `require`, etc) as template context. The goal is to make writing reusable/configurable/parameterized salt files easier and cleaner.

To use this renderer, either set it as the default renderer via the `renderer` option in master/minion's config, or use the shebang line in each individual sls file, like so: `#!stateconf`. Note, due to the way this renderer works, it must be specified as the first renderer in a render pipeline. That is, you cannot specify `#!mako|yaml|stateconf`, for example. Instead, you specify them as renderer arguments: `#!stateconf mako . yaml`.

Here's a list of features enabled by this renderer.

- Prefixes any state id (declaration or reference) that starts with a dot (`.`) to avoid duplicated state ids when the salt file is included by other salt files.

  For example, in the *salt://some/file.sls*, a state id such as `.sls_params` will be turned into `some.file::sls_params`. Example:

```
#!stateconf yaml . jinja

.vim:
  pkg.installed
```

  Above will be translated into:

```
some.file::vim:
  pkg.installed:
    - name: vim
```

  Notice how that if a state under a dot-prefixed state id has no `name` argument then one will be added automatically by using the state id with the leading dot stripped off.

  The leading dot trick can be used with extending state ids as well, so you can include relatively and extend relatively. For example, when extending a state in *salt://some/other_file.sls*, e.g.:

```
#!stateconf yaml . jinja

include:
  - .file

extend:
  .file::sls_params:
    stateconf.set:
      - name1: something
```

Above will be pre-processed into:

```
include:
  - some.file

extend:
  some.file::sls_params:
    stateconf.set:
      - name1: something
```

- Adds a `sls_dir` context variable that expands to the directory containing the rendering salt file. So, you can write `salt://{{sls_dir}}/...` to reference templates files used by your salt file.
- Recognizes the special state function, `stateconf.set`, that configures a default list of named arguments usable within the template context of the salt file. Example:

```
#!stateconf yaml . jinja

.sls_params:
  stateconf.set:
    - name1: value1
    - name2: value2
    - name3:
      - value1
      - value2
      - value3
    - require_in:
      - cmd: output

# --- end of state config ---

.output:
  cmd.run:
    - name: |
        echo 'name1={{sls_params.name1}}
              name2={{sls_params.name2}}
              name3[1]={{sls_params.name3[1]}}
        '
```

This even works with `include` + `extend` so that you can override the default configured arguments by including the salt file and then `extend` the `stateconf.set` states that come from the included salt file. (*IMPORTANT: Both the included and the extending sls files must use the stateconf renderer for this ``extend`` to work!*)

Notice that the end of configuration marker (`# ---end of state config --`) is needed to separate the use of `stateconf.set' form the rest of your salt file. The regex that matches such marker can be configured via the `stateconf_end_marker` option in your master or minion config file.

Sometimes, it is desirable to set a default argument value that's based on earlier arguments in the same `stateconf.set`. For example, it may be tempting to do something like this:

```
#!stateconf yaml . jinja

.apache:
  stateconf.set:
    - host: localhost
    - port: 1234
    - url: 'http://{{host}}:{{port}}/'
```

```
# --- end of state config ---

.test:
  cmd.run:
    - name: echo '{{apache.url}}'
    - cwd: /
```

However, this won't work. It can however be worked around like so:

```
#!stateconf yaml . jinja

.apache:
  stateconf.set:
    - host: localhost
    - port: 1234
{#  - url: 'http://{{host}}:{{port}}/' #}

# --- end of state config ---
# {{ apache.setdefault('url', "http://%(host)s:%(port)s/" % apache) }}

.test:
  cmd.run:
    - name: echo '{{apache.url}}'
    - cwd: /
```

- Adds support for relative include and exclude of .sls files. Example:

```
#!stateconf yaml . jinja

include:
  - .apache
  - .db.mysql
  - ..app.django

exclude:
  - sls: .users
```

If the above is written in a salt file at *salt://some/where.sls* then it will include *salt://some/apache.sls*, *salt://some/db/mysql.sls* and *salt://app/django.sls*, and exclude *salt://some/users.ssl*. Actually, it does that by rewriting the above `include` and `exclude` into:

```
include:
  - some.apache
  - some.db.mysql
  - app.django

exclude:
  - sls: some.users
```

- Optionally (enabled by default, *disable* via the *-G* renderer option, e.g. in the shebang line: `#!stateconf -G`), generates a `stateconf.set` goal state (state id named as `.goal` by default, configurable via the master/minion config option, `stateconf_goal_state`) that requires all other states in the salt file. Note, the `.goal` state id is subject to dot-prefix rename rule mentioned earlier.

  Such goal state is intended to be required by some state in an including salt file. For example, in your webapp salt file, if you include a sls file that is supposed to setup Tomcat, you might want to make sure that all states in the Tomcat sls file will be executed before some state in the webapp sls file.

- Optionally (enable via the *-o* renderer option, e.g. in the shebang line: `#!stateconf -o`), orders the states in a sls file by adding a `require` requisite to each state such that every state requires the state defined just before it. The order of the states here is the order they are defined in the sls file. (Note: this feature is only available if your minions are using Python >= 2.7. For Python2.6, it should also work if you install the *ordereddict* module from PyPI)

  By enabling this feature, you are basically agreeing to author your sls files in a way that gives up the explicit (or implicit?) ordering imposed by the use of `require`, `watch`, `require_in` or `watch_in` requisites, and instead, you rely on the order of states you define in the sls files. This may or may not be a better way for you. However, if there are many states defined in a sls file, then it tends to be easier to see the order they will be executed with this feature.

  You are still allowed to use all the requisites, with a few restrictions. You cannot `require` or `watch` a state defined *after* the current state. Similarly, in a state, you cannot `require_in` or `watch_in` a state defined *before* it. Breaking any of the two restrictions above will result in a state loop. The renderer will check for such incorrect uses if this feature is enabled.

  Additionally, `names` declarations cannot be used with this feature because the way they are compiled into low states make it impossible to guarantee the order in which they will be executed. This is also checked by the renderer. As a workaround for not being able to use `names`, you can achieve the same effect, by generate your states with the template engine available within your sls file.

  Finally, with the use of this feature, it becomes possible to easily make an included sls file execute all its states *after* some state (say, with id `X`) in the including sls file. All you have to do is to make state, `X`, `require_in` the first state defined in the included sls file.

When writing sls files with this renderer, one should avoid using what can be defined in a `name` argument of a state as the state's id. That is, avoid writing states like this:

```
/path/to/some/file:
  file.managed:
    - source: salt://some/file

cp /path/to/some/file file2:
  cmd.run:
    - cwd: /
    - require:
      - file: /path/to/some/file
```

Instead, define the state id and the `name` argument separately for each state. Also, the ID should be something meaningful and easy to reference within a requisite (which is a good habit anyway, and such extra indirection would also makes the sls file easier to modify later). Thus, the above states should be written like this:

```
add-some-file:
  file.managed:
    - name: /path/to/some/file
    - source: salt://some/file

copy-files:
  cmd.run:
    - name: cp /path/to/some/file file2
    - cwd: /
    - require:
      - file: add-some-file
```

Moreover, when referencing a state from a requisite, you should reference the state's id plus the state name rather than the state name plus its `name` argument. (Yes, in the above example, you can actually `require` the `file: /path/to/some/file`, instead of the `file: add-some-file`). The reason is that this renderer will re-write

or rename state id's and their references for state id's prefixed with `.`. So, if you reference `name` then there's no way to reliably rewrite such reference.

salt.renderers.wempy

`salt.renderers.wempy.render`(*template_file*, *saltenv=u'base'*, *sls=u''*, *argline=u''*, *context=None*, ***kws*)

Render the data passing the functions and grains into the rendering system

Return type *string*

salt.renderers.yaml

Understanding YAML

The default renderer for SLS files is the YAML renderer. YAML is a markup language with many powerful features. However, Salt uses a small subset of YAML that maps over very commonly used data structures, like lists and dictionaries. It is the job of the YAML renderer to take the YAML data structure and compile it into a Python data structure for use by Salt.

Though YAML syntax may seem daunting and terse at first, there are only three very simple rules to remember when writing YAML for SLS files.

Rule One: Indentation

YAML uses a fixed indentation scheme to represent relationships between data layers. Salt requires that the indentation for each level consists of exactly two spaces. Do not use tabs.

Rule Two: Colons

Python dictionaries are, of course, simply key-value pairs. Users from other languages may recognize this data type as hashes or associative arrays.

Dictionary keys are represented in YAML as strings terminated by a trailing colon. Values are represented by either a string following the colon, separated by a space:

```
my_key: my_value
```

In Python, the above maps to:

```
{'my_key': 'my_value'}
```

Dictionaries can be nested:

```
first_level_dict_key:
  second_level_dict_key: value_in_second_level_dict
```

And in Python:

```
{'first_level_dict_key': {'second_level_dict_key': 'value_in_second_level_dict' }
```

Rule Three: Dashes

To represent lists of items, a single dash followed by a space is used. Multiple items are a part of the same list as a function of their having the same level of indentation.

```
- list_value_one
- list_value_two
- list_value_three
```

Lists can be the value of a key-value pair. This is quite common in Salt:

```
my_dictionary:
  - list_value_one
  - list_value_two
  - list_value_three
```

Reference

YAML Renderer for Salt

For YAML usage information see *Understanding YAML*.

`salt.renderers.yaml.`**`get_yaml_loader`**(*argline*)
Return the ordered dict yaml loader

`salt.renderers.yaml.`**`render`**(*yaml_data, saltenv=u'base', sls=u'`, argline=u'`, **kws*)
Accepts YAML as a string or as a file object and runs it through the YAML parser.

Return type A Python data structure

salt.renderers.yamlex

YAMLEX renderer is a replacement of the YAML renderer. It's 100% YAML with a pinch of Salt magic:

- All mappings are automatically OrderedDict
- All strings are automatically str obj
- data aggregation with !aggregation yaml tag, based on the `salt.utils.aggregation` module.
- data aggregation over documents for pillar

Instructed aggregation within the `!aggregation` and the `!reset` tags:

```
#!yamlex
foo: !aggregate first
foo: !aggregate second
bar: !aggregate {first: foo}
bar: !aggregate {second: bar}
baz: !aggregate 42
qux: !aggregate default
!reset qux: !aggregate my custom data
```

is roughly equivalent to

```
foo: [first, second]
bar: {first: foo, second: bar}
baz: [42]
qux: [my custom data]
```

Reference

salt.renderers.yamlex.**render**(*sls_data*, *saltenv=u'base'*, *sls=u''*, ***kws*)
Accepts YAML_EX as a string or as a file object and runs it through the YAML_EX parser.

Return type A Python data structure

CHAPTER 4

Using Salt

This section describes the fundamental components and concepts that you need to understand to use Salt.

4.1 Grains

Salt comes with an interface to derive information about the underlying system. This is called the grains interface, because it presents salt with grains of information. Grains are collected for the operating system, domain name, IP address, kernel, OS type, memory, and many other system properties.

The grains interface is made available to Salt modules and components so that the right salt minion commands are automatically available on the right systems.

Grain data is relatively static, though if system information changes (for example, if network settings are changed), or if a new value is assigned to a custom grain, grain data is refreshed.

Note: Grains resolve to lowercase letters. For example, `FOO`, and `foo` target the same grain.

4.1.1 Listing Grains

Available grains can be listed by using the `grains.ls' module:

```
salt '*' grains.ls
```

Grains data can be listed by using the `grains.items' module:

```
salt '*' grains.items
```

4.1.2 Grains in the Minion Config

Grains can also be statically assigned within the minion configuration file. Just add the option *grains* and pass options to it:

```
grains:
  roles:
    - webserver
    - memcache
  deployment: datacenter4
```

```
  cabinet: 13
  cab_u: 14-15
```

Then status data specific to your servers can be retrieved via Salt, or used inside of the State system for matching. It also makes targeting, in the case of the example above, simply based on specific data about your deployment.

4.1.3 Grains in /etc/salt/grains

If you do not want to place your custom static grains in the minion config file, you can also put them in `/etc/salt/grains` on the minion. They are configured in the same way as in the above example, only without a top-level `grains:` key:

```
roles:
  - webserver
  - memcache
deployment: datacenter4
cabinet: 13
cab_u: 14-15
```

Note: Grains in `/etc/salt/grains` are ignored if you specify the same grains in the minion config.

Note: Grains are static, and since they are not often changed, they will need a grains refresh when they are updated. You can do this by calling: `salt minion saltutil.refresh_modules`

Note: You can equally configure static grains for Proxy Minions. As multiple Proxy Minion processes can run on the same machine, you need to index the files using the Minion ID, under `/etc/salt/proxy.d/<minion ID>/grains`. For example, the grains for the Proxy Minion `router1` can be defined under `/etc/salt/proxy.d/router1/grains`, while the grains for the Proxy Minion `switch7` can be put in `/etc/salt/proxy.d/switch7/grains`.

4.1.4 Matching Grains in the Top File

With correctly configured grains on the Minion, the *top file* used in Pillar or during Highstate can be made very efficient. For example, consider the following configuration:

```
'roles:webserver':
  - match: grain
  - state0

'roles:memcache':
  - match: grain
  - state1
  - state2
```

For this example to work, you would need to have defined the grain `role` for the minions you wish to match.

4.1.5 Writing Grains

The grains are derived by executing all of the ``public'' functions (i.e. those which do not begin with an underscore) found in the modules located in the Salt's core grains code, followed by those in any custom grains modules. The functions in a grains module must return a Python dictionary, where the dictionary keys are the names of grains, and each key's value is that value for that grain.

Custom grains modules should be placed in a subdirectory named `_grains` located under the *file_roots* specified by the master config file. The default path would be `/srv/salt/_grains`. Custom grains modules will be distributed to the minions when *state.highstate* is run, or by executing the *saltutil.sync_grains* or *saltutil.sync_all* functions.

Grains modules are easy to write, and (as noted above) only need to return a dictionary. For example:

```
def yourfunction():
     # initialize a grains dictionary
     grains = {}
     # Some code for logic that sets grains like
     grains['yourcustomgrain'] = True
     grains['anothergrain'] = 'somevalue'
     return grains
```

The name of the function does not matter and will not factor into the grains data at all; only the keys/values returned become part of the grains.

When to Use a Custom Grain

Before adding new grains, consider what the data is and remember that grains should (for the most part) be static data.

If the data is something that is likely to change, consider using *Pillar* or an execution module instead. If it's a simple set of key/value pairs, pillar is a good match. If compiling the information requires that system commands be run, then putting this information in an execution module is likely a better idea.

Good candidates for grains are data that is useful for targeting minions in the *top file* or the Salt CLI. The name and data structure of the grain should be designed to support many platforms, operating systems or applications. Also, keep in mind that Jinja templating in Salt supports referencing pillar data as well as invoking functions from execution modules, so there's no need to place information in grains to make it available to Jinja templates. For example:

```
...
...
{{ salt['module.function_name']('argument_1', 'argument_2') }}
{{ pillar['my_pillar_key'] }}
...
...
```

Warning: Custom grains will not be available in the top file until after the first *highstate*. To make custom grains available on a minion's first highstate, it is recommended to use *this example* to ensure that the custom grains are synced when the minion starts.

Loading Custom Grains

If you have multiple functions specifying grains that are called from a `main` function, be sure to prepend grain function names with an underscore. This prevents Salt from including the loaded grains from the grain functions in the final grain data structure. For example, consider this custom grain file:

```
#!/usr/bin/env python
def _my_custom_grain():
    my_grain = {'foo': 'bar', 'hello': 'world'}
    return my_grain


def main():
    # initialize a grains dictionary
    grains = {}
    grains['my_grains'] = _my_custom_grain()
    return grains
```

The output of this example renders like so:

```
# salt-call --local grains.items
local:
    ----------
    <Snipped for brevity>
    my_grains:
        ----------
        foo:
            bar
        hello:
            world
```

However, if you don't prepend the `my_custom_grain` function with an underscore, the function will be rendered twice by Salt in the items output: once for the `my_custom_grain` call itself, and again when it is called in the `main` function:

```
# salt-call --local grains.items
local:
----------
    <Snipped for brevity>
    foo:
        bar
    <Snipped for brevity>
    hello:
        world
    <Snipped for brevity>
    my_grains:
        ----------
        foo:
            bar
        hello:
            world
```

4.1.6 Precedence

Core grains can be overridden by custom grains. As there are several ways of defining custom grains, there is an order of precedence which should be kept in mind when defining them. The order of evaluation is as follows:

1. Core grains.
2. Custom grains in `/etc/salt/grains`.
3. Custom grains in `/etc/salt/minion`.
4. Custom grain modules in `_grains` directory, synced to minions.

Each successive evaluation overrides the previous ones, so any grains defined by custom grains modules synced to minions that have the same name as a core grain will override that core grain. Similarly, grains from `/etc/salt/minion` override both core grains and custom grain modules, and grains in `_grains` will override *any* grains of the same name.

4.1.7 Examples of Grains

The core module in the grains package is where the main grains are loaded by the Salt minion and provides the principal example of how to write grains:

https://github.com/saltstack/salt/blob/develop/salt/grains/core.py

4.1.8 Syncing Grains

Syncing grains can be done a number of ways, they are automatically synced when *state.highstate* is called, or (as noted above) the grains can be manually synced and reloaded by calling the *saltutil.sync_grains* or *saltutil.sync_all* functions.

Note: When the *grains_cache* is set to False, the grains dictionary is built and stored in memory on the minion. Every time the minion restarts or `saltutil.refresh_grains` is run, the grain dictionary is rebuilt from scratch.

4.2 Storing Static Data in the Pillar

Pillar is an interface for Salt designed to offer global values that can be distributed to minions. Pillar data is managed in a similar way as the Salt State Tree.

Pillar was added to Salt in version 0.9.8

Note: Storing sensitive data

Pillar data is compiled on the master. Additionally, pillar data for a given minion is only accessible by the minion for which it is targeted in the pillar configuration. This makes pillar useful for storing sensitive data specific to a particular minion.

4.2.1 Declaring the Master Pillar

The Salt Master server maintains a *pillar_roots* setup that matches the structure of the *file_roots* used in the Salt file server. Like *file_roots*, the *pillar_roots* option maps environments to directories. The pillar data is then mapped to minions based on matchers in a top file which is laid out in the same way as the state top file. Salt pillars can use the same matcher types as the standard *top file*.

conf_master:*pillar_roots* is configured just like *file_roots*. For example:

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

This example configuration declares that the base environment will be located in the `/srv/pillar` directory. It must not be in a subdirectory of the state tree.

The top file used matches the name of the top file used for States, and has the same structure:

/srv/pillar/top.sls

```
base:
  '*':
    - packages
```

In the above top file, it is declared that in the `base` environment, the glob matching all minions will have the pillar data found in the `packages` pillar available to it. Assuming the `pillar_roots` value of `/srv/pillar` taken from above, the `packages` pillar would be located at `/srv/pillar/packages.sls`.

Any number of matchers can be added to the base environment. For example, here is an expanded version of the Pillar top file stated above:

/srv/pillar/top.sls:

```
base:
  '*':
    - packages
  'web*':
    - vim
```

In this expanded top file, minions that match `web*` will have access to the `/srv/pillar/packages.sls` file, as well as the `/srv/pillar/vim.sls` file.

Another example shows how to use other standard top matching types to deliver specific salt pillar data to minions with different properties.

Here is an example using the `grains` matcher to target pillars to minions by their `os` grain:

```
dev:
  'os:Debian':
    - match: grain
    - servers
```

/srv/pillar/packages.sls

```
{% if grains['os'] == 'RedHat' %}
apache: httpd
git: git
{% elif grains['os'] == 'Debian' %}
apache: apache2
git: git-core
{% endif %}

company: Foo Industries
```

Important: See *Is Targeting using Grain Data Secure?* for important security information.

The above pillar sets two key/value pairs. If a minion is running RedHat, then the apache key is set to httpd and the git key is set to the value of git. If the minion is running Debian, those values are changed to apache2 and git-core respectively. All minions that have this pillar targeting to them via a top file will have the key of company with a value of Foo Industries.

Consequently this data can be used from within modules, renderers, State SLS files, and more via the shared pillar dictionary:

```
apache:
  pkg.installed:
    - name: {{ pillar['apache'] }}
```

```
git:
  pkg.installed:
    - name: {{ pillar['git'] }}
```

Finally, the above states can utilize the values provided to them via Pillar. All pillar values targeted to a minion are available via the `pillar' dictionary. As seen in the above example, Jinja substitution can then be utilized to access the keys and values in the Pillar dictionary.

Note that you cannot just list key/value-information in top.sls. Instead, target a minion to a pillar file and then list the keys and values in the pillar. Here is an example top file that illustrates this point:

```
base:
  '*':
     - common_pillar
```

And the actual pillar file at `/srv/pillar/common_pillar.sls':

```
foo: bar
boo: baz
```

Note: When working with multiple pillar environments, assuming that each pillar environment has its own top file, the jinja placeholder {{ saltenv }} can be used in place of the environment name:

```
{{ saltenv }}:
  '*':
     - common_pillar
```

Yes, this is {{ saltenv }}, and not {{ pillarenv }}. The reason for this is because the Pillar top files are parsed using some of the same code which parses top files when *running states*, so the pillar environment takes the place of {{ saltenv }} in the jinja context.

4.2.2 Dynamic Pillar Environments

If environment __env__ is specified in *pillar_roots*, all environments that are not explicitly specified in *pillar_roots* will map to the directories from __env__. This allows one to use dynamic git branch based environments for state/pillar files with the same file-based pillar applying to all environments. For example:

```
pillar_roots:
  __env__:
    - /srv/pillar

ext_pillar:
```

```
  - git:
    - __env__ https://example.com/git-pillar.git
```

New in version 2017.7.5,2018.3.1.

4.2.3 Pillar Namespace Flattening

The separate pillar SLS files all merge down into a single dictionary of key-value pairs. When the same key is defined in multiple SLS files, this can result in unexpected behavior if care is not taken to how the pillar SLS files are laid out.

For example, given a `top.sls` containing the following:

```
base:
  '*':
    - packages
    - services
```

with `packages.sls` containing:

```
bind: bind9
```

and `services.sls` containing:

```
bind: named
```

Then a request for the `bind` pillar key will only return `named`. The `bind9` value will be lost, because `services.sls` was evaluated later.

Note: Pillar files are applied in the order they are listed in the top file. Therefore conflicting keys will be overwritten in a `last one wins' manner! For example, in the above scenario conflicting key values in `services` will overwrite those in `packages` because it's at the bottom of the list.

It can be better to structure your pillar files with more hierarchy. For example the `package.sls` file could be configured like so:

```
packages:
  bind: bind9
```

This would make the `packages` pillar key a nested dictionary containing a `bind` key.

4.2.4 Pillar Dictionary Merging

If the same pillar key is defined in multiple pillar SLS files, and the keys in both files refer to nested dictionaries, then the content from these dictionaries will be recursively merged.

For example, keeping the `top.sls` the same, assume the following modifications to the pillar SLS files:

`packages.sls`:

```
bind:
  package-name: bind9
  version: 9.9.5
```

`services.sls`:

```
bind:
  port: 53
  listen-on: any
```

The resulting pillar dictionary will be:

```
$ salt-call pillar.get bind
local:
    ----------
    listen-on:
        any
    package-name:
        bind9
    port:
        53
    version:
        9.9.5
```

Since both pillar SLS files contained a `bind` key which contained a nested dictionary, the pillar dictionary's `bind` key contains the combined contents of both SLS files' `bind` keys.

4.2.5 Including Other Pillars

New in version 0.16.0.

Pillar SLS files may include other pillar files, similar to State files. Two syntaxes are available for this purpose. The simple form simply includes the additional pillar as if it were part of the same file:

```
include:
  - users
```

The full include form allows two additional options -- passing default values to the templating engine for the included pillar file as well as an optional key under which to nest the results of the included pillar:

```
include:
  - users:
      defaults:
          sudo: ['bob', 'paul']
      key: users
```

With this form, the included file (users.sls) will be nested within the `users' key of the compiled pillar. Additionally, the `sudo' value will be available as a template variable to users.sls.

4.2.6 In-Memory Pillar Data vs. On-Demand Pillar Data

Since compiling pillar data is computationally expensive, the minion will maintain a copy of the pillar data in memory to avoid needing to ask the master to recompile and send it a copy of the pillar data each time pillar data is requested. This in-memory pillar data is what is returned by the *`pillar.item`*, *`pillar.get`*, and *`pillar.raw`* functions.

Also, for those writing custom execution modules, or contributing to Salt's existing execution modules, the in-memory pillar data is available as the `__pillar__` dunder dictionary.

The in-memory pillar data is generated on minion start, and can be refreshed using the *`saltutil.refresh_pillar`* function:

```
salt '*' saltutil.refresh_pillar
```

This function triggers the minion to asynchronously refresh the in-memory pillar data and will always return None.

In contrast to in-memory pillar data, certain actions trigger pillar data to be compiled to ensure that the most up-to-date pillar data is available. These actions include:

- Running states
- Running *pillar.items*

Performing these actions will *not* refresh the in-memory pillar data. So, if pillar data is modified, and then states are run, the states will see the updated pillar data, but *pillar.item*, *pillar.get*, and *pillar.raw* will not see this data unless refreshed using *saltutil.refresh_pillar*.

4.2.7 How Pillar Environments Are Handled

When multiple pillar environments are used, the default behavior is for the pillar data from all environments to be merged together. The pillar dictionary will therefore contain keys from all configured environments.

The *pillarenv* minion config option can be used to force the minion to only consider pillar configuration from a single environment. This can be useful in cases where one needs to run states with alternate pillar data, either in a testing/QA environment or to test changes to the pillar data before pushing them live.

For example, assume that the following is set in the minion config file:

```
pillarenv: base
```

This would cause that minion to ignore all other pillar environments besides base when compiling the in-memory pillar data. Then, when running states, the pillarenv CLI argument can be used to override the minion's *pillarenv* config value:

```
salt '*' state.apply mystates pillarenv=testing
```

The above command will run the states with pillar data sourced exclusively from the testing environment, without modifying the in-memory pillar data.

Note: When running states, the pillarenv CLI option does not require a *pillarenv* option to be set in the minion config file. When *pillarenv* is left unset, as mentioned above all configured environments will be combined. Running states with pillarenv=testing in this case would still restrict the states' pillar data to just that of the testing pillar environment.

Starting in the 2017.7.0 release, it is possible to pin the pillarenv to the effective saltenv, using the *pillarenv_from_saltenv* minion config option. When this is set to True, if a specific saltenv is specified when running states, the pillarenv will be the same. This essentially makes the following two commands equivalent:

```
salt '*' state.apply mystates saltenv=dev
salt '*' state.apply mystates saltenv=dev pillarenv=dev
```

However, if a pillarenv is specified, it will override this behavior. So, the following command will use the qa pillar environment but source the SLS files from the dev saltenv:

```
salt '*' state.apply mystates saltenv=dev pillarenv=qa
```

So, if a pillarenv is set in the minion config file, *pillarenv_from_saltenv* will be ignored, and passing a pillarenv on the CLI will temporarily override *pillarenv_from_saltenv*.

4.2.8 Viewing Pillar Data

To view pillar data, use the *pillar* execution module. This module includes several functions, each of them with their own use. These functions include:

- *pillar.item* - Retrieves the value of one or more keys from the *in-memory pillar datj*.
- *pillar.items* - Compiles a fresh pillar dictionary and returns it, leaving the *in-memory pillar data* untouched. If pillar keys are passed to this function however, this function acts like *pillar.item* and returns their values from the *in-memory pillar data*.
- *pillar.raw* - Like *pillar.items*, it returns the entire pillar dictionary, but from the *in-memory pillar data* instead of compiling fresh pillar data.
- *pillar.get* - Described in detail below.

4.2.9 The pillar.get Function

New in version 0.14.0.

The *pillar.get* function works much in the same way as the get method in a python dict, but with an enhancement: nested dictonaries can be traversed using a colon as a delimiter.

If a structure like this is in pillar:

```
foo:
  bar:
    baz: qux
```

Extracting it from the raw pillar in an sls formula or file template is done this way:

```
{{ pillar['foo']['bar']['baz'] }}
```

Now, with the new *pillar.get* function the data can be safely gathered and a default can be set, allowing the template to fall back if the value is not available:

```
{{ salt['pillar.get']('foo:bar:baz', 'qux') }}
```

This makes handling nested structures much easier.

Note: `pillar.get()` vs `salt['pillar.get']()`

It should be noted that within templating, the `pillar` variable is just a dictionary. This means that calling `pillar.get()` inside of a template will just use the default dictionary `.get()` function which does not include the extra `:` delimiter functionality. It must be called using the above syntax (`salt['pillar.get']('foo:bar:baz','qux')`) to get the salt function, instead of the default dictionary behavior.

4.2.10 Setting Pillar Data at the Command Line

Pillar data can be set at the command line like the following example:

```
salt '*' state.apply pillar='{"cheese": "spam"}'
```

This will add a pillar key of `cheese` with its value set to `spam`.

Note: Be aware that when sending sensitive data via pillar on the command-line that the publication containing that data will be received by all minions and will not be restricted to the targeted minions. This may represent a security concern in some cases.

4.2.11 Pillar Encryption

Salt's renderer system can be used to decrypt pillar data. This allows for pillar items to be stored in an encrypted state, and decrypted during pillar compilation.

Encrypted Pillar SLS

New in version 2017.7.0.

Consider the following pillar SLS file:

```
secrets:
  vault:
    foo: |
      -----BEGIN PGP MESSAGE-----

      hQEMAw2B674HRhwSAQgAhTrN8NizwUv/VunVrqa4/X8t6EUulrnhKcSeb8sZS4th
      W1Qz3K2NjL4lkUHCQHKZVx/VoZY7zsddBIFvvoGGfj8+2wjkEDwFmFjGE4DEsS74
      ZLRFIFJC1iB/O0AiQ+oU745skQkU6OEKxqavmKMrKo3rvJ8ZCXDC470+i2/Hqrp7
      +KWGmaDOO422JaSKRm5D9bQZr9oX7KqnrPG9I1+UbJyQSJdsdtquPWmeIpamEVHb
      VMDNQRjSezZ1yKC4kCWm3YQbBF76qTHzG1VlLF5qOzuGI9VkyvlMaLfMibriqY73
      zBbPzf6Bkp2+Y9qyzuveYMmwS4sEOuZL/PetqisWe9JGAWD/O+slQ2KRu9hNww06
      KMDPJRdyj5bRuBVE4hHkkP23KrYr7SuhW2vpe7O/MvWEJ9uDNegpMLhTWruGngJh
      iFndxegN9w==
      =bAuo
      -----END PGP MESSAGE-----
    bar: this was unencrypted already
    baz: |
      -----BEGIN PGP MESSAGE-----

      hQEMAw2B674HRhwSAQf+Ne+IfsP2IcPDrUWct8sTJrga47jQvlPCmO+7zJjOVcqz
      gLjUKvMajrbI/jorBWxyAbF+5E7WdG9WHHVnuoywsyTB9rbmzuPqYCJCe+ZVyqWf
      9qgJ+oUjcvYIFmH3h7H68ldqbxaAUkAOQbTRHdr253wwaTIC91ZeX0SCj64HfTg7
      Izwk383CRWonEktXJpientApQFSUWNeLUWagEr/YPNFA3vzpPF5/Ia9X8/z/6oO2
      q+D5W5mVsns3i2HHbg2A8Y+pm4TWnH6mTSh/gdxPqssi9qIrzGQ6H1tEoFFOEq1V
      kJBe0izlfudqMq62XswzuRB4CYT5Iqw1c97T+1RqENJCASG0Wz8AGhinTdlU5iQl
      JkLKqBxcBz4L70LYWyHhYwYROJWjHgKAywX5T67ftq0wi8APuZl9olnOkwSK+wrY
      1OZi
      =7epf
      -----END PGP MESSAGE-----
    qux:
      - foo
      - bar
      - |
        -----BEGIN PGP MESSAGE-----

        hQEMAw2B674HRhwSAQgAg1YCmokrweoOI1c9HO0BLamWBaFPTMblOaTo0WJLZoTS
        ksbQ3OJAMkrkn3BnnM/djJc5C7vNs86ZfSJ+pvE8Sp1Rhtuxh25EKMqGOn/SBedI
```

```
        gR6N5vGUNiIpG5Tf3DuYAMNFDUqw8uY0MyDJI+ZW3o3xrMUABzTH0ew+Piz85FDA
        YrVgwZfqyL+9OQuu6T66jOIdwQNRX2NPFZqvon8liZUPus5VzD8E5cAL9OPxQ3sF
        f7/zE91YIXUTimrv3L7eCgU1dSxKhhfvA2bEUi+AskMWFXFuETYVrIhFJAKnkFmE
        uZx+O9R9hADW3hM5hWHKH9/CRtb0/cC84I9oCWIQPdI+AaPtICxtsD2N8Q98hhhd
        4M7I0sLZhV+4ZJqzpUsOnSpaGyfh1Zy/1d3ijJi99/l+uVHuvmMllsNmgR+ZTj0=
        =LrCQ
        -----END PGP MESSAGE-----
```

When the pillar data is compiled, the results will be decrypted:

```
# salt myminion pillar.items
myminion:
    ----------
    secrets:
        ----------
        vault:
            ----------
            bar:
                this was unencrypted already
            baz:
                rosebud
            foo:
                supersecret
            qux:
                - foo
                - bar
                - baz
```

Salt must be told what portions of the pillar data to decrypt. This is done using the *decrypt_pillar* config option:

```
decrypt_pillar:
  - 'secrets:vault': gpg
```

The notation used to specify the pillar item(s) to be decrypted is the same as the one used in *pillar.get* function.

If a different delimiter is needed, it can be specified using the *decrypt_pillar_delimiter* config option:

```
decrypt_pillar:
  - 'secrets|vault': gpg

decrypt_pillar_delimiter: '|'
```

The name of the renderer used to decrypt a given pillar item can be omitted, and if so it will fall back to the value specified by the *decrypt_pillar_default* config option, which defaults to `gpg`. So, the first example above could be rewritten as:

```
decrypt_pillar:
  - 'secrets:vault'
```

Encrypted Pillar Data on the CLI

New in version 2016.3.0.

The following functions support passing pillar data on the CLI via the `pillar` argument:

- *pillar.items*

- `state.apply`
- *state.highstate*
- *state.sls*

Triggerring decryption of this CLI pillar data can be done in one of two ways:

1. Using the `pillar_enc` argument:

```
# salt myminion pillar.items pillar_enc=gpg pillar='{foo: "-----BEGIN PGP MESSAGE-
↪----\n\nhQEMAw2B674HRhwSAQf+OvPqEdDoA2fk15I5dYUTDoj1yf/
↪pVolAma6iU4v8Zixn\nRDgWsaAnFz99FEiFACsAGDEFdZaVOxG80T0Lj+PnW4pVy0OXmXHnY2KjV9zx8FLS\nQxfvmhRR
↪OvZHhxH7cnIiGQIHc7N9nQH7ibyoKQzQMSZeilSMGr2abAHun\nmLzscr4wKMb+81Z0/
↪fdBfP6g3bLWMJga3hSzSldU9ovu7KR8rDJI1qOlENj3Wm8C\nwTpDOB33kWIKMqiAjY3JFtb5MCHrafyggwQL7cX1+tI+
↪FbjZ9CTWrQ=\n=0hO/\n-----END PGP MESSAGE-----"}'
```

   The newlines in this example are specified using a literal `\n`. Newlines can be replaced with a literal `\n` using `sed`:

```
$ echo -n bar | gpg --armor --trust-model always --encrypt -r user@domain.tld |
↪sed ':a;N;$!ba;s/\n/\\n/g'
```

   Note: Using `pillar_enc` will perform the decryption minion-side, so for this to work it will be necessary to set up the keyring in `/etc/salt/gpgkeys` on the minion just as one would typically do on the master. The easiest way to do this is to first export the keys from the master:

```
# gpg --homedir /etc/salt/gpgkeys --export-secret-key -a user@domain.tld >/tmp/
↪keypair.gpg
```

   Then, copy the file to the minion, setup the keyring, and import:

```
# mkdir -p /etc/salt/gpgkeys
# chmod 0700 /etc/salt/gpgkeys
# gpg --homedir /etc/salt/gpgkeys --list-keys
# gpg --homedir /etc/salt/gpgkeys --import --allow-secret-key-import keypair.gpg
```

   The `--list-keys` command is run create a keyring in the newly-created directory.

   Pillar data which is decrypted minion-side will still be securely transferred to the master, since the data sent between minion and master is encrypted with the master's public key.

2. Use the *decrypt_pillar* option. This is less flexible in that the pillar key passed on the CLI must be pre-configured on the master, but it doesn't require a keyring to be setup on the minion. One other caveat to this method is that pillar decryption on the master happens at the end of pillar compilation, so if the encrypted pillar data being passed on the CLI needs to be referenced by pillar or ext_pillar *during pillar compilation*, it *must* be decrypted minion-side.

Adding New Renderers for Decryption

Those looking to add new renderers for decryption should look at the *gpg* renderer for an example of how to do so. The function that performs the decryption should be recursive and be able to traverse a mutable type such as a dictionary, and modify the values in-place.

Once the renderer has been written, *decrypt_pillar_renderers* should be modified so that Salt allows it to be used for decryption.

If the renderer is being submitted upstream to the Salt project, the renderer should be added in salt/renderers/. Additionally, the following should be done:

- Both occurrences of *decrypt_pillar_renderers* in salt/config/__init__.py should be updated to include the name of the new renderer so that it is included in the default value for this config option.
- The documentation for the *decrypt_pillar_renderers* config option in the master config file and minion config file should be updated to show the correct new default value.
- The commented example for the *decrypt_pillar_renderers* config option in the master config template should be updated to show the correct new default value.

4.2.12 Master Config in Pillar

For convenience the data stored in the master configuration file can be made available in all minion's pillars. This makes global configuration of services and systems very easy but may not be desired if sensitive data is stored in the master configuration. This option is disabled by default.

To enable the master config from being added to the pillar set `pillar_opts` to `True` in the minion config file:

```
pillar_opts: True
```

4.2.13 Minion Config in Pillar

Minion configuration options can be set on pillars. Any option that you want to modify, should be in the first level of the pillars, in the same way you set the options in the config file. For example, to configure the MySQL root password to be used by MySQL Salt execution module, set the following pillar variable:

```
mysql.pass: hardtoguesspassword
```

4.2.14 Master Provided Pillar Error

By default if there is an error rendering a pillar, the detailed error is hidden and replaced with:

```
Rendering SLS 'my.sls' failed. Please see master log for details.
```

The error is protected because it's possible to contain templating data which would give that minion information it shouldn't know, like a password!

To have the master provide the detailed error that could potentially carry protected data set `pillar_safe_render_error` to `False`:

```
pillar_safe_render_error: False
```

Pillar Walkthrough

Note: This walkthrough assumes that the reader has already completed the initial Salt *walkthrough*.

Pillars are tree-like structures of data defined on the Salt Master and passed through to minions. They allow confidential, targeted data to be securely sent only to the relevant minion.

Note: Grains and Pillar are sometimes confused, just remember that Grains are data about a minion which is stored or generated from the minion. This is why information like the OS and CPU type are found in Grains. Pillar is information about a minion or many minions stored or generated on the Salt Master.

Pillar data is useful for:

Highly Sensitive Data: Information transferred via pillar is guaranteed to only be presented to the minions that are targeted, making Pillar suitable for managing security information, such as cryptographic keys and passwords.

Minion Configuration: Minion modules such as the execution modules, states, and returners can often be configured via data stored in pillar.

Variables: Variables which need to be assigned to specific minions or groups of minions can be defined in pillar and then accessed inside sls formulas and template files.

Arbitrary Data: Pillar can contain any basic data structure in dictionary format, so a key/value store can be defined making it easy to iterate over a group of values in sls formulas.

Pillar is therefore one of the most important systems when using Salt. This walkthrough is designed to get a simple Pillar up and running in a few minutes and then to dive into the capabilities of Pillar and where the data is available.

Setting Up Pillar

The pillar is already running in Salt by default. To see the minion's pillar data:

```
salt '*' pillar.items
```

Note: Prior to version 0.16.2, this function is named `pillar.data`. This function name is still supported for backwards compatibility.

By default, the contents of the master configuration file are not loaded into pillar for all minions. This default is stored in the `pillar_opts` setting, which defaults to `False`.

The contents of the master configuration file can be made available to minion pillar files. This makes global configuration of services and systems very easy, but note that this may not be desired or appropriate if sensitive data is stored in the master's configuration file. To enable the master configuration file to be available to a minion's pillar files, set `pillar_opts` to `True` in the minion configuration file.

Similar to the state tree, the pillar is comprised of sls files and has a top file. The default location for the pillar is in /srv/pillar.

Note: The pillar location can be configured via the `pillar_roots` option inside the master configuration file. It must not be in a subdirectory of the state tree or file_roots. If the pillar is under file_roots, any pillar targeting can be bypassed by minions.

To start setting up the pillar, the /srv/pillar directory needs to be present:

```
mkdir /srv/pillar
```

Now create a simple top file, following the same format as the top file used for states:

`/srv/pillar/top.sls`:

```
base:
  '*':
    - data
```

This top file associates the data.sls file to all minions. Now the `/srv/pillar/data.sls` file needs to be populated:

`/srv/pillar/data.sls`:

```
info: some data
```

To ensure that the minions have the new pillar data, issue a command to them asking that they fetch their pillars from the master:

```
salt '*' saltutil.refresh_pillar
```

Now that the minions have the new pillar, it can be retrieved:

```
salt '*' pillar.items
```

The key `info` should now appear in the returned pillar data.

More Complex Data

Unlike states, pillar files do not need to define **formulas**. This example sets up user data with a UID:

`/srv/pillar/users/init.sls`:

```
users:
  thatch: 1000
  shouse: 1001
  utahdave: 1002
  redbeard: 1003
```

Note: The same directory lookups that exist in states exist in pillar, so the file `users/init.sls` can be referenced with `users` in the *top file*.

The top file will need to be updated to include this sls file:

`/srv/pillar/top.sls`:

```
base:
  '*':
    - data
    - users
```

Now the data will be available to the minions. To use the pillar data in a state, you can use Jinja:

`/srv/salt/users/init.sls`

```
{% for user, uid in pillar.get('users', {}).items() %}
{{user}}:
  user.present:
    - uid: {{uid}}
{% endfor %}
```

This approach allows for users to be safely defined in a pillar and then the user data is applied in an sls file.

Parameterizing States With Pillar

Pillar data can be accessed in state files to customise behavior for each minion. All pillar (and grain) data applicable to each minion is substituted into the state files through templating before being run. Typical uses include setting directories appropriate for the minion and skipping states that don't apply.

A simple example is to set up a mapping of package names in pillar for separate Linux distributions:

`/srv/pillar/pkg/init.sls`:

```
pkgs:
  {% if grains['os_family'] == 'RedHat' %}
  apache: httpd
  vim: vim-enhanced
  {% elif grains['os_family'] == 'Debian' %}
  apache: apache2
  vim: vim
  {% elif grains['os'] == 'Arch' %}
  apache: apache
  vim: vim
  {% endif %}
```

The new `pkg` sls needs to be added to the top file:

`/srv/pillar/top.sls`:

```
base:
  '*':
    - data
    - users
    - pkg
```

Now the minions will auto map values based on respective operating systems inside of the pillar, so sls files can be safely parameterized:

`/srv/salt/apache/init.sls`:

```
apache:
  pkg.installed:
    - name: {{ pillar['pkgs']['apache'] }}
```

Or, if no pillar is available a default can be set as well:

Note: The function `pillar.get` used in this example was added to Salt in version 0.14.0

`/srv/salt/apache/init.sls`:

```
apache:
  pkg.installed:
    - name: {{ salt['pillar.get']('pkgs:apache', 'httpd') }}
```

In the above example, if the pillar value `pillar['pkgs']['apache']` is not set in the minion's pillar, then the default of `httpd` will be used.

Note: Under the hood, pillar is just a Python dict, so Python dict methods such as `get` and `items` can be used.

Pillar Makes Simple States Grow Easily

One of the design goals of pillar is to make simple sls formulas easily grow into more flexible formulas without refactoring or complicating the states.

A simple formula:

`/srv/salt/edit/vim.sls`:

```
vim:
  pkg.installed: []

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
    - mode: 644
    - user: root
    - group: root
    - require:
      - pkg: vim
```

Can be easily transformed into a powerful, parameterized formula:

`/srv/salt/edit/vim.sls`:

```
vim:
  pkg.installed:
    - name: {{ pillar['pkgs']['vim'] }}

/etc/vimrc:
  file.managed:
    - source: {{ pillar['vimrc'] }}
    - mode: 644
    - user: root
    - group: root
    - require:
      - pkg: vim
```

Where the vimrc source location can now be changed via pillar:

`/srv/pillar/edit/vim.sls`:

```
{% if grains['id'].startswith('dev') %}
vimrc: salt://edit/dev_vimrc
{% elif grains['id'].startswith('qa') %}
vimrc: salt://edit/qa_vimrc
{% else %}
vimrc: salt://edit/vimrc
{% endif %}
```

Ensuring that the right vimrc is sent out to the correct minions.

The pillar top file must include a reference to the new sls pillar file:

`/srv/pillar/top.sls`:

```
base:
  '*':
    - pkg
    - edit.vim
```

Setting Pillar Data on the Command Line

Pillar data can be set on the command line when running state.apply <salt.modules.state.apply_() like so:

```
salt '*' state.apply pillar='{"foo": "bar"}'
salt '*' state.apply my_sls_file pillar='{"hello": "world"}'
```

Nested pillar values can also be set via the command line:

```
salt '*' state.sls my_sls_file pillar='{"foo": {"bar": "baz"}}'
```

Lists can be passed via command line pillar data as follows:

```
salt '*' state.sls my_sls_file pillar='{"some_list": ["foo", "bar", "baz"]}'
```

Note: If a key is passed on the command line that already exists on the minion, the key that is passed in will overwrite the entire value of that key, rather than merging only the specified value set via the command line.

The example below will swap the value for vim with telnet in the previously specified list, notice the nested pillar dict:

```
salt '*' state.apply edit.vim pillar='{"pkgs": {"vim": "telnet"}}'
```

This will attempt to install telnet on your minions, feel free to uninstall the package or replace telnet value with anything else.

Note: Be aware that when sending sensitive data via pillar on the command-line that the publication containing that data will be received by all minions and will not be restricted to the targeted minions. This may represent a security concern in some cases.

More On Pillar

Pillar data is generated on the Salt master and securely distributed to minions. Salt is not restricted to the pillar sls files when defining the pillar but can retrieve data from external sources. This can be useful when information about an infrastructure is stored in a separate location.

Reference information on pillar and the external pillar interface can be found in the Salt documentation:

Pillar

Minion Config in Pillar

Minion configuration options can be set on pillars. Any option that you want to modify, should be in the first level of the pillars, in the same way you set the options in the config file. For example, to configure the MySQL root

password to be used by MySQL Salt execution module:

```
mysql.pass: hardtoguesspassword
```

This is very convenient when you need some dynamic configuration change that you want to be applied on the fly. For example, there is a chicken and the egg problem if you do this:

```
mysql-admin-passwd:
  mysql_user.present:
    - name: root
    - password: somepasswd

mydb:
  mysql_db.present
```

The second state will fail, because you changed the root password and the minion didn't notice it. Setting mysql.pass in the pillar, will help to sort out the issue. But always change the root admin password in the first place.

This is very helpful for any module that needs credentials to apply state changes: mysql, keystone, etc.

4.3 Targeting Minions

Targeting minions is specifying which minions should run a command or execute a state by matching against hostnames, or system information, or defined groups, or even combinations thereof.

For example the command `salt web1 apache.signal restart` to restart the Apache httpd server specifies the machine `web1` as the target and the command will only be run on that one minion.

Similarly when using States, the following *top file* specifies that only the `web1` minion should execute the contents of `webserver.sls`:

```
base:
  'web1':
    - webserver
```

The simple target specifications, glob, regex, and list will cover many use cases, and for some will cover all use cases, but more powerful options exist.

4.3.1 Targeting with Grains

The Grains interface was built into Salt to allow minions to be targeted by system properties. So minions running on a particular operating system can be called to execute a function, or a specific kernel.

Calling via a grain is done by passing the -G option to salt, specifying a grain and a glob expression to match the value of the grain. The syntax for the target is the grain key followed by a glob expression: ``os:Arch*''.

```
salt -G 'os:Fedora' test.ping
```

Will return True from all of the minions running Fedora.

To discover what grains are available and what the values are, execute the grains.item salt function:

```
salt '*' grains.items
```

More info on using targeting with grains can be found *here*.

4.3.2 Compound Targeting

New in version 0.9.5.

Multiple target interfaces can be used in conjunction to determine the command targets. These targets can then be combined using `and` or `or` statements. This is well defined with an example:

```
salt -C 'G@os:Debian and webser* or E@db.*' test.ping
```

In this example any minion who's id starts with `webser` and is running Debian, or any minion who's id starts with db will be matched.

The type of matcher defaults to glob, but can be specified with the corresponding letter followed by the `@` symbol. In the above example a grain is used with `G@` as well as a regular expression with `E@`. The `webser*` target does not need to be prefaced with a target type specifier because it is a glob.

More info on using compound targeting can be found *here*.

4.3.3 Node Group Targeting

New in version 0.9.5.

For certain cases, it can be convenient to have a predefined group of minions on which to execute commands. This can be accomplished using what are called *nodegroups*. Nodegroups allow for predefined compound targets to be declared in the master configuration file, as a sort of shorthand for having to type out complicated compound expressions.

```
nodegroups:
  group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com and bl*.domain.com'
  group2: 'G@os:Debian and foo.domain.com'
  group3: 'G@os:Debian and N@group1'
```

4.3.4 Advanced Targeting Methods

There are many ways to target individual minions or groups of minions in Salt:

Matching the `minion id`

Each minion needs a unique identifier. By default when a minion starts for the first time it chooses its FQDN (fully qualified domain name) as that identifier. The minion id can be overridden via the minion's *`id`* configuration setting.

Tip: minion id and minion keys

The *minion id* is used to generate the minion's public/private keys and if it ever changes the master must then accept the new key as though the minion was a new host.

Globbing

The default matching that Salt utilizes is `shell-style globbing` around the *minion id*. This also works for states in the *top file*.

Note: You must wrap **salt** calls that use globbing in single-quotes to prevent the shell from expanding the globs before Salt is invoked.

Match all minions:

```
salt '*' test.ping
```

Match all minions in the example.net domain or any of the example domains:

```
salt '*.example.net' test.ping
salt '*.example.*' test.ping
```

Match all the `webN` minions in the example.net domain (`web1.example.net`, `web2.example.net` … `webN.example.net`):

```
salt 'web?.example.net' test.ping
```

Match the `web1` through `web5` minions:

```
salt 'web[1-5]' test.ping
```

Match the `web1` and `web3` minions:

```
salt 'web[1,3]' test.ping
```

Match the `web-x`, `web-y`, and `web-z` minions:

```
salt 'web-[x-z]' test.ping
```

Note: For additional targeting methods please review the *compound matchers* documentation.

Regular Expressions

Minions can be matched using Perl-compatible `regular expressions` (which is globbing on steroids and a ton of caffeine).

Match both `web1-prod` and `web1-devel` minions:

```
salt -E 'web1-(prod|devel)' test.ping
```

When using regular expressions in a State's *top file*, you must specify the matcher as the first option. The following example executes the contents of `webserver.sls` on the above-mentioned minions.

```
base:
  'web1-(prod|devel)':
  - match: pcre
  - webserver
```

Lists

At the most basic level, you can specify a flat list of minion IDs:

```
salt -L 'web1,web2,web3' test.ping
```

Targeting using Grains

Grain data can be used when targeting minions.

For example, the following matches all CentOS minions:

```
salt -G 'os:CentOS' test.ping
```

Match all minions with 64-bit CPUs, and return number of CPU cores for each matching minion:

```
salt -G 'cpuarch:x86_64' grains.item num_cpus
```

Additionally, globs can be used in grain matches, and grains that are nested in a dictionary can be matched by adding a colon for each level that is traversed. For example, the following will match hosts that have a grain called `ec2_tags`, which itself is a dictionary with a key named `environment`, which has a value that contains the word `production`:

```
salt -G 'ec2_tags:environment:*production*'
```

Important: See *Is Targeting using Grain Data Secure?* for important security information.

Targeting using Pillar

Pillar data can be used when targeting minions. This allows for ultimate control and flexibility when targeting minions.

Note: To start using Pillar targeting it is required to make a Pillar data cache on Salt Master for each Minion via following commands: `salt '*' saltutil.refresh_pillar` or `salt '*' saltutil.sync_all`. Also Pillar data cache will be populated during the *highstate* run. Once Pillar data changes, you must refresh the cache by running above commands for this targeting method to work correctly.

Example:

```
salt -I 'somekey:specialvalue' test.ping
```

Like with *Grains*, it is possible to use globbing as well as match nested values in Pillar, by adding colons for each level that is being traversed. The below example would match minions with a pillar named `foo`, which is a dict containing a key `bar`, with a value beginning with `baz`:

```
salt -I 'foo:bar:baz*' test.ping
```

Subnet/IP Address Matching

Minions can easily be matched based on IP address, or by subnet (using CIDR notation).

```
salt -S 192.168.40.20 test.ping
salt -S 2001:db8::/64 test.ping
```

Ipcidr matching can also be used in compound matches

```
salt -C 'S@10.0.0.0/24 and G@os:Debian' test.ping
```

It is also possible to use in both pillar and state-matching

```
'172.16.0.0/12':
   - match: ipcidr
   - internal
```

Compound matchers

Compound matchers allow very granular minion targeting using any of Salt's matchers. The default matcher is a `glob` match, just as with CLI and *top file* matching. To match using anything other than a glob, prefix the match string with the appropriate letter from the table below, followed by an @ sign.

Letter	Match Type	Example	*Alt Delimiter?*
G	Grains glob	`G@os:Ubuntu`	Yes
E	PCRE Minion ID	`E@web\d+\.(dev|qa|prod)\.loc`	No
P	Grains PCRE	`P@os:(RedHat|Fedora|CentOS)`	Yes
L	List of minions	`L@minion1.example.com,minion3.domain.com or bl*.domain.com`	No
I	Pillar glob	`I@pdata:foobar`	Yes
J	Pillar PCRE	`J@pdata:^(foo|bar)$`	Yes
S	Subnet/IP address	`S@192.168.1.0/24` or `S@192.168.1.100`	No
R	Range cluster	`R@%foo.bar`	No

Matchers can be joined using boolean `and`, `or`, and `not` operators.

For example, the following string matches all Debian minions with a hostname that begins with `webserv`, as well as any minions that have a hostname which matches the `regular expression` `web-dc1-srv.*`:

```
salt -C 'webserv* and G@os:Debian or E@web-dc1-srv.*' test.ping
```

That same example expressed in a *top file* looks like the following:

```
base:
  'webserv* and G@os:Debian or E@web-dc1-srv.*':
    - match: compound
    - webserver
```

New in version 2015.8.0.

Excluding a minion based on its ID is also possible:

```
salt -C 'not web-dc1-srv' test.ping
```

Versions prior to 2015.8.0 a leading `not` was not supported in compound matches. Instead, something like the following was required:

```
salt -C '* and not G@kernel:Darwin' test.ping
```

Excluding a minion based on its ID was also possible:

```
salt -C '* and not web-dc1-srv' test.ping
```

Precedence Matching

Matchers can be grouped together with parentheses to explicitly declare precedence amongst groups.

```
salt -C '( ms-1 or G@id:ms-3 ) and G@id:ms-3' test.ping
```

Note: Be certain to note that spaces are required between the parentheses and targets. Failing to obey this rule may result in incorrect targeting!

Alternate Delimiters

New in version 2015.8.0.

Matchers that target based on a key value pair use a colon (`:`) as a delimiter. Matchers with a `Yes` in the `Alt Delimiters` column in the previous table support specifying an alternate delimiter character.

This is done by specifying an alternate delimiter character between the leading matcher character and the `@` pattern separator character. This avoids incorrect interpretation of the pattern in the case that `:` is part of the grain or pillar data structure traversal.

```
salt -C 'J|@foo|bar|^foo:bar$ or J!@gitrepo!https://github.com:example/project.git'
↪test.ping
```

Node groups

Nodegroups are declared using a compound target specification. The compound target documentation can be found *here*.

The *nodegroups* master config file parameter is used to define nodegroups. Here's an example nodegroup configuration within `/etc/salt/master`:

```
nodegroups:
  group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl*.domain.com'
  group2: 'G@os:Debian and foo.domain.com'
  group3: 'G@os:Debian and N@group1'
  group4:
    - 'G@foo:bar'
    - 'or'
    - 'G@foo:baz'
```

Note: The `L` within group1 is matching a list of minions, while the `G` in group2 is matching specific grains. See the *compound matchers* documentation for more details.

As of the 2017.7.0 release of Salt, group names can also be prepended with a dash. This brings the usage in line with many other areas of Salt. For example:

```
nodegroups:
  - group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl*.domain.com'
```

New in version 2015.8.0.

Note: Nodegroups can reference other nodegroups as seen in `group3`. Ensure that you do not have circular references. Circular references will be detected and cause partial expansion with a logged error message.

New in version 2015.8.0.

Compound nodegroups can be either string values or lists of string values. When the nodegroup is A string value will be tokenized by splitting on whitespace. This may be a problem if whitespace is necessary as part of a pattern. When a nodegroup is a list of strings then tokenization will happen for each list element as a whole.

To match a nodegroup on the CLI, use the `-N` command-line option:

```
salt -N group1 test.ping
```

Note: The `N@` classifier cannot be used in compound matches within the CLI or *top file*, it is only recognized in the `nodegroups` master config file parameter.

To match a nodegroup in your *top file*, make sure to put `-match: nodegroup` on the line directly following the nodegroup name.

```
base:
  group1:
    - match: nodegroup
    - webserver
```

Note: When adding or modifying nodegroups to a master configuration file, the master must be restarted for those changes to be fully recognized.

A limited amount of functionality, such as targeting with -N from the command-line may be available without a restart.

Defining Nodegroups as Lists of Minion IDs

A simple list of minion IDs would traditionally be defined like this:

```
nodegroups:
  group1: L@host1,host2,host3
```

They can now also be defined as a YAML list, like this:

```
nodegroups:
  group1:
    - host1
    - host2
    - host3
```

New in version 2016.11.0.

Batch Size

The `-b` (or `--batch-size`) option allows commands to be executed on only a specified number of minions at a time. Both percentages and finite numbers are supported.

```
salt '*' -b 10 test.ping

salt -G 'os:RedHat' --batch-size 25% apache.signal restart
```

This will only run test.ping on 10 of the targeted minions at a time and then restart apache on 25% of the minions matching `os:RedHat` at a time and work through them all until the task is complete. This makes jobs like rolling web server restarts behind a load balancer or doing maintenance on BSD firewalls using carp much easier with salt.

The batch system maintains a window of running minions, so, if there are a total of 150 minions targeted and the batch size is 10, then the command is sent to 10 minions, when one minion returns then the command is sent to one additional minion, so that the job is constantly running on 10 minions.

New in version 2016.3.

The `--batch-wait` argument can be used to specify a number of seconds to wait after a minion returns, before sending the command to a new minion.

SECO Range

SECO range is a cluster-based metadata store developed and maintained by Yahoo!

The Range project is hosted here:

https://github.com/ytoolshed/range

Learn more about range here:

https://github.com/ytoolshed/range/wiki/

Prerequisites

To utilize range support in Salt, a range server is required. Setting up a range server is outside the scope of this document. Apache modules are included in the range distribution.

With a working range server, cluster files must be defined. These files are written in YAML and define hosts contained inside a cluster. Full documentation on writing YAML range files is here:

https://github.com/ytoolshed/range/wiki/%22yamlfile%22-module-file-spec

Additionally, the Python seco range libraries must be installed on the salt master. One can verify that they have been installed correctly via the following command:

```
python -c 'import seco.range'
```

If no errors are returned, range is installed successfully on the salt master.

Preparing Salt

Range support must be enabled on the salt master by setting the hostname and port of the range server inside the master configuration file:

```
range_server: my.range.server.com:80
```

Following this, the master must be restarted for the change to have an effect.

Targeting with Range

Once a cluster has been defined, it can be targeted with a salt command by using the `-R` or `--range` flags.

For example, given the following range YAML file being served from a range server:

```
$ cat /etc/range/test.yaml
CLUSTER: host1..100.test.com
APPS:
  - frontend
  - backend
  - mysql
```

One might target host1 through host100 in the test.com domain with Salt as follows:

```
salt --range %test:CLUSTER test.ping
```

The following salt command would target three hosts: `frontend`, `backend`, and `mysql`:

```
salt --range %test:APPS test.ping
```

4.4 The Salt Mine

The Salt Mine is used to collect arbitrary data from Minions and store it on the Master. This data is then made available to all Minions via the *salt.modules.mine* module.

Mine data is gathered on the Minion and sent back to the Master where only the most recent data is maintained (if long term data is required use returners or the external job cache).

4.4.1 Mine vs Grains

Mine data is designed to be much more up-to-date than grain data. Grains are refreshed on a very limited basis and are largely static data. Mines are designed to replace slow peer publishing calls when Minions need data from other Minions. Rather than having a Minion reach out to all the other Minions for a piece of data, the Salt Mine, running on the Master, can collect it from all the Minions every *Mine Interval*, resulting in almost fresh data at any given time, with much less overhead.

4.4.2 Mine Functions

To enable the Salt Mine the `mine_functions` option needs to be applied to a Minion. This option can be applied via the Minion's configuration file, or the Minion's Pillar. The `mine_functions` option dictates what functions are being executed and allows for arguments to be passed in. The list of functions are available in the `salt.module`. If no arguments are passed, an empty list must be added like in the `test.ping` function in the example below:

```
mine_functions:
  test.ping: []
  network.ip_addrs:
```

```
    interface: eth0
    cidr: '10.0.0.0/8'
```

In the example above *salt.modules.network.ip_addrs* has additional filters to help narrow down the results. In the above example IP addresses are only returned if they are on a eth0 interface and in the 10.0.0.0/8 IP range.

Mine Functions Aliases

Function aliases can be used to provide friendly names, usage intentions or to allow multiple calls of the same function with different arguments. There is a different syntax for passing positional and key-value arguments. Mixing positional and key-value arguments is not supported.

New in version 2014.7.0.

```
mine_functions:
  network.ip_addrs: [eth0]
  networkplus.internal_ip_addrs: []
  internal_ip_addrs:
    mine_function: network.ip_addrs
    cidr: 192.168.0.0/16
  ip_list:
    - mine_function: grains.get
    - ip_interfaces
```

4.4.3 Mine Interval

The Salt Mine functions are executed when the Minion starts and at a given interval by the scheduler. The default interval is every 60 minutes and can be adjusted for the Minion via the `mine_interval` option:

```
mine_interval: 60
```

4.4.4 Mine in Salt-SSH

As of the 2015.5.0 release of salt, salt-ssh supports `mine.get`.

Because the Minions cannot provide their own `mine_functions` configuration, we retrieve the args for specified mine functions in one of three places, searched in the following order:

1. Roster data
2. Pillar
3. Master config

The `mine_functions` are formatted exactly the same as in normal salt, just stored in a different location. Here is an example of a flat roster containing `mine_functions`:

```
test:
  host: 104.237.131.248
  user: root
  mine_functions:
    cmd.run: ['echo "hello!"']
    network.ip_addrs:
      interface: eth0
```

Note: Because of the differences in the architecture of salt-ssh, `mine.get` calls are somewhat inefficient. Salt must make a new salt-ssh call to each of the Minions in question to retrieve the requested data, much like a publish call. However, unlike publish, it must run the requested function as a wrapper function, so we can retrieve the function args from the pillar of the Minion in question. This results in a non-trivial delay in retrieving the requested data.

4.4.5 Minions Targeting with Mine

The `mine.get` function supports various methods of *Minions targeting* to fetch Mine data from particular hosts, such as glob or regular expression matching on Minion id (name), grains, pillars and *compound matches*. See the *salt.modules.mine* module documentation for the reference.

Note: Pillar data needs to be cached on Master for pillar targeting to work with Mine. Read the note in *relevant section*.

4.4.6 Example

One way to use data from Salt Mine is in a State. The values can be retrieved via Jinja and used in the SLS file. The following example is a partial HAProxy configuration file and pulls IP addresses from all Minions with the ``web'' grain to add them to the pool of load balanced servers.

`/srv/pillar/top.sls`:

```
base:
  'G@roles:web':
    - web
```

`/srv/pillar/web.sls`:

```
mine_functions:
  network.ip_addrs: [eth0]
```

Then trigger the minions to refresh their pillar data by running:

```
salt '*' saltutil.refresh_pillar
```

Verify that the results are showing up in the pillar on the minions by executing the following and checking for `network.ip_addrs` in the output:

```
salt '*' pillar.items
```

Which should show that the function is present on the minion, but not include the output:

```
minion1.example.com:
    ----------
    mine_functions:
        ----------
        network.ip_addrs:
            - eth0
```

Mine data is typically only updated on the master every 60 minutes, this can be modified by setting:

`/etc/salt/minion.d/mine.conf`:

```
mine_interval: 5
```

To force the mine data to update immediately run:

```
salt '*' mine.update
```

Setup the *salt.states.file.managed* state in /srv/salt/haproxy.sls:

```
haproxy_config:
  file.managed:
    - name: /etc/haproxy/config
    - source: salt://haproxy_config
    - template: jinja
```

Create the Jinja template in /srv/salt/haproxy_config:

```
<...file contents snipped...>

{% for server, addrs in salt['mine.get']('roles:web', 'network.ip_addrs', tgt_type=
↪'grain') | dictsort() %}
server {{ server }} {{ addrs[0] }}:80 check
{% endfor %}

<...file contents snipped...>
```

In the above example, `server` will be expanded to the `minion_id`.

Note: The expr_form argument will be renamed to `tgt_type` in the 2017.7.0 release of Salt.

4.5 Runners

Salt runners are convenience applications executed with the salt-run command.

Salt runners work similarly to Salt execution modules however they execute on the Salt master itself instead of remote Salt minions.

A Salt runner can be a simple client call or a complex application.

See also:

The full list of runners

4.5.1 Writing Salt Runners

A Salt runner is written in a similar manner to a Salt execution module. Both are Python modules which contain functions and each public function is a runner which may be executed via the *salt-run* command.

For example, if a Python module named `test.py` is created in the runners directory and contains a function called `foo`, the `test` runner could be invoked with the following command:

```
# salt-run test.foo
```

Runners have several options for controlling output.

Any `print` statement in a runner is automatically also fired onto the master event bus where. For example:

```
def a_runner(outputter=None, display_progress=False):
    print('Hello world')
    ...
```

The above would result in an event fired as follows:

```
Event fired at Tue Jan 13 15:26:45 2015
*************************
Tag: salt/run/20150113152644070246/print
Data:
{'_stamp': '2015-01-13T15:26:45.078707',
 'data': 'hello',
  'outputter': 'pprint'}
```

A runner may also send a progress event, which is displayed to the user during runner execution and is also passed across the event bus if the `display_progress` argument to a runner is set to True.

A custom runner may send its own progress event by using the `__jid_event_.fire_event()` method as shown here:

```
if display_progress:
    __jid_event__.fire_event({'message': 'A progress message'}, 'progress')
```

The above would produce output on the console reading: `A progress message` as well as an event on the event similar to:

```
Event fired at Tue Jan 13 15:21:20 2015
*************************
Tag: salt/run/20150113152118341421/progress
Data:
{'_stamp': '2015-01-13T15:21:20.390053',
 'message': "A progress message"}
```

A runner could use the same approach to send an event with a customized tag onto the event bus by replacing the second argument (`progress`) with whatever tag is desired. However, this will not be shown on the command-line and will only be fired onto the event bus.

4.5.2 Synchronous vs. Asynchronous

A runner may be fired asynchronously which will immediately return control. In this case, no output will be display to the user if `salt-run` is being used from the command-line. If used programmatically, no results will be returned. If results are desired, they must be gathered either by firing events on the bus from the runner and then watching for them or by some other means.

Note: When running a runner in asynchronous mode, the `--progress` flag will not deliver output to the salt-run CLI. However, progress events will still be fired on the bus.

In synchronous mode, which is the default, control will not be returned until the runner has finished executing.

To add custom runners, put them in a directory and add it to *`runner_dirs`* in the master configuration file.

4.5.3 Examples

Examples of runners can be found in the Salt distribution:

https://github.com/saltstack/salt/blob/develop/salt/runners

A simple runner that returns a well-formatted list of the minions that are responding to Salt calls could look like this:

```
# Import salt modules
import salt.client

def up():
    '''
    Print a list of all of the minions that are up
    '''
    client = salt.client.LocalClient(__opts__['conf_file'])
    minions = client.cmd('*', 'test.ping', timeout=1)
    for minion in sorted(minions):
        print minion
```

4.6 Salt Engines

New in version 2015.8.0.

Salt Engines are long-running, external system processes that leverage Salt.

- Engines have access to Salt configuration, execution modules, and runners (`__opts__`, `__salt__`, and `__runners__`).
- Engines are executed in a separate process that is monitored by Salt. If a Salt engine stops, it is restarted automatically.
- Engines can run on the Salt master and on Salt minions.

Salt engines enhance and replace the external processes functionality.

4.6.1 Configuration

Salt engines are configured under an `engines` top-level section in your Salt master or Salt minion configuration. Provide a list of engines and parameters under this section.

```
engines:
  - logstash:
      host: log.my_network.com
      port: 5959
      proto: tcp
```

Salt engines must be in the Salt path, or you can add the `engines_dirs` option in your Salt master configuration with a list of directories under which Salt attempts to find Salt engines. This option should be formatted as a list of directories to search, such as:

```
engines_dirs:
  - /home/bob/engines
```

4.6.2 Writing an Engine

An example Salt engine, https://github.com/saltstack/salt/blob/develop/salt/engines/test.py, is available in the Salt source. To develop an engine, the only requirement is that your module implement the `start()` function.

4.7 Understanding YAML

The default renderer for SLS files is the YAML renderer. YAML is a markup language with many powerful features. However, Salt uses a small subset of YAML that maps over very commonly used data structures, like lists and dictionaries. It is the job of the YAML renderer to take the YAML data structure and compile it into a Python data structure for use by Salt.

Though YAML syntax may seem daunting and terse at first, there are only three very simple rules to remember when writing YAML for SLS files.

4.7.1 Rule One: Indentation

YAML uses a fixed indentation scheme to represent relationships between data layers. Salt requires that the indentation for each level consists of exactly two spaces. Do not use tabs.

4.7.2 Rule Two: Colons

Python dictionaries are, of course, simply key-value pairs. Users from other languages may recognize this data type as hashes or associative arrays.

Dictionary keys are represented in YAML as strings terminated by a trailing colon. Values are represented by either a string following the colon, separated by a space:

```
my_key: my_value
```

In Python, the above maps to:

```
{'my_key': 'my_value'}
```

Alternatively, a value can be associated with a key through indentation.

```
my_key:
  my_value
```

Note: The above syntax is valid YAML but is uncommon in SLS files because most often, the value for a key is not singular but instead is a *list* of values.

In Python, the above maps to:

```
{'my_key': 'my_value'}
```

Dictionaries can be nested:

```
first_level_dict_key:
  second_level_dict_key: value_in_second_level_dict
```

And in Python:

```
{
    'first_level_dict_key': {
        'second_level_dict_key': 'value_in_second_level_dict'
    }
}
```

4.7.3 Rule Three: Dashes

To represent lists of items, a single dash followed by a space is used. Multiple items are a part of the same list as a function of their having the same level of indentation.

```
- list_value_one
- list_value_two
- list_value_three
```

Lists can be the value of a key-value pair. This is quite common in Salt:

```
my_dictionary:
  - list_value_one
  - list_value_two
  - list_value_three
```

In Python, the above maps to:

```
{'my_dictionary': ['list_value_one', 'list_value_two', 'list_value_three']}
```

4.7.4 Learning More

One easy way to learn more about how YAML gets rendered into Python data structures is to use an online YAML parser to see the Python output.

One excellent choice for experimenting with YAML parsing is: http://yaml-online-parser.appspot.com/

4.7.5 Templating

Jinja statements and expressions are allowed by default in SLS files. See *Understanding Jinja*.

4.8 Understanding Jinja

Jinja is the default templating language in SLS files.

4.8.1 Jinja in States

Jinja is evaluated before YAML, which means it is evaluated before the States are run.

The most basic usage of Jinja in state files is using control structures to wrap conditional or redundant state elements:

```
{% if grains['os'] != 'FreeBSD' %}
tcsh:
    pkg:
        - installed
{% endif %}

motd:
  file.managed:
    {% if grains['os'] == 'FreeBSD' %}
    - name: /etc/motd
    {% elif grains['os'] == 'Debian' %}
    - name: /etc/motd.tail
    {% endif %}
    - source: salt://motd
```

In this example, the first if block will only be evaluated on minions that aren't running FreeBSD, and the second block changes the file name based on the *os* grain.

Writing **if-else** blocks can lead to very redundant state files however. In this case, using *pillars*, or using a previously defined variable might be easier:

```
{% set motd = ['/etc/motd'] %}
{% if grains['os'] == 'Debian' %}
  {% set motd = ['/etc/motd.tail', '/var/run/motd'] %}
{% endif %}

{% for motdfile in motd %}
{{ motdfile }}:
  file.managed:
    - source: salt://motd
{% endfor %}
```

Using a variable set by the template, the for loop will iterate over the list of MOTD files to update, adding a state block for each file.

The filter_by function can also be used to set variables based on grains:

```
{% set auditd = salt['grains.filter_by']({
'RedHat': { 'package': 'audit' },
'Debian': { 'package': 'auditd' },
}) %}
```

4.8.2 Include and Import

Includes and imports can be used to share common, reusable state configuration between state files and between files.

```
{% from 'lib.sls' import test %}
```

This would import the `test` template variable or macro, not the `test` state element, from the file `lib.sls`. In the case that the included file performs checks against grains, or something else that requires context, passing the context into the included file is required:

```
{% from 'lib.sls' import test with context %}
```

Including Context During Include/Import

By adding `with context` to the include/import directive, the current context can be passed to an included/imported template.

```
{% import 'openssl/vars.sls' as ssl with context %}
```

4.8.3 Macros

Macros are helpful for eliminating redundant code. Macros are most useful as mini-templates to repeat blocks of strings with a few parameterized variables. Be aware that stripping whitespace from the template block, as well as contained blocks, may be necessary to emulate a variable return from the macro.

```
# init.sls
{% from 'lib.sls' import pythonpkg with context %}

python-virtualenv:
  pkg.installed:
    - name: {{ pythonpkg('virtualenv') }}

python-fabric:
  pkg.installed:
    - name: {{ pythonpkg('fabric') }}
```

```
# lib.sls
{% macro pythonpkg(pkg) -%}
  {%- if grains['os'] == 'FreeBSD' -%}
    py27-{{ pkg }}
  {%- elif grains['os'] == 'Debian' -%}
    python-{{ pkg }}
  {%- endif -%}
{%- endmacro %}
```

This would define a macro that would return a string of the full package name, depending on the packaging system's naming convention. The whitespace of the macro was eliminated, so that the macro would return a string without line breaks, using whitespace control.

4.8.4 Template Inheritance

Template inheritance works fine from state files and files. The search path starts at the root of the state tree or pillar.

4.8.5 Errors

Saltstack allows raising custom errors using the `raise` jinja function.

```
{{ raise('Custom Error') }}
```

When rendering the template containing the above statement, a `TemplateError` exception is raised, causing the rendering to fail with the following message:

```
TemplateError: Custom Error
```

4.8.6 Filters

Saltstack extends builtin filters with these custom filters:

strftime

Converts any time related object into a time based string. It requires valid strftime directives. An exhaustive list can be found here in the Python documentation.

```
{% set curtime = None | strftime() %}
```

Fuzzy dates require the timelib Python module is installed.

```
{{ "2002/12/25"|strftime("%y") }}
{{ "1040814000"|strftime("%Y-%m-%d") }}
{{ datetime|strftime("%u") }}
{{ "tomorrow"|strftime }}
```

sequence

Ensure that parsed data is a sequence.

yaml_encode

Serializes a single object into a YAML scalar with any necessary handling for escaping special characters. This will work for any scalar YAML data type: ints, floats, timestamps, booleans, strings, unicode. It will *not* work for multi-objects such as sequences or maps.

```
{%- set bar = 7 %}
{%- set baz = none %}
{%- set zip = true %}
{%- set zap = 'The word of the day is "salty"' %}

{%- load_yaml as foo %}
bar: {{ bar|yaml_encode }}
baz: {{ baz|yaml_encode }}
baz: {{ zip|yaml_encode }}
baz: {{ zap|yaml_encode }}
{%- endload %}
```

In the above case `{{ bar }}` and `{{ foo.bar }}` should be identical and `{{ baz }}` and `{{ foo.baz }}` should be identical.

yaml_dquote

Serializes a string into a properly-escaped YAML double-quoted string. This is useful when the contents of a string are unknown and may contain quotes or unicode that needs to be preserved. The resulting string will be emitted with opening and closing double quotes.

```
{%- set bar = '"The quick brown fox . . ."' %}
{%- set baz = 'The word of the day is "salty".' %}

{%- load_yaml as foo %}
```

```
bar: {{ bar|yaml_dquote }}
baz: {{ baz|yaml_dquote }}
{%- endload %}
```

In the above case `{{ bar }}` and `{{ foo.bar }}` should be identical and `{{ baz }}` and `{{ foo.baz }}` should be identical. If variable contents are not guaranteed to be a string then it is better to use `yaml_encode` which handles all YAML scalar types.

yaml_squote

Similar to the `yaml_dquote` filter but with single quotes. Note that YAML only allows special escapes inside double quotes so `yaml_squote` is not nearly as useful (viz. you likely want to use `yaml_encode` or `yaml_dquote`).

to_bool

New in version 2017.7.0.

Returns the logical value of an element.

Example:

```
{{ 'yes' | to_bool }}
{{ 'true' | to_bool }}
{{ 1 | to_bool }}
{{ 'no' | to_bool }}
```

Will be rendered as:

```
True
True
True
False
```

exactly_n_true

New in version 2017.7.0.

Tests that exactly N items in an iterable are ``truthy'' (neither None, False, nor 0).

Example:

```
{{ ['yes', 0, False, 'True'] | exactly_n_true(2) }}
```

Returns:

```
True
```

exactly_one_true

New in version 2017.7.0.

Tests that exactly one item in an iterable is ``truthy'' (neither None, False, nor 0).

Example:

```
{{ ['yes', False, 0, None] | exactly_one_true }}
```

Returns:

```
True
```

quote

New in version 2017.7.0.

This text will be wrapped in quotes.

regex_search

New in version 2017.7.0.

Scan through string looking for a location where this regular expression produces a match. Returns `None` in case there were no matches found

Example:

```
{{ 'abcdefabcdef' | regex_search('BC(.*)', ignorecase=True) }}
```

Returns:

```
('defabcdef',)
```

regex_match

New in version 2017.7.0.

If zero or more characters at the beginning of string match this regular expression, otherwise returns `None`.

Example:

```
{{ 'abcdefabcdef' | regex_match('BC(.*)', ignorecase=True) }}
```

Returns:

```
None
```

uuid

New in version 2017.7.0.

Return a UUID.

Example:

```
{{ 'random' | uuid }}
```

Returns:

```
3652b285-26ad-588e-a5dc-c2ee65edc804
```

is_list

New in version 2017.7.0.

Return if an object is list.

Example:

```
{{ [1, 2, 3] | is_list }}
```

Returns:

```
True
```

is_iter

New in version 2017.7.0.

Return if an object is iterable.

Example:

```
{{ [1, 2, 3] | is_iter }}
```

Returns:

```
True
```

min

New in version 2017.7.0.

Return the minimum value from a list.

Example:

```
{{ [1, 2, 3] | min }}
```

Returns:

```
1
```

max

New in version 2017.7.0.

Returns the maximum value from a list.

Example:

```
{{ [1, 2, 3] | max }}
```

Returns:

```
3
```

avg

New in version 2017.7.0.

Returns the average value of the elements of a list

Example:

```
{{ [1, 2, 3] | avg }}
```

Returns:

```
2
```

union

New in version 2017.7.0.

Return the union of two lists.

Example:

```
{{ [1, 2, 3] | union([2, 3, 4]) | join(', ') }}
```

Returns:

```
1, 2, 3, 4
```

intersect

New in version 2017.7.0.

Return the intersection of two lists.

Example:

```
{{ [1, 2, 3] | intersect([2, 3, 4]) | join(', ') }}
```

Returns:

```
2, 3
```

difference

New in version 2017.7.0.

Return the difference of two lists.

Example:

```
{{ [1, 2, 3] | difference([2, 3, 4]) | join(', ') }}
```

Returns:

```
1
```

symmetric_difference

New in version 2017.7.0.

Return the symmetric difference of two lists.

Example:

```
{{ [1, 2, 3] | symmetric_difference([2, 3, 4]) | join(', ') }}
```

Returns:

```
1, 4
```

is_sorted

New in version 2017.7.0.

Return is an iterable object is already sorted.

Example:

```
{{ [1, 2, 3] | is_sorted }}
```

Returns:

```
True
```

compare_lists

New in version 2017.7.0.

Compare two lists and return a dictionary with the changes.

Example:

```
{{ [1, 2, 3] | compare_lists([1, 2, 4]) }}
```

Returns:

```
{'new': 4, 'old': 3}
```

compare_dicts

New in version 2017.7.0.

Compare two dictionaries and return a dictionary with the changes.

Example:

```
{{ {'a': 'b'} | compare_lists({'a': 'c'}) }}
```

Returns:

```
{'a': {'new': 'c', 'old': 'b'}}
```

is_hex

New in version 2017.7.0.

Return True if the value is hexazecimal.

Example:

```
{{ '0xabcd' | is_hex }}
{{ 'xyzt' | is_hex }}
```

Returns:

```
True
False
```

contains_whitespace

New in version 2017.7.0.

Return True if a text contains whitespaces.

Example:

```
{{ 'abcd' | contains_whitespace }}
{{ 'ab cd' | contains_whitespace }}
```

Returns:

```
False
True
```

substring_in_list

New in version 2017.7.0.

Return is a substring is found in a list of string values.

Example:

```
{{ 'abcd' | substring_in_list(['this', 'is', 'an abcd example']) }}
```

Returns:

```
True
```

check_whitelist_blacklist

New in version 2017.7.0.

Check a whitelist and/or blacklist to see if the value matches it.

This filter can be used with either a whitelist or a blacklist individually, or a whitelist and a blacklist can be passed simultaneously.

If whitelist is used alone, value membership is checked against the whitelist only. If the value is found, the function returns `True`. Otherwise, it returns `False`.

If blacklist is used alone, value membership is checked against the blacklist only. If the value is found, the function returns `False`. Otherwise, it returns `True`.

If both a whitelist and a blacklist are provided, value membership in the blacklist will be examined first. If the value is not found in the blacklist, then the whitelist is checked. If the value isn't found in the whitelist, the function returns `False`.

Whitelist Example:

```
{{ 5 | check_whitelist_blacklist(whitelist=[5, 6, 7]) }}
```

Returns:

```
True
```

Blacklist Example:

```
{{ 5 | check_whitelist_blacklist(blacklist=[5, 6, 7]) }}
```

```
False
```

date_format

New in version 2017.7.0.

Converts unix timestamp into human-readable string.

Example:

```
{{ 1457456400 | date_format }}
{{ 1457456400 | date_format('%d.%m.%Y %H:%M') }}
```

Returns:

```
2017-03-08
08.03.2017 17:00
```

to_num

New in version 2017.7.0.

New in version 2018.3.0: Renamed from `str_to_num` to `to_num`.

Converts a string to its numerical value.

Example:

```
{{ '5' | to_num }}
```

Returns:

```
5
```

to_bytes

New in version 2017.7.0.

Converts string-type object to bytes.

Example:

```
{{ 'wall of text' | to_bytes }}
```

Note: This option may have adverse effects when using the default renderer, `yaml_jinja`. This is due to the fact that YAML requires proper handling in regard to special characters. Please see the section on *YAML ASCII support* in the *YAML Idiosyncracies* documentation for more information.

json_encode_list

New in version 2017.7.0.

New in version 2018.3.0: Renamed from `json_decode_list` to `json_encode_list`. When you encode something you get bytes, and when you decode, you get your locale's encoding (usually a `unicode` type). This filter was incorrectly-named when it was added. `json_decode_list` will be supported until the Neon release.

Deprecated since version 2018.3.3,Fluorine: The *tojson* filter accomplishes what this filter was designed to do, making this filter redundant.

Recursively encodes all string elements of the list to bytes.

Example:

```
{{ [1, 2, 3] | json_encode_list }}
```

Returns:

```
[1, 2, 3]
```

json_encode_dict

New in version 2017.7.0.

New in version 2018.3.0: Renamed from `json_decode_dict` to `json_encode_dict`. When you encode something you get bytes, and when you decode, you get your locale's encoding (usually a `unicode` type). This filter was incorrectly-named when it was added. `json_decode_dict` will be supported until the Neon release.

Deprecated since version 2018.3.3,Fluorine: The *tojson* filter accomplishes what this filter was designed to do, making this filter redundant.

Recursively encodes all string items in the dictionary to bytes.

Example:

Assuming that `pillar['foo']` contains `{u'a': u'\u0414'}`, and your locale is `en_US.UTF-8`:

```
{{ pillar['foo'] | json_encode_dict }}
```

Returns:

```
{'a': '\xd0\x94'}
```

tojson

New in version 2018.3.3,Fluorine.

Dumps a data structure to JSON.

This filter was added to provide this functionality to hosts which have a Jinja release older than version 2.9 installed. If Jinja 2.9 or newer is installed, then the upstream version of the filter will be used. See the upstream docs for more information.

random_hash

New in version 2017.7.0.

New in version 2018.3.0: Renamed from `rand_str` to `random_hash` to more accurately describe what the filter does. `rand_str` will be supported until the Neon release.

Generates a random number between 1 and the number passed to the filter, and then hashes it. The default hash type is the one specified by the minion's *hash_type* config option, but an alternate hash type can be passed to the filter as an argument.

Example:

```
{% set num_range = 99999999 %}
{{ num_range | random_hash }}
{{ num_range | random_hash('sha512') }}
```

Returns:

```
43ec517d68b6edd3015b3edc9a11367b
d94a45acd81f8e3107d237dbc0d5d195f6a52a0d188bc0284c0763ece1eac9f9496fb6a531a296074c87b3540398dace1222b
```

md5

New in version 2017.7.0.

Return the md5 digest of a string.

Example:

```
{{ 'random' | md5 }}
```

Returns:

```
7ddf32e17a6ac5ce04a8ecbf782ca509
```

sha256

New in version 2017.7.0.

Return the sha256 digest of a string.

Example:

```
{{ 'random' | sha256 }}
```

Returns:

```
a441b15fe9a3cf56661190a0b93b9dec7d04127288cc87250967cf3b52894d11
```

sha512

New in version 2017.7.0.

Return the sha512 digest of a string.

Example:

```
{{ 'random' | sha512 }}
```

Returns:

```
811a90e1c8e86c7b4c0eef5b2c0bf0ec1b19c4b1b5a242e6455be93787cb473cb7bc9b0fdeb960d00d5c6881c2094dd63c5c
```

base64_encode

New in version 2017.7.0.

Encode a string as base64.

Example:

```
{{ 'random' | base64_encode }}
```

Returns:

```
cmFuZG9t
```

base64_decode

New in version 2017.7.0.

Decode a base64-encoded string.

```
{{ 'Z2V0IHNhbHRlZA==' | base64_decode }}
```

Returns:

```
get salted
```

hmac

New in version 2017.7.0.

Verify a challenging hmac signature against a string / shared-secret. Returns a boolean value.

Example:

```
{{ 'get salted' | hmac('shared secret', 'eBWf9bstXg+NiP5AOwppB5HMvZiYMPzEM9W5YMm/AmQ=
↪') }}
```

Returns:

```
True
```

http_query

New in version 2017.7.0.

Return the HTTP reply object from a URL.

Example:

```
{{ 'http://jsonplaceholder.typicode.com/posts/1' | http_query }}
```

Returns:

```
{
  'body': '{
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi option
↪reprehenderit",
    "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et
↪cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem
↪eveniet architecto"
  }'
}
```

traverse

New in version 2018.3.3.

Traverse a dict or list using a colon-delimited target string. The target `foo:bar:0' will return data['foo']['bar'][0] if this value exists, and will otherwise return the provided default value.

Example:

```
{{ {'a1': {'b1': {'c1': 'foo'}}, 'a2': 'bar'} | traverse('a1:b1', 'default') }}
```

Returns:

```
{'c1': 'foo'}
```

```
{{ {'a1': {'b1': {'c1': 'foo'}}, 'a2': 'bar'} | traverse('a2:b2', 'default') }}
```

Returns:

```
'default'
```

Networking Filters

The following networking-related filters are supported:

is_ip

New in version 2017.7.0.

Return if a string is a valid IP Address.

```
{{ '192.168.0.1' | is_ip }}
```

Additionally accepts the following options:

- global
- link-local
- loopback
- multicast
- private
- public
- reserved
- site-local
- unspecified

Example - test if a string is a valid loopback IP address.

```
{{ '192.168.0.1' | is_ip(options='loopback') }}
```

is_ipv4

New in version 2017.7.0.

Returns if a string is a valid IPv4 address. Supports the same options as `is_ip`.

```
{{ '192.168.0.1' | is_ipv4 }}
```

is_ipv6

New in version 2017.7.0.

Returns if a string is a valid IPv6 address. Supports the same options as `is_ip`.

```
{{ 'fe80::' | is_ipv6 }}
```

ipaddr

New in version 2017.7.0.

From a list, returns only valid IP entries. Supports the same options as `is_ip`. The list can contains also IP interfaces/networks.

Example:

```
{{ ['192.168.0.1', 'foo', 'bar', 'fe80::'] | ipaddr }}
```

Returns:

```
['192.168.0.1', 'fe80::']
```

ipv4

New in version 2017.7.0.

From a list, returns only valid IPv4 entries. Supports the same options as `is_ip`. The list can contains also IP interfaces/networks.

Example:

```
{{ ['192.168.0.1', 'foo', 'bar', 'fe80::'] | ipv4 }}
```

Returns:

```
['192.168.0.1']
```

ipv6

New in version 2017.7.0.

From a list, returns only valid IPv6 entries. Supports the same options as `is_ip`. The list can contains also IP interfaces/networks.

Example:

```
{{ ['192.168.0.1', 'foo', 'bar', 'fe80::'] | ipv6 }}
```

Returns:

```
['fe80::']
```

network_hosts

New in version 2017.7.0.

Return the list of hosts within a networks. This utility works for both IPv4 and IPv6.

Note: When running this command with a large IPv6 network, the command will take a long time to gather all of the hosts.

Example:

```
{{ '192.168.0.1/30' | network_hosts }}
```

Returns:

```
['192.168.0.1', '192.168.0.2']
```

network_size

New in version 2017.7.0.

Return the size of the network. This utility works for both IPv4 and IPv6.

Example:

```
{{ '192.168.0.1/8' | network_size }}
```

Returns:

```
16777216
```

gen_mac

New in version 2017.7.0.

Generates a MAC address with the defined OUI prefix.

Common prefixes:

- `00:16:3E` -- Xen
- `00:18:51` -- OpenVZ
- `00:50:56` -- VMware (manually generated)
- `52:54:00` -- QEMU/KVM
- `AC:DE:48` -- PRIVATE

Example:

```
{{ '00:50' | gen_mac }}
```

Returns:

```
00:50:71:52:1C
```

mac_str_to_bytes

New in version 2017.7.0.

Converts a string representing a valid MAC address to bytes.

Example:

```
{{ '00:11:22:33:44:55' | mac_str_to_bytes }}
```

Note: This option may have adverse effects when using the default renderer, `yaml_jinja`. This is due to the fact that YAML requires proper handling in regard to special characters. Please see the section on *YAML ASCII support* in the *YAML Idiosyncracies* documentation for more information.

dns_check

New in version 2017.7.0.

Return the ip resolved by dns, but do not exit on failure, only raise an exception. Obeys system preference for IPv4/6 address resolution.

Example:

```
{{ 'www.google.com' | dns_check(port=443) }}
```

Returns:

```
'172.217.3.196'
```

File filters

is_text_file

New in version 2017.7.0.

Return if a file is text.

Uses heuristics to guess whether the given file is text or binary, by reading a single block of bytes from the file. If more than 30% of the chars in the block are non-text, or there are NUL ('x00') bytes in the block, assume this is a binary file.

Example:

```
{{ '/etc/salt/master' | is_text_file }}
```

Returns:

```
True
```

is_binary_file

New in version 2017.7.0.

Return if a file is binary.

Detects if the file is a binary, returns bool. Returns True if the file is a bin, False if the file is not and None if the file is not available.

Example:

```
{{ '/etc/salt/master' | is_binary_file }}
```

Returns:

```
False
```

is_empty_file

New in version 2017.7.0.

Return if a file is empty.

Example:

```
{{ '/etc/salt/master' | is_empty_file }}
```

Returns:

```
False
```

file_hashsum

New in version 2017.7.0.

Return the hashsum of a file.

Example:

```
{{ '/etc/salt/master' | file_hashsum }}
```

Returns:

```
02d4ef135514934759634f10079653252c7ad594ea97bd385480c532bca0fdda
```

list_files

New in version 2017.7.0.

Return a recursive list of files under a specific path.

Example:

```
{{ '/etc/salt/' | list_files | join('\n') }}
```

Returns:

```
/etc/salt/master
/etc/salt/proxy
/etc/salt/minion
/etc/salt/pillar/top.sls
/etc/salt/pillar/device1.sls
```

path_join

New in version 2017.7.0.

Joins absolute paths.

Example:

```
{{ '/etc/salt/' | path_join('pillar', 'device1.sls') }}
```

Returns:

```
/etc/salt/pillar/device1.sls
```

which

New in version 2017.7.0.

Python clone of /usr/bin/which.

Example:

```
{{ 'salt-master' | which }}
```

Returns:

```
/usr/local/salt/virtualenv/bin/salt-master
```

4.8.7 Tests

Saltstack extends builtin tests with these custom tests:

equalto

Tests the equality between two values.

Can be used in an `if` statement directly:

```
{% if 1 is equalto(1) %}
    < statements >
{% endif %}
```

If clause evaluates to `True`

or with the `selectattr` filter:

```
{{ [{'value': 1}, {'value': 2} , {'value': 3}] | selectattr('value', 'equalto', 3) |
↪list }}
```

Returns:

```
[{'value': 3}]
```

match

Tests that a string matches the regex passed as an argument.

Can be used in a `if` statement directly:

```
{% if 'a' is match('[a-b]') %}
    < statements >
{% endif %}
```

If clause evaluates to True

or with the selectattr filter:

```
{{ [{'value': 'a'}, {'value': 'b'}, {'value': 'c'}] | selectattr('value', 'match', '[b-
↪e]') | list }}
```

Returns:

```
[{'value': 'b'}, {'value': 'c'}]
```

Test supports additional optional arguments: ignorecase, multiline

Escape filters

regex_escape

New in version 2017.7.0.

Allows escaping of strings so they can be interpreted literally by another function.

Example:

```
regex_escape = {{ 'https://example.com?foo=bar%20baz' | regex_escape }}
```

will be rendered as:

```
regex_escape = https\:\/\/example\.com\?foo\=bar\%20baz
```

Set Theory Filters

unique

New in version 2017.7.0.

Performs set math using Jinja filters.

Example:

```
unique = {{ ['foo', 'foo', 'bar'] | unique }}
```

will be rendered as:

```
unique = ['foo', 'bar']
```

4.8.8 Jinja in Files

Jinja can be used in the same way in managed files:

```
# redis.sls
/etc/redis/redis.conf:
    file.managed:
        - source: salt://redis.conf
        - template: jinja
        - context:
            bind: 127.0.0.1
```

```
# lib.sls
{% set port = 6379 %}
```

```
# redis.conf
{% from 'lib.sls' import port with context %}
port {{ port }}
bind {{ bind }}
```

As an example, configuration was pulled from the file context and from an external template file.

Note: Macros and variables can be shared across templates. They should not be starting with one or more underscores, and should be managed by one of the following tags: *macro*, *set*, *load_yaml*, *load_json*, *import_yaml* and *import_json*.

4.8.9 Escaping Jinja

Occasionally, it may be necessary to escape Jinja syntax. There are two ways to do this in Jinja. One is escaping individual variables or strings and the other is to escape entire blocks.

To escape a string commonly used in Jinja syntax such as `{{`, you can use the following syntax:

```
{{ '{{' }}
```

For larger blocks that contain Jinja syntax that needs to be escaped, you can use raw blocks:

```
{% raw %}
    some text that contains jinja characters that need to be escaped
{% endraw %}
```

See the Escaping section of Jinja's documentation to learn more.

A real-word example of needing to use raw tags to escape a larger block of code is when using `file.managed` with the `contents_pillar` option to manage files that contain something like consul-template, which shares a syntax subset with Jinja. Raw blocks are necessary here because the Jinja in the pillar would be rendered before the file.managed is ever called, so the Jinja syntax must be escaped:

```
{% raw %}
- contents_pillar: |
    job "example-job" {
      <snipped>
      task "example" {
          driver = "docker"

          config {
              image = "docker-registry.service.consul:5000/example-job:{{key "nomad/
↪jobs/example-job/version"}}"
      <snipped>
{% endraw %}
```

4.8.10 Calling Salt Functions

The Jinja renderer provides a shorthand lookup syntax for the `salt` dictionary of *execution function*.

New in version 2014.7.0.

```
# The following two function calls are equivalent.
{{ salt['cmd.run']('whoami') }}
{{ salt.cmd.run('whoami') }}
```

4.8.11 Debugging

The `show_full_context` function can be used to output all variables present in the current Jinja context.

New in version 2014.7.0.

```
Context is: {{ show_full_context()|yaml(False) }}
```

Logs

New in version 2017.7.0.

Yes, in Salt, one is able to debug a complex Jinja template using the logs. For example, making the call:

```
{%- do salt.log.error('testing jinja logging') -%}
```

Will insert the following message in the minion logs:

```
2017-02-01 01:24:40,728 [salt.module.logmod][ERROR   ][3779] testing jinja logging
```

4.8.12 Python Methods

A powerful feature of jinja that is only hinted at in the official jinja documentation is that you can use the native python methods of the variable type. Here is the python documentation for string methods.

```
{% set hostname,domain = grains.id.partition('.')[::2] %}{{ hostname }}
```

```
{% set strings = grains.id.split('-') %}{{ strings[0] }}
```

4.8.13 Custom Execution Modules

Custom execution modules can be used to supplement or replace complex Jinja. Many tasks that require complex looping and logic are trivial when using Python in a Salt execution module. Salt execution modules are easy to write and distribute to Salt minions.

Functions in custom execution modules are available in the Salt execution module dictionary just like the built-in execution modules:

```
{{ salt['my_custom_module.my_custom_function']() }}
```

- *How to Convert Jinja Logic to an Execution Module*
- *Writing Execution Modules*

4.8.14 Custom Jinja filters

Given that all execution modules are available in the Jinja template, one can easily define a custom module as in the previous paragraph and use it as a Jinja filter. However, please note that it will not be accessible through the pipe.

For example, instead of:

```
{{ my_variable | my_jinja_filter }}
```

The user will need to define `my_jinja_filter` function under an extension module, say `my_filters` and use as:

```
{{ salt.my_filters.my_jinja_filter(my_variable) }}
```

The greatest benefit is that you are able to access thousands of existing functions, e.g.:

- get the DNS AAAA records for a specific address using the *dnsutil*:

  ```
  {{ salt.dnsutil.AAAA('www.google.com') }}
  ```

- retrieve a specific field value from a `Redis` hash:

  ```
  {{ salt.redis.hget('foo_hash', 'bar_field') }}
  ```

- get the routes to `0.0.0.0/0` using the *NAPALM route*:

  ```
  {{ salt.route.show('0.0.0.0/0') }}
  ```

4.9 Tutorials Index

4.9.1 Autoaccept minions from Grains

New in version 2018.3.0.

To automatically accept minions based on certain characteristics, e.g. the `uuid` you can specify certain grain values on the salt master. Minions with matching grains will have their keys automatically accepted.

1. Configure the autosign_grains_dir in the master config file:

```
autosign_grains_dir: /etc/salt/autosign_grains
```

2. Configure the grain values to be accepted

Place a file named like the grain in the autosign_grains_dir and write the values that should be accepted automatically inside that file. For example to automatically accept minions based on their `uuid` create a file named `/etc/salt/autosign_grains/uuid`:

```
8f7d68e2-30c5-40c6-b84a-df7e978a03ee
1d3c5473-1fbc-479e-b0c7-877705a0730f
```

The master is now setup to accept minions with either of the two specified uuids. Multiple values must always be written into separate lines. Lines starting with a # are ignored.

3. Configure the minion to send the specific grains to the master in the minion config file:

```
autosign_grains:
  - uuid
```

Now you should be able to start salt-minion and run `salt-call state.apply` or any other salt commands that require master authentication.

4.9.2 Salt as a Cloud Controller

In Salt 0.14.0, an advanced cloud control system were introduced, allow private cloud vms to be managed directly with Salt. This system is generally referred to as **Salt Virt**.

The Salt Virt system already exists and is installed within Salt itself, this means that besides setting up Salt, no additional salt code needs to be deployed.

Note: The `libvirt` python module and the `certtool` binary are required.

The main goal of Salt Virt is to facilitate a very fast and simple cloud. The cloud that can scale and is fully featured. Salt Virt comes with the ability to set up and manage complex virtual machine networking, powerful image and disk management, as well as virtual machine migration with and without shared storage.

This means that Salt Virt can be used to create a cloud from a blade center and a SAN, but can also create a cloud out of a swarm of Linux Desktops without a single shared storage system. Salt Virt can make clouds from truly commodity hardware, but can also stand up the power of specialized hardware as well.

Setting up Hypervisors

The first step to set up the hypervisors involves getting the correct software installed and setting up the hypervisor network interfaces.

Installing Hypervisor Software

Salt Virt is made to be hypervisor agnostic but currently the only fully implemented hypervisor is KVM via libvirt.

The required software for a hypervisor is libvirt and kvm. For advanced features install libguestfs or qemu-nbd.

Note: Libguestfs and qemu-nbd allow for virtual machine images to be mounted before startup and get pre-seeded with configurations and a salt minion

This sls will set up the needed software for a hypervisor, and run the routines to set up the libvirt pki keys.

Note: Package names and setup used is Red Hat specific, different package names will be required for different platforms

```
libvirt:
  pkg.installed: []
  file.managed:
    - name: /etc/sysconfig/libvirtd
    - contents: 'LIBVIRTD_ARGS="--listen"'
    - require:
      - pkg: libvirt
  virt.keys:
    - require:
      - pkg: libvirt
```

```
  service.running:
    - name: libvirtd
    - require:
      - pkg: libvirt
      - network: br0
      - libvirt: libvirt
    - watch:
      - file: libvirt

libvirt-python:
  pkg.installed: []

libguestfs:
  pkg.installed:
    - pkgs:
      - libguestfs
      - libguestfs-tools
```

Hypervisor Network Setup

The hypervisors will need to be running a network bridge to serve up network devices for virtual machines, this formula will set up a standard bridge on a hypervisor connecting the bridge to eth0:

```
eth0:
  network.managed:
    - enabled: True
    - type: eth
    - bridge: br0

br0:
  network.managed:
    - enabled: True
    - type: bridge
    - proto: dhcp
    - require:
      - network: eth0
```

Virtual Machine Network Setup

Salt Virt comes with a system to model the network interfaces used by the deployed virtual machines; by default a single interface is created for the deployed virtual machine and is bridged to `br0`. To get going with the default networking setup, ensure that the bridge interface named `br0` exists on the hypervisor and is bridged to an active network device.

Note: To use more advanced networking in Salt Virt, read the *Salt Virt Networking* document:

Salt Virt Networking

Libvirt State

One of the challenges of deploying a libvirt based cloud is the distribution of libvirt certificates. These certificates allow for virtual machine migration. Salt comes with a system used to auto deploy these certificates. Salt manages the signing authority key and generates keys for libvirt clients on the master, signs them with the certificate authority and uses pillar to distribute them. This is managed via the `libvirt` state. Simply execute this formula on the minion to ensure that the certificate is in place and up to date:

Note: The above formula includes the calls needed to set up libvirt keys.

```
libvirt_keys:
  virt.keys
```

Getting Virtual Machine Images Ready

Salt Virt, requires that virtual machine images be provided as these are not generated on the fly. Generating these virtual machine images differs greatly based on the underlying platform.

Virtual machine images can be manually created using KVM and running through the installer, but this process is not recommended since it is very manual and prone to errors.

Virtual Machine generation applications are available for many platforms:

kiwi: (openSUSE, SLES, RHEL, CentOS) https://suse.github.io/kiwi/

vm-builder: https://wiki.debian.org/VMBuilder

> **See also**:
>
> vmbuilder-formula

Once virtual machine images are available, the easiest way to make them available to Salt Virt is to place them in the Salt file server. Just copy an image into `/srv/salt` and it can now be used by Salt Virt.

For purposes of this demo, the file name `centos.img` will be used.

Existing Virtual Machine Images

Many existing Linux distributions distribute virtual machine images which can be used with Salt Virt. Please be advised that NONE OF THESE IMAGES ARE SUPPORTED BY SALTSTACK.

CentOS

These images have been prepared for OpenNebula but should work without issue with Salt Virt, only the raw qcow image file is needed: http://wiki.centos.org/Cloud/OpenNebula

Fedora Linux

Images for Fedora Linux can be found here: http://fedoraproject.org/en/get-fedora#clouds

openSUSE

http://download.opensuse.org/repositories/openSUSE:/Leap:/42.1:/Images/images

(look for JeOS-for-kvm-and-xen variant)

SUSE

https://www.suse.com/products/server/jeos

Ubuntu Linux

Images for Ubuntu Linux can be found here: http://cloud-images.ubuntu.com/

Using Salt Virt

With hypervisors set up and virtual machine images ready, Salt can start issuing cloud commands using the *virt runner*.

Start by running a Salt Virt hypervisor info command:

```
salt-run virt.host_info
```

This will query the running hypervisor(s) for stats and display useful information such as the number of cpus and amount of memory.

You can also list all VMs and their current states on all hypervisor nodes:

```
salt-run virt.list
```

Now that hypervisors are available a virtual machine can be provisioned. The `virt.init` routine will create a new virtual machine:

```
salt-run virt.init centos1 2 512 salt://centos.img
```

The Salt Virt runner will now automatically select a hypervisor to deploy the new virtual machine on. Using `salt://` assumes that the CentOS virtual machine image is located in the root of the *Salt File Server* on the master. When images are cloned (i.e. copied locatlly after retrieval from the file server) the destination directory on the hypervisor minion is determined by the `virt.images` config option; by default this is `/srv/salt/salt-images/`.

When a VM is initialized using `virt.init` the image is copied to the hypervisor using `cp.cache_file` and will be mounted and seeded with a minion. Seeding includes setting pre-authenticated keys on the new machine. A minion will only be installed if one can not be found on the image using the default arguments to `seed.apply`.

Note: The biggest bottleneck in starting VMs is when the Salt Minion needs to be installed. Making sure that the source VM images already have Salt installed will GREATLY speed up virtual machine deployment.

You can also deploy an image on a particular minion by directly calling the *virt* execution module with an absolute image path. This can be quite handy for testing:

```
salt 'hypervisor*' virt.init centos1 2 512 image=/var/lib/libvirt/images/centos.img
```

Now that the new VM has been prepared, it can be seen via the `virt.query` command:

```
salt-run virt.query
```

This command will return data about all of the hypervisors and respective virtual machines.

Now that the new VM is booted it should have contacted the Salt Master, a `test.ping` will reveal if the new VM is running.

QEMU copy on write support

For fast image cloning you can use the qcow disk image format. Pass the `enable_qcow` flag and a *.qcow2* image path to *virt.init*:

```
salt 'hypervisor*' virt.init centos1 2 512 image=/var/lib/libvirt/images/centos.qcow2
↪enable_qcow=True start=False
```

Note: Beware that attempting to boot a qcow image too quickly after cloning can result in a race condition where libvirt may try to boot the machine before image seeding has completed. For that reason it is recommended to also pass `start=False` to `virt.init`.

Also know that you **must not** modify the original base image without first making a copy and then *rebasing* all overlay images onto it. See the `qemu-img rebase` usage docs.

Migrating Virtual Machines

Salt Virt comes with full support for virtual machine migration, and using the libvirt state in the above formula makes migration possible.

A few things need to be available to support migration. Many operating systems turn on firewalls when originally set up, the firewall needs to be opened up to allow for libvirt and kvm to cross communicate and execution migration routines. On Red Hat based hypervisors in particular port 16514 needs to be opened on hypervisors:

```
iptables -A INPUT -m state --state NEW -m tcp -p tcp --dport 16514 -j ACCEPT
```

Note: More in-depth information regarding distribution specific firewall settings can read in:

Opening the Firewall up for Salt

Salt also needs the `virt.tunnel` option to be turned on. This flag tells Salt to run migrations securely via the libvirt TLS tunnel and to use port 16514. Without `virt.tunnel` libvirt tries to bind to random ports when running migrations.

To turn on `virt.tunnel` simple apply it to the master config file:

```
virt.tunnel: True
```

Once the master config has been updated, restart the master and send out a call to the minions to refresh the pillar to pick up on the change:

```
salt \* saltutil.refresh_modules
```

Now, migration routines can be run! To migrate a VM, simply run the Salt Virt migrate routine:

```
salt-run virt.migrate centos <new hypervisor>
```

VNC Consoles

Although not enabled by default, Salt Virt can also set up VNC consoles allowing for remote visual consoles to be opened up. When creating a new VM using `virt.init` pass the `enable_vnc=True` parameter to have a console configured for the new VM.

The information from a `virt.query` routine will display the vnc console port for the specific vms:

```
centos
  CPU: 2
  Memory: 524288
  State: running
  Graphics: vnc - hyper6:5900
  Disk - vda:
    Size: 2.0G
    File: /srv/salt-images/ubuntu2/system.qcow2
    File Format: qcow2
  Nic - ac:de:48:98:08:77:
    Source: br0
    Type: bridge
```

The line *Graphics: vnc - hyper6:5900* holds the key. First the port named, in this case 5900, will need to be available in the hypervisor's firewall. Once the port is open, then the console can be easily opened via vncviewer:

```
vncviewer hyper6:5900
```

By default there is no VNC security set up on these ports, which suggests that keeping them firewalled and mandating that SSH tunnels be used to access these VNC interfaces. Keep in mind that activity on a VNC interface that is accessed can be viewed by any other user that accesses that same VNC interface, and any other user logging in can also operate with the logged in user on the virtual machine.

Conclusion

Now with Salt Virt running, new hypervisors can be seamlessly added just by running the above states on new bare metal machines, and these machines will be instantly available to Salt Virt.

4.9.3 Running Salt States and Commands in Docker Containers

The 2016.11.0 release of Salt introduces the ability to execute Salt States and Salt remote execution commands directly inside of Docker containers.

This addition makes it possible to not only deploy fresh containers using Salt States. This also allows for running containers to be audited and modified using Salt, but without running a Salt Minion inside the container. Some of the applications include security audits of running containers as well as gathering operating data from containers.

This new feature is simple and straightforward, and can be used via a running Salt Minion, the Salt Call command, or via Salt SSH. For this tutorial we will use the *salt-call* command, but like all salt commands these calls are directly translatable to *salt* and *salt-ssh*.

Step 1 - Install Docker

Since setting up Docker is well covered in the Docker documentation we will make no such effort to describe it here. Please see the Docker Installation Documentation for installing and setting up Docker: https://docs.docker.com/engine/installation/

The Docker integration also requires that the *docker-py* library is installed. This can easily be done using pip or via your system package manager:

```
pip install docker-py
```

Step 2 - Install Salt

For this tutorial we will be using Salt Call, which is available in the *salt-minion* package, please follow the Salt Installation docs found here: https://repo.saltstack.com/

Step 3 - Create With Salt States

Next some Salt States are needed, for this example a very basic state which installs *vim* is used, but anything Salt States can do can be done here, please see the Salt States Introduction Tutorial to learn more about Salt States: https://docs.saltstack.com/en/stage/getstarted/config/

For this tutorial, simply create a small state file in */srv/salt/vim.sls*:

```
vim:
  pkg.installed
```

Note: The base image you choose will need to have python 2.6 or 2.7 installed. We are hoping to resolve this constraint in a future release.

If *base* is omitted the default image used is a minimal openSUSE image with Python support, maintained by SUSE

Next run the *docker.sls_build* command:

```
salt-call --local dockerng.sls_build test base=my_base_image mods=vim
```

Now we have a fresh image called *test* to work with and vim has been installed.

Step 4 - Running Commands Inside the Container

Salt can now run remote execution functions inside the container with another simple *salt-call* command:

```
salt-call --local dockerng.call test test.ping
salt-call --local dockerng.call test network.interfaces
salt-call --local dockerng.call test disk.usage
salt-call --local dockerng.call test pkg.list_pkgs
salt-call --local dockerng.call test service.running httpd
salt-call --local dockerng.call test cmd.run 'ls -l /etc'
```

4.9.4 Automatic Updates / Frozen Deployments

New in version 0.10.3.d.

Salt has support for the Esky application freezing and update tool. This tool allows one to build a complete zipfile out of the salt scripts and all their dependencies - including shared objects / DLLs.

Getting Started

To build frozen applications, suitable build environment will be needed for each platform. You should probably set up a virtualenv in order to limit the scope of Q/A.

This process does work on Windows. Directions are available at https://github.com/saltstack/salt-windows-install for details on installing Salt in Windows. Only the 32-bit Python and dependencies have been tested, but they have been tested on 64-bit Windows.

Install `bbfreeze`, and then `esky` from PyPI in order to enable the `bdist_esky` command in `setup.py`. Salt itself must also be installed, in addition to its dependencies.

Building and Freezing

Once you have your tools installed and the environment configured, use `setup.py` to prepare the distribution files.

```
python setup.py sdist
python setup.py bdist
```

Once the distribution files are in place, Esky can be used traverse the module tree and pack all the scripts up into a redistributable.

```
python setup.py bdist_esky
```

There will be an appropriately versioned `salt-VERSION.zip` in `dist/` if everything went smoothly.

Windows

`C:\Python27\lib\site-packages\zmq` will need to be added to the PATH variable. This helps bbfreeze find the zmq DLL so it can pack it up.

Using the Frozen Build

Unpack the zip file in the desired install location. Scripts like `salt-minion` and `salt-call` will be in the root of the zip file. The associated libraries and bootstrapping will be in the directories at the same level. (Check the Esky documentation for more information)

To support updating your minions in the wild, put the builds on a web server that the minions can reach. *salt.modules.saltutil.update()* will trigger an update and (optionally) a restart of the minion service under the new version.

Troubleshooting

A Windows minion isn't responding

The process dispatch on Windows is slower than it is on *nix. It may be necessary to add `-t 15' to salt commands to give minions plenty of time to return.

Windows and the Visual Studio Redist

The Visual C++ 2008 32-bit redistributable will need to be installed on all Windows minions. Esky has an option to pack the library into the zipfile, but OpenSSL does not seem to acknowledge the new location. If a `no OPENSSL_Applink` error appears on the console when trying to start a frozen minion, the redistributable is not installed.

Mixed Linux environments and Yum

The Yum Python module doesn't appear to be available on any of the standard Python package mirrors. If RHEL/CentOS systems need to be supported, the frozen build should created on that platform to support all the Linux nodes. Remember to build the virtualenv with `--system-site-packages` so that the `yum` module is included.

Automatic (Python) module discovery

Automatic (Python) module discovery does not work with the late-loaded scheme that Salt uses for (Salt) modules. Any misbehaving modules will need to be explicitly added to the `freezer_includes` in Salt's `setup.py`. Always check the zipped application to make sure that the necessary modules were included.

4.9.5 ESXi Proxy Minion

New in version 2015.8.4.

Note: This tutorial assumes basic knowledge of Salt. To get up to speed, check out the *Salt Walkthrough*.

This tutorial also assumes a basic understanding of Salt Proxy Minions. If you're unfamiliar with Salt's Proxy Minion system, please read the *Salt Proxy Minion* documentation and the *Salt Proxy Minion End-to-End Example* tutorial.

The third assumption that this tutorial makes is that you also have a basic understanding of ESXi hosts. You can learn more about ESXi hosts on VMware's various resources.

Salt's ESXi Proxy Minion allows a VMware ESXi host to be treated as an individual Salt Minion, without installing a Salt Minion on the ESXi host.

Since an ESXi host may not necessarily run on an OS capable of hosting a Python stack, the ESXi host can't run a regular Salt Minion directly. Therefore, Salt's Proxy Minion functionality enables you to designate another machine to host a proxy process that ``proxies'' communication from the Salt Master to the ESXi host. The master does not know or care that the ESXi target is not a ``real'' Salt Minion.

More in-depth conceptual reading on Proxy Minions can be found in the *Proxy Minion* section of Salt's documentation.

Salt's ESXi Proxy Minion was added in the 2015.8.4 release of Salt.

Note: Be aware that some functionality for the ESXi Proxy Minion may depend on the type of license attached the ESXi host(s).

For example, certain services are only available to manipulate service state or policies with a VMware vSphere Enterprise or Enterprise Plus license, while others are available with a Standard license. The `ntpd` service is restricted to an Enterprise Plus license, while `ssh` is available via the Standard license.

Please see the vSphere Comparison page for more information.

Dependencies

Manipulation of the ESXi host via a Proxy Minion requires the machine running the Proxy Minion process to have the ESXCLI package (and all of it's dependencies) and the pyVmomi Python Library to be installed.

ESXi Password

The ESXi Proxy Minion uses VMware's API to perform tasks on the host as if it was a regular Salt Minion. In order to access the API that is already running on the ESXi host, the ESXi host must have a username and password that is used to log into the host. The username is usually `root`. Before Salt can access the ESXi host via VMware's API, a default password *must* be set on the host.

pyVmomi

The pyVmomi Python library must be installed on the machine that is running the proxy process. pyVmomi can be installed via pip:

```
pip install pyVmomi
```

Note: Version 6.0 of pyVmomi has some problems with SSL error handling on certain versions of Python. If using version 6.0 of pyVmomi, the machine that you are running the proxy minion process from must have either Python 2.6, Python 2.7.9, or newer. This is due to an upstream dependency in pyVmomi 6.0 that is not supported in Python version 2.7 to 2.7.8. If the version of Python running the proxy process is not in the supported range, you will need to install an earlier version of pyVmomi. See Issue #29537 for more information.

Based on the note above, to install an earlier version of pyVmomi than the version currently listed in PyPi, run the following:

```
pip install pyVmomi==5.5.0.2014.1.1
```

The 5.5.0.2014.1.1 is a known stable version that the original ESXi Proxy Minion was developed against.

ESXCLI

Currently, about a third of the functions used for the ESXi Proxy Minion require the ESXCLI package be installed on the machine running the Proxy Minion process.

The ESXCLI package is also referred to as the VMware vSphere CLI, or vCLI. VMware provides vCLI package installation instructions for vSphere 5.5 and vSphere 6.0.

Once all of the required dependencies are in place and the vCLI package is installed, you can check to see if you can connect to your ESXi host by running the following command:

```
esxcli -s <host-location> -u <username> -p <password> system syslog config get
```

If the connection was successful, ESXCLI was successfully installed on your system. You should see output related to the ESXi host's syslog configuration.

Configuration

There are several places where various configuration values need to be set in order for the ESXi Proxy Minion to run and connect properly.

Proxy Config File

On the machine that will be running the Proxy Minon process(es), a proxy config file must be in place. This file should be located in the `/etc/salt/` directory and should be named `proxy`. If the file is not there by default, create it.

This file should contain the location of your Salt Master that the Salt Proxy will connect to.

Example Proxy Config File:

```
# /etc/salt/proxy

master: <salt-master-location>
```

Pillar Profiles

Proxy minions get their configuration from Salt's Pillar. Every proxy must have a stanza in Pillar and a reference in the Pillar top-file that matches the Proxy ID. At a minimum for communication with the ESXi host, the pillar should look like this:

```
proxy:
  proxytype: esxi
  host: <ip or dns name of esxi host>
  username: <ESXi username>
  passwords:
    - first_password
    - second_password
    - third_password
```

Some other optional settings are `protocol` and `port`. These can be added to the pillar configuration.

proxytype

The `proxytype` key and value pair is critical, as it tells Salt which interface to load from the `proxy` directory in Salt's install hierarchy, or from `/srv/salt/_proxy` on the Salt Master (if you have created your own proxy module, for example). To use this ESXi Proxy Module, set this to `esxi`.

host

The location, or ip/dns, of the ESXi host. Required.

username

The username used to login to the ESXi host, such as `root`. Required.

passwords

A list of passwords to be used to try and login to the ESXi host. At least one password in this list is required.

The proxy integration will try the passwords listed in order. It is configured this way so you can have a regular password and the password you may be updating for an ESXi host either via the `vsphere.update_host_password` execution module function or via the *`esxi.password_present`* state function. This way, after the password is changed, you should not need to restart the proxy minion--it should just pick up the new password provided in the list. You can then change pillar at will to move that password to the front and retire the unused ones.

Use-case/reasoning for using a list of passwords: You are setting up an ESXi host for the first time, and the host comes with a default password. You know that you'll be changing this password during your initial setup from the default to a new password. If you only have one password option, and if you have a state changing the password, any remote execution commands or states that run after the password change will not be able to run on the host until the password is updated in Pillar and the Proxy Minion process is restarted.

This allows you to use any number of potential fallback passwords.

Note: When a password is changed on the host to one in the list of possible passwords, the further down on the list the password is, the longer individual commands will take to return. This is due to the nature of pyVmomi's login system. We have to wait for the first attempt to fail before trying the next password on the list.

This scenario is especially true, and even slower, when the proxy minion first starts. If the correct password is not the first password on the list, it may take up to a minute for `test.ping` to respond with a `True` result. Once the initial authorization is complete, the responses for commands will be a little faster.

To avoid these longer waiting periods, SaltStack recommends moving the correct password to the top of the list and restarting the proxy minion at your earliest convenience.

---

protocol

If the ESXi host is not using the default protocol, set this value to an alternate protocol. Default is `https`. For example:

port

If the ESXi host is not using the default port, set this value to an alternate port. Default is `443`.

Example Configuration Files

An example of all of the basic configurations that need to be in place before starting the Proxy Minion processes includes the Proxy Config File, Pillar Top File, and any individual Proxy Minion Pillar files.

In this example, we'll assuming there are two ESXi hosts to connect to. Therefore, we'll be creating two Proxy Minion config files, one config for each ESXi host.

Proxy Config File:

```
# /etc/salt/proxy

master: <salt-master-location>
```

Pillar Top File:

```
# /srv/pillar/top.sls

base:
  'esxi-1':
    - esxi-1
  'esxi-2':
    - esxi-2
```

Pillar Config File for the first ESXi host, esxi-1:

```
# /srv/pillar/esxi-1.sls

proxy:
  proxytype: esxi
  host: esxi-1.example.com
  username: 'root'
  passwords:
    - bad-password-1
    - backup-bad-password-1
```

Pillar Config File for the second ESXi host, esxi-2:

```
# /srv/pillar/esxi-2.sls

proxy:
  proxytype: esxi
  host: esxi-2.example.com
  username: 'root'
  passwords:
    - bad-password-2
    - backup-bad-password-2
```

Starting the Proxy Minion

Once all of the correct configuration files are in place, it is time to start the proxy processes!

1. First, make sure your Salt Master is running.
2. Start the first Salt Proxy, in debug mode, by giving the Proxy Minion process and ID that matches the config file name created in the *Configuration* section.

```
salt-proxy --proxyid='esxi-1' -l debug
```

1. Accept the `esxi-1` Proxy Minion's key on the Salt Master:

```
# salt-key -L
Accepted Keys:
```

```
Denied Keys:
Unaccepted Keys:
esxi-1
Rejected Keys:
#
# salt-key -a esxi-1
The following keys are going to be accepted:
Unaccepted Keys:
esxi-1
Proceed? [n/Y] y
Key for minion esxi-1 accepted.
```

1. Repeat for the second Salt Proxy, this time we'll run the proxy process as a daemon, as an example.

```
salt-proxy --proxyid='esxi-2' -d
```

1. Accept the `esxi-2` Proxy Minion's key on the Salt Master:

```
# salt-key -L
Accepted Keys:
esxi-1
Denied Keys:
Unaccepted Keys:
esxi-2
Rejected Keys:
#
# salt-key -a esxi-1
The following keys are going to be accepted:
Unaccepted Keys:
esxi-2
Proceed? [n/Y] y
Key for minion esxi-1 accepted.
```

1. Check and see if your Proxy Minions are responding:

```
# salt 'esxi-*' test.ping
esxi-1:
    True
esxi-3:
    True
```

Executing Commands

Now that you've configured your Proxy Minions and have them responding successfully to a `test.ping`, we can start executing commands against the ESXi hosts via Salt.

It's important to understand how this particular proxy works, and there are a couple of important pieces to be aware of in order to start running remote execution and state commands against the ESXi host via a Proxy Minion: the *vSphere Execution Module*, the *ESXi Execution Module*, and the *ESXi State Module*.

vSphere Execution Module

The *Salt.modules.vsphere* is a standard Salt execution module that does the bulk of the work for the ESXi Proxy Minion. If you pull up the docs for it you'll see that almost every function in the module takes credentials (`username` and `password`) and a target `host` argument. When credentials and a host aren't passed, Salt runs

commands through pyVmomi or ESXCLI against the local machine. If you wanted, you could run functions from this module on any machine where an appropriate version of pyVmomi and ESXCLI are installed, and that machine would reach out over the network and communicate with the ESXi host.

You'll notice that most of the functions in the vSphere module require a host, username, and password. These parameters are contained in the Pillar files and passed through to the function via the proxy process that is already running. You don't need to provide these parameters when you execute the commands. See the *Running Remote Execution Commands* section below for an example.

ESXi Execution Module

In order for the Pillar information set up in the *Configuration* section above to be passed to the function call in the vSphere Execution Module, the *salt.modules.esxi* execution module acts as a ``shim'' between the vSphere execution module functions and the proxy process.

The ``shim'' takes the authentication credentials specified in the Pillar files and passes them through to the host, username, password, and optional protocol and port options required by the vSphere Execution Module functions.

If the function takes more positional, or keyword, arguments you can append them to the call. It's this shim that speaks to the ESXi host through the proxy, arranging for the credentials and hostname to be pulled from the Pillar section for the ESXi Proxy Minion.

Because of the presence of the shim, to lookup documentation for what functions you can use to interface with the ESXi host, you'll want to look in *salt.modules.vsphere* instead of *salt.modules.esxi*.

Running Remote Execution Commands

To run commands from the Salt Master to execute, via the ESXi Proxy Minion, against the ESXi host, you use the esxi.cmd <vsphere-function-name> syntax to call functions located in the vSphere Execution Module. Both args and kwargs needed for various vsphere execution module functions must be passed through in a kwarg-type manor. For example:

```
salt 'esxi-*' esxi.cmd system_info
salt 'exsi-*' esxi.cmd get_service_running service_name='ssh'
```

ESXi State Module

The ESXi State Module functions similarly to other state modules. The ``shim'' provided by the *ESXi Execution Module* passes the necessary host, username, and password credentials through, so those options don't need to be provided in the state. Other than that, state files are written and executed just like any other Salt state. See the *salt.modules.esxi* state for ESXi state functions.

The follow state file is an example of how to configure various pieces of an ESXi host including enabling SSH, uploading and SSH key, configuring a coredump network config, syslog, ntp, enabling VMotion, resetting a host password, and more.

```
# /srv/salt/configure-esxi.sls

configure-host-ssh:
  esxi.ssh_configured:
    - service_running: True
    - ssh_key_file: /etc/salt/ssh_keys/my_key.pub
    - service_policy: 'automatic'
```

```
    - service_restart: True
    - certificate_verify: True

configure-host-coredump:
  esxi.coredump_configured:
    - enabled: True
    - dump_ip: 'my-coredump-ip.example.com'

configure-host-syslog:
  esxi.syslog_configured:
    - syslog_configs:
        loghost: ssl://localhost:5432,tcp://10.1.0.1:1514
        default-timeout: 120
    - firewall: True
    - reset_service: True
    - reset_syslog_config: True
    - reset_configs: loghost,default-timeout

configure-host-ntp:
  esxi.ntp_configured:
    - service_running: True
    - ntp_servers:
      - 192.174.1.100
      - 192.174.1.200
    - service_policy: 'automatic'
    - service_restart: True

configure-vmotion:
  esxi.vmotion_configured:
    - enabled: True

configure-host-vsan:
  esxi.vsan_configured:
    - enabled: True
    - add_disks_to_vsan: True

configure-host-password:
  esxi.password_present:
    - password: 'new-bad-password'
```

States are called via the ESXi Proxy Minion just as they would on a regular minion. For example:

```
salt 'esxi-*' state.sls configure-esxi test=true
salt 'esxi-*' state.sls configure-esxi
```

Relevant Salt Files and Resources

- *ESXi Proxy Minion*
- *ESXi Execution Module*
- *ESXi State Module*
- *Salt Proxy Minion Docs*
- *Salt Proxy Minion End-to-End Example*
- *vSphere Execution Module*

4.9.6 Installing and Configuring Halite

Warning: Halite is deprecated

The Halite project is retired. The code will remain available on GitHub.

In this tutorial, we'll walk through installing and setting up Halite. The current version of Halite is considered pre-alpha and is supported only in Salt `v2014.1.0` or greater. Additional information is available on GitHub: https://github.com/saltstack/halite

Before beginning this tutorial, ensure that the salt-master is installed. To install the salt-master, please review the installation documentation: http://docs.saltstack.com/topics/installation/index.html

Note: Halite only works with Salt versions greater than 2014.1.0.

Installing Halite Via Package

On CentOS, RHEL, or Fedora:

```
$ yum install python-halite
```

Note: By default python-halite only installs CherryPy. If you would like to use a different webserver please review the instructions below to install pip and your server of choice. The package does not modify the master configuration with `/etc/salt/master`.

Installing Halite Using pip

To begin the installation of Halite from PyPI, you'll need to install pip. The Salt package, as well as the bootstrap, do not install pip by default.

On CentOS, RHEL, or Fedora:

```
$ yum install python-pip
```

On Debian:

```
$ apt-get install python-pip
```

Once you have pip installed, use it to install halite:

```
$ pip install -U halite
```

Depending on the webserver you want to run halite through, you'll need to install that piece as well. On RHEL based distros, use one of the following:

```
$ pip install cherrypy
```

```
$ pip install paste
```

```
$ yum install python-devel
$ yum install gcc
$ pip install gevent
$ pip install pyopenssl
```

On Debian based distributions:

```
$ pip install CherryPy
```

```
$ pip install paste
```

```
$ apt-get install gcc
$ apt-get install python-dev
$ apt-get install libevent-dev
$ pip install gevent
$ pip install pyopenssl
```

Configuring Halite Permissions

Configuring Halite access permissions is easy. By default, you only need to ensure that the @runner group is configured. In the `/etc/salt/master` file, uncomment and modify the following lines:

```
external_auth:
  pam:
    testuser:
      - .*
      - '@runner'
```

Note: You cannot use the root user for pam login; it will fail to authenticate.

Halite uses the runner manage.present to get the status of minions, so runner permissions are required. For example:

```
external_auth:
  pam:
    mytestuser:
      - .*
      - '@runner'
      - '@wheel'
```

Currently Halite allows, but does not require, any wheel modules.

Configuring Halite Settings

Once you've configured the permissions for Halite, you'll need to set up the Halite settings in the /etc/salt/master file. Halite supports CherryPy, Paste, and Gevent out of the box.

To configure cherrypy, add the following to the bottom of your /etc/salt/master file:

```
halite:
  level: 'debug'
  server: 'cherrypy'
  host: '0.0.0.0'
  port: '8080'
```

```
  cors: False
  tls: True
  certpath: '/etc/pki/tls/certs/localhost.crt'
  keypath: '/etc/pki/tls/certs/localhost.key'
  pempath: '/etc/pki/tls/certs/localhost.pem'
```

If you wish to use paste:

```
halite:
  level: 'debug'
  server: 'paste'
  host: '0.0.0.0'
  port: '8080'
  cors: False
  tls: True
  certpath: '/etc/pki/tls/certs/localhost.crt'
  keypath: '/etc/pki/tls/certs/localhost.key'
  pempath: '/etc/pki/tls/certs/localhost.pem'
```

To use gevent:

```
halite:
  level: 'debug'
  server: 'gevent'
  host: '0.0.0.0'
  port: '8080'
  cors: False
  tls: True
  certpath: '/etc/pki/tls/certs/localhost.crt'
  keypath: '/etc/pki/tls/certs/localhost.key'
  pempath: '/etc/pki/tls/certs/localhost.pem'
```

The ``cherrypy'' and ``gevent'' servers require the certpath and keypath files to run tls/ssl. The .crt file holds the public cert and the .key file holds the private key. Whereas the ``paste'' server requires a single .pem file that contains both the cert and key. This can be created simply by concatenating the .crt and .key files.

If you want to use a self-signed cert, you can create one using the Salt.tls module:

Note: The following command needs to be run on your salt master.

```
salt-call tls.create_self_signed_cert tls
```

Note that certs generated by the above command can be found under the `/etc/pki/tls/certs/` directory. When using self-signed certs, browsers will need approval before accepting the cert. If the web application page has been cached with a non-HTTPS version of the app, then the browser cache will have to be cleared before it will recognize and prompt to accept the self-signed certificate.

Starting Halite

Once you've configured the halite section of your /etc/salt/master, you can restart the salt-master service, and your halite instance will be available. Depending on your configuration, the instance will be available either at https://localhost:8080/app, https://domain:8080/app, or https://123.456.789.012:8080/app .

Note: halite requires an HTML 5 compliant browser.

All logs relating to halite are logged to the default /var/log/salt/master file.

4.9.7 HTTP Modules

This tutorial demonstrates using the various HTTP modules available in Salt. These modules wrap the Python `tornado`, `urllib2`, and `requests` libraries, extending them in a manner that is more consistent with Salt workflows.

The `salt.utils.http` Library

This library forms the core of the HTTP modules. Since it is designed to be used from the minion as an execution module, in addition to the master as a runner, it was abstracted into this multi-use library. This library can also be imported by 3rd-party programs wishing to take advantage of its extended functionality.

Core functionality of the execution, state, and runner modules is derived from this library, so common usages between them are described here. Documentation specific to each module is described below.

This library can be imported with:

```
import salt.utils.http
```

Configuring Libraries

This library can make use of either `tornado`, which is required by Salt, `urllib2`, which ships with Python, or `requests`, which can be installed separately. By default, `tornado` will be used. In order to switch to `urllib2`, set the following variable:

```
backend: urllib2
```

In order to switch to `requests`, set the following variable:

```
backend: requests
```

This can be set in the master or minion configuration file, or passed as an option directly to any `http.query()` functions.

`salt.utils.http.query()`

This function forms a basic query, but with some add-ons not present in the `tornado`, `urllib2`, and `requests` libraries. Not all functionality currently available in these libraries has been added, but can be in future iterations.

HTTPS Request Methods

A basic query can be performed by calling this function with no more than a single URL:

```
salt.utils.http.query('http://example.com')
```

By default the query will be performed with a GET method. The method can be overridden with the method argument:

```
salt.utils.http.query('http://example.com/delete/url', 'DELETE')
```

When using the POST method (and others, such as PUT), extra data is usually sent as well. This data can be sent directly, in whatever format is required by the remote server (XML, JSON, plain text, etc).

```
salt.utils.http.query(
    'http://example.com/delete/url',
    method='POST',
    data=json.loads(mydict)
)
```

Data Formatting and Templating

Bear in mind that the data must be sent pre-formatted; this function will not format it for you. However, a templated file stored on the local system may be passed through, along with variables to populate it with. To pass through only the file (untemplated):

```
salt.utils.http.query(
    'http://example.com/post/url',
    method='POST',
    data_file='/srv/salt/somefile.xml'
)
```

To pass through a file that contains jinja + yaml templating (the default):

```
salt.utils.http.query(
    'http://example.com/post/url',
    method='POST',
    data_file='/srv/salt/somefile.jinja',
    data_render=True,
    template_dict={'key1': 'value1', 'key2': 'value2'}
)
```

To pass through a file that contains mako templating:

```
salt.utils.http.query(
    'http://example.com/post/url',
    method='POST',
    data_file='/srv/salt/somefile.mako',
    data_render=True,
    data_renderer='mako',
    template_dict={'key1': 'value1', 'key2': 'value2'}
)
```

Because this function uses Salt's own rendering system, any Salt renderer can be used. Because Salt's renderer requires __opts__ to be set, an opts dictionary should be passed in. If it is not, then the default __opts__ values for the node type (master or minion) will be used. Because this library is intended primarily for use by minions, the default node type is minion. However, this can be changed to master if necessary.

```
salt.utils.http.query(
    'http://example.com/post/url',
    method='POST',
    data_file='/srv/salt/somefile.jinja',
```

```
    data_render=True,
    template_dict={'key1': 'value1', 'key2': 'value2'},
    opts=__opts__
)

salt.utils.http.query(
    'http://example.com/post/url',
    method='POST',
    data_file='/srv/salt/somefile.jinja',
    data_render=True,
    template_dict={'key1': 'value1', 'key2': 'value2'},
    node='master'
)
```

Headers

Headers may also be passed through, either as a `header_list`, a `header_dict`, or as a `header_file`. As with the `data_file`, the `header_file` may also be templated. Take note that because HTTP headers are normally syntactically-correct YAML, they will automatically be imported as an a Python dict.

```
salt.utils.http.query(
    'http://example.com/delete/url',
    method='POST',
    header_file='/srv/salt/headers.jinja',
    header_render=True,
    header_renderer='jinja',
    template_dict={'key1': 'value1', 'key2': 'value2'}
)
```

Because much of the data that would be templated between headers and data may be the same, the `template_dict` is the same for both. Correcting possible variable name collisions is up to the user.

Authentication

The `query()` function supports basic HTTP authentication. A username and password may be passed in as `username` and `password`, respectively.

```
salt.utils.http.query(
    'http://example.com',
    username='larry',
    password=`5700g3543v4r`,
)
```

Cookies and Sessions

Cookies are also supported, using Python's built-in `cookielib`. However, they are turned off by default. To turn cookies on, set `cookies` to True.

```
salt.utils.http.query(
    'http://example.com',
    cookies=True
)
```

By default cookies are stored in Salt's cache directory, normally `/var/cache/salt`, as a file called `cookies.txt`. However, this location may be changed with the `cookie_jar` argument:

```
salt.utils.http.query(
    'http://example.com',
    cookies=True,
    cookie_jar='/path/to/cookie_jar.txt'
)
```

By default, the format of the cookie jar is LWP (aka, lib-www-perl). This default was chosen because it is a human-readable text file. If desired, the format of the cookie jar can be set to Mozilla:

```
salt.utils.http.query(
    'http://example.com',
    cookies=True,
    cookie_jar='/path/to/cookie_jar.txt',
    cookie_format='mozilla'
)
```

Because Salt commands are normally one-off commands that are piped together, this library cannot normally behave as a normal browser, with session cookies that persist across multiple HTTP requests. However, the session can be persisted in a separate cookie jar. The default filename for this file, inside Salt's cache directory, is `cookies.session.p`. This can also be changed.

```
salt.utils.http.query(
    'http://example.com',
    persist_session=True,
    session_cookie_jar='/path/to/jar.p'
)
```

The format of this file is msgpack, which is consistent with much of the rest of Salt's internal structure. Historically, the extension for this file is `.p`. There are no current plans to make this configurable.

Proxy

If the `tornado` backend is used (`tornado` is the default), proxy information configured in `proxy_host`, `proxy_port`, `proxy_username`, and `proxy_password` from the `__opts__` dictionary will be used. Normally these are set in the minion configuration file.

```
proxy_host: proxy.my-domain
proxy_port: 31337
proxy_username: charon
proxy_password: obolus
```

```
salt.utils.http.query(
    'http://example.com',
    opts=__opts__,
    backend='tornado'
)
```

Return Data

Note: Return data encoding

If decode is set to True, query() will attempt to decode the return data. decode_type defaults to auto. Set it to a specific encoding, xml, for example, to override autodetection.

Because Salt's http library was designed to be used with REST interfaces, query() will attempt to decode the data received from the remote server when decode is set to True. First it will check the Content-type header to try and find references to XML. If it does not find any, it will look for references to JSON. If it does not find any, it will fall back to plain text, which will not be decoded.

JSON data is translated into a dict using Python's built-in json library. XML is translated using salt.utils.xml_util, which will use Python's built-in XML libraries to attempt to convert the XML into a dict. In order to force either JSON or XML decoding, the decode_type may be set:

```
salt.utils.http.query(
    'http://example.com',
    decode_type='xml'
)
```

Once translated, the return dict from query() will include a dict called dict.

If the data is not to be translated using one of these methods, decoding may be turned off.

```
salt.utils.http.query(
    'http://example.com',
    decode=False
)
```

If decoding is turned on, and references to JSON or XML cannot be found, then this module will default to plain text, and return the undecoded data as text (even if text is set to False; see below).

The query() function can return the HTTP status code, headers, and/or text as required. However, each must individually be turned on.

```
salt.utils.http.query(
    'http://example.com',
    status=True,
    headers=True,
    text=True
)
```

The return from these will be found in the return dict as status, headers and text, respectively.

Writing Return Data to Files

It is possible to write either the return data or headers to files, as soon as the response is received from the server, but specifying file locations via the text_out or headers_out arguments. text and headers do not need to be returned to the user in order to do this.

```
salt.utils.http.query(
    'http://example.com',
    text=False,
    headers=False,
    text_out='/path/to/url_download.txt',
    headers_out='/path/to/headers_download.txt',
)
```

SSL Verification

By default, this function will verify SSL certificates. However, for testing or debugging purposes, SSL verification can be turned off.

```
salt.utils.http.query(
    'https://example.com',
    verify_ssl=False,
)
```

CA Bundles

The `requests` library has its own method of detecting which CA (certificate authority) bundle file to use. Usually this is implemented by the packager for the specific operating system distribution that you are using. However, `urllib2` requires a little more work under the hood. By default, Salt will try to auto-detect the location of this file. However, if it is not in an expected location, or a different path needs to be specified, it may be done so using the `ca_bundle` variable.

```
salt.utils.http.query(
    'https://example.com',
    ca_bundle='/path/to/ca_bundle.pem',
)
```

Updating CA Bundles

The `update_ca_bundle()` function can be used to update the bundle file at a specified location. If the target location is not specified, then it will attempt to auto-detect the location of the bundle file. If the URL to download the bundle from does not exist, a bundle will be downloaded from the cURL website.

CAUTION: The `target` and the `source` should always be specified! Failure to specify the `target` may result in the file being written to the wrong location on the local system. Failure to specify the `source` may cause the upstream URL to receive excess unnecessary traffic, and may cause a file to be download which is hazardous or does not meet the needs of the user.

```
salt.utils.http.update_ca_bundle(
    target='/path/to/ca-bundle.crt',
    source='https://example.com/path/to/ca-bundle.crt',
    opts=__opts__,
)
```

The `opts` parameter should also always be specified. If it is, then the `target` and the `source` may be specified in the relevant configuration file (master or minion) as `ca_bundle` and `ca_bundle_url`, respectively.

```
ca_bundle: /path/to/ca-bundle.crt
ca_bundle_url: https://example.com/path/to/ca-bundle.crt
```

If Salt is unable to auto-detect the location of the CA bundle, it will raise an error.

The `update_ca_bundle()` function can also be passed a string or a list of strings which represent files on the local system, which should be appended (in the specified order) to the end of the CA bundle file. This is useful in environments where private certs need to be made available, and are not otherwise reasonable to add to the bundle file.

```
salt.utils.http.update_ca_bundle(
    opts=__opts__,
    merge_files=[
        '/etc/ssl/private_cert_1.pem',
        '/etc/ssl/private_cert_2.pem',
        '/etc/ssl/private_cert_3.pem',
    ]
)
```

Test Mode

This function may be run in test mode. This mode will perform all work up until the actual HTTP request. By default, instead of performing the request, an empty dict will be returned. Using this function with `TRACE` logging turned on will reveal the contents of the headers and POST data to be sent.

Rather than returning an empty dict, an alternate `test_url` may be passed in. If this is detected, then test mode will replace the `url` with the `test_url`, set `test` to `True` in the return data, and perform the rest of the requested operations as usual. This allows a custom, non-destructive URL to be used for testing when necessary.

Execution Module

The `http` execution module is a very thin wrapper around the `salt.utils.http` library. The `opts` can be passed through as well, but if they are not specified, the minion defaults will be used as necessary.

Because passing complete data structures from the command line can be tricky at best and dangerous (in terms of execution injection attacks) at worse, the `data_file`, and `header_file` are likely to see more use here.

All methods for the library are available in the execution module, as kwargs.

```
salt myminion http.query http://example.com/restapi method=POST \
    username='larry' password='5700g3543v4r' headers=True text=True \
    status=True decode_type=xml data_render=True \
    header_file=/tmp/headers.txt data_file=/tmp/data.txt \
    header_render=True cookies=True persist_session=True
```

Runner Module

Like the execution module, the `http` runner module is a very thin wrapper around the `salt.utils.http` library. The only significant difference is that because runners execute on the master instead of a minion, a target is not required, and default opts will be derived from the master config, rather than the minion config.

All methods for the library are available in the runner module, as kwargs.

```
salt-run http.query http://example.com/restapi method=POST \
    username='larry' password='5700g3543v4r' headers=True text=True \
    status=True decode_type=xml data_render=True \
    header_file=/tmp/headers.txt data_file=/tmp/data.txt \
    header_render=True cookies=True persist_session=True
```

State Module

The state module is a wrapper around the runner module, which applies stateful logic to a query. All kwargs as listed above are specified as usual in state files, but two more kwargs are available to apply stateful logic. A required

parameter is match, which specifies a pattern to look for in the return text. By default, this will perform a string comparison of looking for the value of match in the return text. In Python terms this looks like:

```
if match in html_text:
    return True
```

If more complex pattern matching is required, a regular expression can be used by specifying a match_type. By default this is set to string, but it can be manually set to pcre instead. Please note that despite the name, this will use Python's re.search() rather than re.match().

Therefore, the following states are valid:

```
http://example.com/restapi:
  http.query:
    - match: 'SUCCESS'
    - username: 'larry'
    - password: '5700g3543v4r'
    - data_render: True
    - header_file: /tmp/headers.txt
    - data_file: /tmp/data.txt
    - header_render: True
    - cookies: True
    - persist_session: True

http://example.com/restapi:
  http.query:
    - match_type: pcre
    - match: '(?i)succe[ss|ed]'
    - username: 'larry'
    - password: '5700g3543v4r'
    - data_render: True
    - header_file: /tmp/headers.txt
    - data_file: /tmp/data.txt
    - header_render: True
    - cookies: True
    - persist_session: True
```

In addition to, or instead of a match pattern, the status code for a URL can be checked. This is done using the status argument:

```
http://example.com/:
  http.query:
    - status: '200'
```

If both are specified, both will be checked, but if only one is True and the other is False, then False will be returned. In this case, the comments in the return data will contain information for troubleshooting.

Because this is a monitoring state, it will return extra data to code that expects it. This data will always include text and status. Optionally, headers and dict may also be requested by setting the headers and decode arguments to True, respectively.

4.9.8 Using Salt at scale

The focus of this tutorial will be building a Salt infrastructure for handling large numbers of minions. This will include tuning, topology, and best practices.

For how to install the Salt Master please go here: Installing saltstack

Note: This tutorial is intended for large installations, although these same settings won't hurt, it may not be worth the complexity to smaller installations.

When used with minions, the term `many' refers to at least a thousand and `a few' always means 500.

For simplicity reasons, this tutorial will default to the standard ports used by Salt.

The Master

The most common problems on the Salt Master are:

1. too many minions authing at once
2. too many minions re-authing at once
3. too many minions re-connecting at once
4. too many minions returning at once
5. too few resources (CPU/HDD)

The first three are all ``thundering herd'' problems. To mitigate these issues we must configure the minions to back-off appropriately when the Master is under heavy load.

The fourth is caused by masters with little hardware resources in combination with a possible bug in ZeroMQ. At least that's what it looks like till today (Issue 118651, Issue 5948, Mail thread)

To fully understand each problem, it is important to understand, how Salt works.

Very briefly, the Salt Master offers two services to the minions.

- a job publisher on port 4505
- an open port 4506 to receive the minions returns

All minions are always connected to the publisher on port 4505 and only connect to the open return port 4506 if necessary. On an idle Master, there will only be connections on port 4505.

Too many minions authing

When the Minion service is first started up, it will connect to its Master's publisher on port 4505. If too many minions are started at once, this can cause a ``thundering herd''. This can be avoided by not starting too many minions at once.

The connection itself usually isn't the culprit, the more likely cause of master-side issues is the authentication that the Minion must do with the Master. If the Master is too heavily loaded to handle the auth request it will time it out. The Minion will then wait *acceptance_wait_time* to retry. If *acceptance_wait_time_max* is set then the Minion will increase its wait time by the *acceptance_wait_time* each subsequent retry until reaching *acceptance_wait_time_max*.

Too many minions re-authing

This is most likely to happen in the testing phase of a Salt deployment, when all Minion keys have already been accepted, but the framework is being tested and parameters are frequently changed in the Salt Master's configuration file(s).

The Salt Master generates a new AES key to encrypt its publications at certain events such as a Master restart or the removal of a Minion key. If you are encountering this problem of too many minions re-authing against the Master,

you will need to recalibrate your setup to reduce the rate of events like a Master restart or Minion key removal (`salt-key -d`).

When the Master generates a new AES key, the minions aren't notified of this but will discover it on the next pub job they receive. When the Minion receives such a job it will then re-auth with the Master. Since Salt does minion-side filtering this means that all the minions will re-auth on the next command published on the master-- causing another ``thundering herd''. This can be avoided by setting the

```
random_reauth_delay: 60
```

in the minions configuration file to a higher value and stagger the amount of re-auth attempts. Increasing this value will of course increase the time it takes until all minions are reachable via Salt commands.

Too many minions re-connecting

By default the zmq socket will re-connect every 100ms which for some larger installations may be too quick. This will control how quickly the TCP session is re-established, but has no bearing on the auth load.

To tune the minions sockets reconnect attempts, there are a few values in the sample configuration file (default values)

```
recon_default: 1000
recon_max: 5000
recon_randomize: True
```

- recon_default: the default value the socket should use, i.e. 1000. This value is in milliseconds. (1000ms = 1 second)
- recon_max: the max value that the socket should use as a delay before trying to reconnect This value is in milliseconds. (5000ms = 5 seconds)
- recon_randomize: enables randomization between recon_default and recon_max

To tune this values to an existing environment, a few decision have to be made.

1. How long can one wait, before the minions should be online and reachable via Salt?
2. How many reconnects can the Master handle without a syn flood?

These questions can not be answered generally. Their answers depend on the hardware and the administrators requirements.

Here is an example scenario with the goal, to have all minions reconnect within a 60 second time-frame on a Salt Master service restart.

```
recon_default: 1000
recon_max: 59000
recon_randomize: True
```

Each Minion will have a randomized reconnect value between `recon_default' and `recon_default + recon_max', which in this example means between 1000ms and 60000ms (or between 1 and 60 seconds). The generated random-value will be doubled after each attempt to reconnect (ZeroMQ default behavior).

Lets say the generated random value is 11 seconds (or 11000ms).

```
reconnect 1: wait 11 seconds
reconnect 2: wait 22 seconds
reconnect 3: wait 33 seconds
reconnect 4: wait 44 seconds
```

```
reconnect 5: wait 55 seconds
reconnect 6: wait time is bigger than 60 seconds (recon_default + recon_max)
reconnect 7: wait 11 seconds
reconnect 8: wait 22 seconds
reconnect 9: wait 33 seconds
reconnect x: etc.
```

With a thousand minions this will mean

```
1000/60 = ~16
```

round about 16 connection attempts a second. These values should be altered to values that match your environment. Keep in mind though, that it may grow over time and that more minions might raise the problem again.

Too many minions returning at once

This can also happen during the testing phase, if all minions are addressed at once with

```
$ salt * disk.usage
```

it may cause thousands of minions trying to return their data to the Salt Master open port 4506. Also causing a flood of syn-flood if the Master can't handle that many returns at once.

This can be easily avoided with Salt's batch mode:

```
$ salt * disk.usage -b 50
```

This will only address 50 minions at once while looping through all addressed minions.

Too few resources

The masters resources always have to match the environment. There is no way to give good advise without knowing the environment the Master is supposed to run in. But here are some general tuning tips for different situations:

The Master is CPU bound

Salt uses RSA-Key-Pairs on the masters and minions end. Both generate 4096 bit key-pairs on first start. While the key-size for the Master is currently not configurable, the minions keysize can be configured with different key-sizes. For example with a 2048 bit key:

```
keysize: 2048
```

With thousands of decryptions, the amount of time that can be saved on the masters end should not be neglected. See here for reference: Pull Request 9235 how much influence the key-size can have.

Downsizing the Salt Master's key is not that important, because the minions do not encrypt as many messages as the Master does.

In installations with large or with complex pillar files, it is possible for the master to exhibit poor performance as a result of having to render many pillar files at once. This exhibit itself in a number of ways, both as high load on the master and on minions which block on waiting for their pillar to be delivered to them.

To reduce pillar rendering times, it is possible to cache pillars on the master. To do this, see the set of master configuration options which are prefixed with *pillar_cache*.

Note: Caching pillars on the master may introduce security considerations. Be certain to read caveats outlined in the master configuration file to understand how pillar caching may affect a master's ability to protect sensitive data!

The Master is disk IO bound

By default, the Master saves every Minion's return for every job in its job-cache. The cache can then be used later, to lookup results for previous jobs. The default directory for this is:

```
cachedir: /var/cache/salt
```

and then in the `/proc` directory.

Each job return for every Minion is saved in a single file. Over time this directory can grow quite large, depending on the number of published jobs. The amount of files and directories will scale with the number of jobs published and the retention time defined by

```
keep_jobs: 24
```

```
250 jobs/day * 2000 minions returns = 500,000 files a day
```

Use and External Job Cache

An external job cache allows for job storage to be placed on an external system, such as a database.

- ext_job_cache: this will have the minions store their return data directly into a returner (not sent through the Master)
- master_job_cache (New in *2014.7.0*): this will make the Master store the job data using a returner (instead of the local job cache on disk).

If a master has many accepted keys, it may take a long time to publish a job because the master much first determine the matching minions and deliver that information back to the waiting client before the job can be published.

To mitigate this, a key cache may be enabled. This will reduce the load on the master to a single file open instead of thousands or tens of thousands.

This cache is updated by the maintanence process, however, which means that minions with keys that are accepted may not be targeted by the master for up to sixty seconds by default.

To enable the master key cache, set *key_cache: `sched'* in the master configuration file.

Disable The Job Cache

The job cache is a central component of the Salt Master and many aspects of the Salt Master will not function correctly without a running job cache.

Disabling the job cache is **STRONGLY DISCOURAGED** and should not be done unless the master is being used to execute routines that require no history or reliable feedback!

The job cache can be disabled:

```
job_cache: False
```

4.9.9 How to Convert Jinja Logic to an Execution Module

The Problem: Jinja Gone Wild

It is often said in the Salt community that ``Jinja is not a Programming Language''. There's an even older saying known as Maslow's hammer. It goes something like ``if all you have is a hammer, everything looks like a nail''. Jinja is a reliable hammer, and so is the *maps.jinja* idiom. Unfortunately, it can lead to code that looks like the following.

```
# storage/maps.yaml

{% import_yaml 'storage/defaults.yaml' as default_settings %}
{% set storage = default_settings.storage %}
{% do storage.update(salt['grains.filter_by']({
    'Debian': {
    },
    'RedHat': {
    }
}, merge=salt['pillar.get']('storage:lookup'))) %}

{% if 'VirtualBox' == grains.get('virtual', None) or 'oracle' == grains.get('virtual',
↪None) %}
{%   do storage.update({'depot_ip': '192.168.33.81', 'server_ip':  '192.168.33.51'}) %}
{% else %}
{%   set colo = pillar.get('inventory', {}).get('colo', 'Unknown') %}
{%   set servers_list = pillar.get('storage_servers', {}).get(colo, [storage.depot_ip,
↪]) %}
{%   if opts.id.startswith('foo') %}
{%     set modulus = servers_list | count %}
{%     set integer_id = opts.id | replace('foo', '') | int %}
{%     set server_index = integer_id % modulus %}
{%   else %}
{%     set server_index = 0 %}
{%   endif %}
{%   do storage.update({'server_ip': servers_list[server_index]}) %}
{% endif %}

{% for network, _ in salt.pillar.get('inventory:networks', {}) | dictsort %}
{%   do storage.ipsets.hash_net.foo_networks.append(network) %}
{% endfor %}
```

This is an example from the author's salt formulae demonstrating misuse of jinja. Aside from being difficult to read and maintain, accessing the logic it contains from a non-jinja renderer while probably possible is a significant barrier!

Refactor

The first step is to reduce the maps.jinja file to something reasonable. This gives us an idea of what the module we are writing needs to do. There is a lot of logic around selecting a storage server ip. Let's move that to an execution module.

```
# storage/maps.yaml

{% import_yaml 'storage/defaults.yaml' as default_settings %}
{% set storage = default_settings.storage %}
{% do storage.update(salt['grains.filter_by']({
    'Debian': {
    },
```

```
    'RedHat': {
    }
}, merge=salt['pillar.get']('storage:lookup'))) %}

{% if 'VirtualBox' == grains.get('virtual', None) or 'oracle' == grains.get('virtual',
↪None) %}
{%   do storage.update({'depot_ip': '192.168.33.81'}) %}
{% endif %}

{% do storage.update({'server_ip': salt['storage.ip']()}) %}

{% for network, _ in salt.pillar.get('inventory:networks', {}) | dictsort %}
{%   do storage.ipsets.hash_net.af_networks.append(network) %}
{% endfor %}
```

And then, write the module. Note how the module encapsulates all of the logic around finding the storage server IP.

```
# _modules/storage.py
#!python

'''
Functions related to storage servers.
'''

import re


def ips():
    '''
    Provide a list of all local storage server IPs.

    CLI Example::

        salt \* storage.ips
    '''

    if  __grains__.get('virtual', None) in ['VirtualBox', 'oracle']:
        return ['192.168.33.51', ]

    colo = __pillar__.get('inventory', {}).get('colo', 'Unknown')
    return __pillar__.get('storage_servers', {}).get(colo, ['unknown', ])


def ip():
    '''
    Select and return a local storage server IP.

    This loadbalances across storage servers by using the modulus of the client's id
↪number.

    :maintainer:    Andrew Hammond <ahammond@anchorfree.com>
    :maturity:      new
    :depends:       None
    :platform:      all

    CLI Example::

        salt \* storage.ip
```

```
    '''

    numerical_suffix = re.compile(r'^.*(\d+)$')
    servers_list = ips()

    m = numerical_suffix.match(__grains__['id'])
    if m:
        modulus = len(servers_list)
        server_number = int(m.group(1))
        server_index = server_number % modulus
    else:
        server_index = 0

    return servers_list[server_index]
```

Conclusion

That was… surprisingly straight-forward. Now the logic is available in every renderer, instead of just Jinja. Best of all, it can be maintained in Python, which is a whole lot easier than Jinja.

4.9.10 Using Apache Libcloud for declarative and procedural multi-cloud orchestration

New in version 2018.3.0.

Note: This walkthrough assumes basic knowledge of Salt and Salt States. To get up to speed, check out the *Salt Walkthrough*.

Apache Libcloud is a Python library which hides differences between different cloud provider APIs and allows you to manage different cloud resources through a unified and easy to use API. Apache Libcloud supports over 60 cloud platforms, including Amazon, Microsoft Azure, DigitalOcean, Google Cloud Platform and OpenStack.

Execution and state modules are available for Compute, DNS, Storage and Load Balancer drivers from Apache Libcloud in
SaltStack.

- ***libcloud_compute*** - **Compute** - services such as OpenStack Nova, Amazon EC2, Microsoft Azure VMs
- ***libcloud_dns*** - **DNS as a Service** - services such as Amazon Route 53 and Zerigo
- ***libcloud_loadbalancer*** - **Load Balancers as a Service** - services such as Amazon Elastic Load Balancer and GoGrid LoadBalancers
- ***libcloud_storage*** - **Cloud Object Storage and CDN** - services such as Amazon S3 and Rackspace CloudFiles, OpenStack Swift

These modules are designed as a way of having a multi-cloud deployment and abstracting simple differences between platform to design a high-availability architecture.

The Apache Libcloud functionality is available through both execution modules and Salt states.

Configuring Drivers

Drivers can be configured in the Salt Configuration/Minion settings. All libcloud modules expect a list of ``profiles'' to be configured with authentication details for each driver.

Each driver will have a string identifier, these can be found in the libcloud.<api>.types.Provider class for each API, http://libcloud.readthedocs.io/en/latest/supported_providers.html

Some drivers require additional parameters, which are documented in the Apache Libcloud documentation. For example, GoDaddy DNS expects ``*shopper_id*'', which is the customer ID. These additional parameters can be added to the profile settings and will be passed directly to the driver instantiation method.

```
libcloud_dns:
    godaddy:
        driver: godaddy
        shopper_id: 90425123
        key: AFDDJFGIjDFVNSDIFNASMC
        secret: FG(#f8vdfgjlkm)

libcloud_storage:
    google:
        driver: google_storage
        key: GOOG4ASDIDFNVIdfnIVW
        secret: R+qYE9hkfdhv89h4invhdfvird4Pq3an8rnK
```

You can have multiple profiles for a single driver, for example if you wanted 2 DNS profiles for Amazon Route53, naming them ``route53_prod'' and ``route54_test'' would help your administrators distinguish their purpose.

```
libcloud_dns:
    route53_prod:
        driver: route53
        key: AFDDJFGIjDFVNSDIFNASMC
        secret: FG(#f8vdfgjlkm)
    route53_test:
        driver: route53
        key: AFDDJFGIjdfgdfgdf
        secret: FG(#f8vdfgjlkm)
```

Using the execution modules

Amongst over 60 clouds that Apache Libcloud supports, you can add profiles to your Salt configuration to access and control these clouds. Each of the libcloud execution modules exposes the common API methods for controlling Compute, DNS, Load Balancers and Object Storage. To see which functions are supported across specific clouds, see the Libcloud supported methods documentation.

The module documentation explains each of the API methods and how to leverage them.

- ***libcloud_compute*** - **Compute** - services such as OpenStack Nova, Amazon EC2, Microsoft Azure VMs
- ***libcloud_dns*** - **DNS as a Service** - services such as Amazon Route 53 and Zerigo
- ***libcloud_loadbalancer*** - **Load Balancers as a Service** - services such as Amazon Elastic Load Balancer and GoGrid LoadBalancers
- ***libcloud_storage*** - **Cloud Object Storage and CDN** - services such as Amazon S3 and Rackspace CloudFiles, OpenStack Swift

For example, listing buckets in the Google Storage platform:

```
$ salt-call libcloud_storage.list_containers google

    local:
        |_
        ----------
```

```
        extra:
            ----------
            creation_date:
                2017-01-05T05:44:56.324Z
        name:
            anthonypjshaw
```

The Apache Libcloud storage module can be used to syncronize files between multiple storage clouds, such as Google Storage, S3 and OpenStack Swift

```
$ salt '*' libcloud_storage.download_object DeploymentTools test.sh /tmp/test.sh␣
↪google_storage
```

Using the state modules

For each configured profile, the assets available in the API (e.g. storage objects, containers, DNS records and load balancers) can be deployed via Salt's state system.

The state module documentation explains the specific states that each module supports

- ***libcloud_storage*** **- Cloud Object Storage and CDN**
  - services such as Amazon S3 and Rackspace CloudFiles, OpenStack Swift
- ***libcloud_loadbalancer*** **- Load Balancers as a Service**
  - services such as Amazon Elastic Load Balancer and GoGrid LoadBalancers
- ***libcloud_dns*** **- DNS as a Service**
  - services such as Amazon Route 53 and Zerigo

For DNS, the state modules can be used to provide DNS resilience for multiple nameservers, for example:

```
libcloud_dns:
    godaddy:
        driver: godaddy
        shopper_id: 12345
        key: 2orgk34kgk34g
        secret: fjgoidhjgoim
    amazon:
        driver: route53
        key: blah
        secret: blah
```

And then in a state file:

```
webserver:
  libcloud_dns.zone_present:
    name: mywebsite.com
    profile: godaddy
  libcloud_dns.record_present:
    name: www
    zone: mywebsite.com
    type: A
    data: 12.34.32.3
    profile: godaddy
  libcloud_dns.zone_present:
    name: mywebsite.com
```

```
    profile: amazon
  libcloud_dns.record_present:
    name: www
    zone: mywebsite.com
    type: A
    data: 12.34.32.3
    profile: amazon
```

This could be combined with a multi-cloud load balancer deployment,

```
webserver:
  libcloud_dns.zone_present:
    - name: mywebsite.com
    - profile: godaddy
    ...
  libcloud_loadbalancer.balancer_present:
    - name: web_main
    - port: 80
    - protocol: http
    - members:
        - ip: 1.2.4.5
          port: 80
        - ip: 2.4.5.6
          port: 80
    - profile: google_gce
  libcloud_loadbalancer.balancer_present:
    - name: web_main
    - port: 80
    - protocol: http
    - members:
        - ip: 1.2.4.5
          port: 80
        - ip: 2.4.5.6
          port: 80
    - profile: amazon_elb
```

Extended parameters can be passed to the specific cloud, for example you can specify the region with the Google Cloud API, because *create_balancer* can accept a *ex_region* argument. Adding this argument to the state will pass the additional command to the driver.

```
lb_test:
    libcloud_loadbalancer.balancer_absent:
        - name: example
        - port: 80
        - protocol: http
        - profile: google
        - ex_region: us-east1
```

Accessing custom arguments in execution modules

Some cloud providers have additional functionality that can be accessed on top of the base API, for example the Google Cloud Engine load balancer service offers the ability to provision load balancers into a specific region.

Looking at the API documentation, we can see that it expects an *ex_region* in the *create_balancer* method, so when we execute the salt command, we can add this additional parameter like this:

```
$ salt myminion libcloud_storage.create_balancer my_balancer 80 http profile1 ex_
↪region=us-east1
$ salt myminion libcloud_storage.list_container_objects my_bucket profile1 ex_prefix=me
```

Accessing custom methods in Libcloud drivers

Some cloud APIs have additional methods that are prefixed with *ex_* in Apache Libcloud, these methods are part of the non-standard API but can still be accessed from the Salt modules for *libcloud_storage*, *libcloud_loadbalancer* and *libcloud_dns*. The extra methods are available via the *extra* command, which expects the name of the method as the first argument, the profile as the second and then accepts a list of keyword arguments to pass onto the driver method, for example, accessing permissions in Google Storage objects:

```
$ salt myminion libcloud_storage.extra ex_get_permissions google container_name=my_
↪container object_name=me.jpg --out=yaml
```

Example profiles

Google Cloud

Using Service Accounts with GCE, you can provide a path to the JSON file and the project name in the parameters.

```
google:
    driver: gce
    user_id: 234234-compute@developer.gserviceaccount.com
    key: /path/to/service_account_download.json
    auth_type: SA
    project: project-name
```

4.9.11 LXC Management with Salt

Note: This walkthrough assumes basic knowledge of Salt. To get up to speed, check out the *Salt Walkthrough*.

Dependencies

Manipulation of LXC containers in Salt requires the minion to have an LXC version of at least 1.0 (an alpha or beta release of LXC 1.0 is acceptable). The following distributions are known to have new enough versions of LXC packaged:

- RHEL/CentOS 6 and later (via EPEL)
- Fedora (All non-EOL releases)
- Debian 8.0 (Jessie)
- Ubuntu 14.04 LTS and later (LXC templates are packaged separately as **lxc-templates**, it is recommended to also install this package)
- openSUSE 13.2 and later

Profiles

Profiles allow for a sort of shorthand for commonly-used configurations to be defined in the minion config file, *grains*, *pillar*, or the master config file. The profile is retrieved by Salt using the `config.get` function, which looks in those locations, in that order. This allows for profiles to be defined centrally in the master config file, with several options for overriding them (if necessary) on groups of minions or individual minions.

There are two types of profiles:

- One for defining the parameters used in container creation/clone.
- One for defining the container's network interface(s) settings.

Container Profiles

LXC container profiles are defined underneath the `lxc.container_profile` config option:

```
lxc.container_profile:
  centos:
    template: centos
    backing: lvm
    vgname: vg1
    lvname: lxclv
    size: 10G
  centos_big:
    template: centos
    backing: lvm
    vgname: vg1
    lvname: lxclv
    size: 20G
```

Profiles are retrieved using the `config.get` function, with the **recurse** merge strategy. This means that a profile can be defined at a lower level (for example, the master config file) and then parts of it can be overridden at a higher level (for example, in pillar data). Consider the following container profile data:

In the Master config file:

```
lxc.container_profile:
  centos:
    template: centos
    backing: lvm
    vgname: vg1
    lvname: lxclv
    size: 10G
```

In the Pillar data

```
lxc.container_profile:
  centos:
    size: 20G
```

Any minion with the above Pillar data would have the **size** parameter in the **centos** profile overridden to 20G, while those minions without the above Pillar data would have the 10G **size** value. This is another way of achieving the same result as the **centos_big** profile above, without having to define another whole profile that differs in just one value.

Note: In the 2014.7.x release cycle and earlier, container profiles are defined under `lxc.profile`. This parameter will still work in version 2015.5.0, but is deprecated and will be removed in a future release. Please note however that the profile merging feature described above will only work with profiles defined under `lxc.container_profile`, and only in versions 2015.5.0 and later.

Additionally, in version 2015.5.0 container profiles have been expanded to support passing template-specific CLI options to *lxc.create*. Below is a table describing the parameters which can be configured in container profiles:

Parameter	2015.5.0 and Newer	2014.7.x and Earlier
template[1]	Yes	Yes
options[1]	Yes	No
image[1]	Yes	Yes
backing	Yes	Yes
snapshot[2]	Yes	Yes
lvname[1]	Yes	Yes
fstype[1]	Yes	Yes
size	Yes	Yes

1. Parameter is only supported for container creation, and will be ignored if the profile is used when cloning a container.
2. Parameter is only supported for container cloning, and will be ignored if the profile is used when not cloning a container.

Network Profiles

LXC network profiles are defined defined underneath the `lxc.network_profile` config option. By default, the module uses a DHCP based configuration and try to guess a bridge to get connectivity.

Warning: on pre **2015.5.2**, you need to specify explicitly the network bridge

```
lxc.network_profile:
  centos:
    eth0:
      link: br0
      type: veth
      flags: up
  ubuntu:
    eth0:
      link: lxcbr0
      type: veth
      flags: up
```

As with container profiles, network profiles are retrieved using the *config.get* function, with the **recurse** merge strategy. Consider the following network profile data:

In the Master config file:

```
lxc.network_profile:
  centos:
    eth0:
      link: br0
```

```
      type: veth
      flags: up
```

In the Pillar data

```
lxc.network_profile:
  centos:
    eth0:
      link: lxcbr0
```

Any minion with the above Pillar data would use the **lxcbr0** interface as the bridge interface for any container configured using the **centos** network profile, while those minions without the above Pillar data would use the **br0** interface for the same.

Note: In the 2014.7.x release cycle and earlier, network profiles are defined under `lxc.nic`. This parameter will still work in version 2015.5.0, but is deprecated and will be removed in a future release. Please note however that the profile merging feature described above will only work with profiles defined under `lxc.network_profile`, and only in versions 2015.5.0 and later.

The following are parameters which can be configured in network profiles. These will directly correspond to a parameter in an LXC configuration file (see `man 5 lxc.container.conf`).

- **type** - Corresponds to **lxc.network.type**
- **link** - Corresponds to **lxc.network.link**
- **flags** - Corresponds to **lxc.network.flags**

Interface-specific options (MAC address, IPv4/IPv6, etc.) must be passed on a container-by-container basis, for instance using the `nic_opts` argument to *lxc.create*:

```
salt myminion lxc.create container1 profile=centos network_profile=centos nic_opts='
↪{eth0: {ipv4: 10.0.0.20/24, gateway: 10.0.0.1}}'
```

Warning: The `ipv4`, `ipv6`, `gateway`, and `link` (bridge) settings in network profiles / nic_opts will only work if the container doesn't redefine the network configuration (for example in `/etc/sysconfig/network-scripts/ifcfg-<interface_name>` on RHEL/CentOS, or `/etc/network/interfaces` on Debian/Ubuntu/etc.). Use these with caution. The container images installed using the `download` template, for instance, typically are configured for eth0 to use DHCP, which will conflict with static IP addresses set at the container level.

Note: For LXC < 1.0.7 and DHCP support, set `ipv4.gateway: 'auto'` is your network profile, ie.:

```
lxc.network_profile.nic:
  debian:
    eth0:
      link: lxcbr0
      ipv4.gateway: 'auto'
```

Old lxc support (<1.0.7)

With saltstack **2015.5.2** and above, normally the setting is autoselected, but before, you'll need to teach your network profile to set **lxc.network.ipv4.gateway** to **auto** when using a classic ipv4 configuration.

Thus you'll need

```
lxc.network_profile.foo:
  etho:
    link: lxcbr0
    ipv4.gateway: auto
```

Tricky network setups Examples

This example covers how to make a container with both an internal ip and a public routable ip, wired on two veth pairs.

The another interface which receives directly a public routable ip can't be on the first interface that we reserve for private inter LXC networking.

```
lxc.network_profile.foo:
  eth0: {gateway: null, bridge: lxcbr0}
  eth1:
    # replace that by your main interface
    'link': 'br0'
    'mac': '00:16:5b:01:24:e1'
    'gateway': '2.20.9.14'
    'ipv4': '2.20.9.1'
```

Creating a Container on the CLI

From a Template

LXC is commonly distributed with several template scripts in /usr/share/lxc/templates. Some distros may package these separately in an **lxc-templates** package, so make sure to check if this is the case.

There are LXC template scripts for several different operating systems, but some of them are designed to use tools specific to a given distribution. For instance, the `ubuntu` template uses deb_bootstrap, the `centos` template uses yum, etc., making these templates impractical when a container from a different OS is desired.

The *lxc.create* function is used to create containers using a template script. To create a CentOS container named `container1` on a CentOS minion named `mycentosminion`, using the `centos` LXC template, one can simply run the following command:

```
salt mycentosminion lxc.create container1 template=centos
```

For these instances, there is a `download` template which retrieves minimal container images for several different operating systems. To use this template, it is necessary to provide an `options` parameter when creating the container, with three values:

1. **dist** - the Linux distribution (i.e. `ubuntu` or `centos`)
2. **release** - the release name/version (i.e. `trusty` or `6`)
3. **arch** - CPU architecture (i.e. `amd64` or `i386`)

The *lxc.images* function (new in version 2015.5.0) can be used to list the available images. Alternatively, the releases can be viewed on http://images.linuxcontainers.org/images/. The images are organized in such a way that the **dist**, **release**, and **arch** can be determined using the following URL format: `http://images.linuxcontainers.org/images/dist/release/arch`. For example, `http://images.linuxcontainers.org/images/centos/6/amd64` would correspond to a **dist** of `centos`, a **release** of `6`, and an **arch** of `amd64`.

Therefore, to use the `download` template to create a new 64-bit CentOS 6 container, the following command can be used:

```
salt myminion lxc.create container1 template=download options='{dist: centos, release:
↪6, arch: amd64}'
```

Note: These command-line options can be placed into a *container profile*, like so:

```
lxc.container_profile.cent6:
  template: download
  options:
    dist: centos
    release: 6
    arch: amd64
```

The `options` parameter is not supported in profiles for the 2014.7.x release cycle and earlier, so it would still need to be provided on the command-line.

Cloning an Existing Container

To clone a container, use the *lxc.clone* function:

```
salt myminion lxc.clone container2 orig=container1
```

Using a Container Image

While cloning is a good way to create new containers from a common base container, the source container that is being cloned needs to already exist on the minion. This makes deploying a common container across minions difficult. For this reason, Salt's *lxc.create* is capable of installing a container from a tar archive of another container's rootfs. To create an image of a container named `cent6`, run the following command as root:

```
tar czf cent6.tar.gz -C /var/lib/lxc/cent6 rootfs
```

Note: Before doing this, it is recommended that the container is stopped.

The resulting tarball can then be placed alongside the files in the salt fileserver and referenced using a `salt://` URL. To create a container using an image, use the `image` parameter with *lxc.create*:

```
salt myminion lxc.create new-cent6 image=salt://path/to/cent6.tar.gz
```

Note: Making images of containers with LVM backing

For containers with LVM backing, the rootfs is not mounted, so it is necessary to mount it first before creating the tar archive. When a container is created using LVM backing, an empty `rootfs` dir is handily created within `/var/lib/lxc/container_name`, so this can be used as the mountpoint. The location of the logical volume for the container will be `/dev/vgname/lvname`, where `vgname` is the name of the volume group, and `lvname` is the name of the logical volume. Therefore, assuming a volume group of `vg1`, a logical volume of `lxc-cent6`, and a container name of `cent6`, the following commands can be used to create a tar archive of the rootfs:

```
mount /dev/vg1/lxc-cent6 /var/lib/lxc/cent6/rootfs
tar czf cent6.tar.gz -C /var/lib/lxc/cent6 rootfs
umount /var/lib/lxc/cent6/rootfs
```

Warning: One caveat of using this method of container creation is that `/etc/hosts` is left unmodified. This could cause confusion for some distros if salt-minion is later installed on the container, as the functions that determine the hostname take `/etc/hosts` into account.

Additionally, when creating an rootfs image, be sure to remove `/etc/salt/minion_id` and make sure that `id` is not defined in `/etc/salt/minion`, as this will cause similar issues.

Initializing a New Container as a Salt Minion

The above examples illustrate a few ways to create containers on the CLI, but often it is desirable to also have the new container run as a Minion. To do this, the *lxc.init* function can be used. This function will do the following:

1. Create a new container
2. Optionally set password and/or DNS
3. Bootstrap the minion (using either salt-bootstrap or a custom command)

By default, the new container will be pointed at the same Salt Master as the host machine on which the container was created. It will then request to authenticate with the Master like any other bootstrapped Minion, at which point it can be accepted.

```
salt myminion lxc.init test1 profile=centos
salt-key -a test1
```

For even greater convenience, the *LXC runner* contains a runner function of the same name (*lxc.init*), which creates a keypair, seeds the new minion with it, and pre-accepts the key, allowing for the new Minion to be created and authorized in a single step:

```
salt-run lxc.init test1 host=myminion profile=centos
```

Running Commands Within a Container

For containers which are not running their own Minion, commands can be run within the container in a manner similar to using (`cmd.run <salt.modules.cmdmod.run`). The means of doing this have been changed significantly in version 2015.5.0 (though the deprecated behavior will still be supported for a few releases). Both the old and new usage are documented below.

2015.5.0 and Newer

New functions have been added to mimic the behavior of the functions in the *cmd* module. Below is a table with the *cmd* functions and their *lxc* module equivalents:

Description	*cmd* module	*lxc* module
Run a command and get all output	*cmd.run*	*lxc.run*
Run a command and get just stdout	*cmd.run_stdout*	*lxc.run_stdout*
Run a command and get just stderr	*cmd.run_stderr*	*lxc.run_stderr*
Run a command and get just the retcode	*cmd.retcode*	*lxc.retcode*
Run a command and get all information	*cmd.run_all*	*lxc.run_all*

2014.7.x and Earlier

Earlier Salt releases use a single function (`lxc.run_cmd`) to run commands within containers. Whether stdout, stderr, etc. are returned depends on how the function is invoked.

To run a command and return the stdout:

```
salt myminion lxc.run_cmd web1 'tail /var/log/messages'
```

To run a command and return the stderr:

```
salt myminion lxc.run_cmd web1 'tail /var/log/messages' stdout=False stderr=True
```

To run a command and return the retcode:

```
salt myminion lxc.run_cmd web1 'tail /var/log/messages' stdout=False stderr=False
```

To run a command and return all information:

```
salt myminion lxc.run_cmd web1 'tail /var/log/messages' stdout=True stderr=True
```

Container Management Using salt-cloud

Salt cloud uses under the hood the salt runner and module to manage containers, Please look at *this chapter*

Container Management Using States

Several states are being renamed or otherwise modified in version 2015.5.0. The information in this tutorial refers to the new states. For 2014.7.x and earlier, please refer to the *documentation for the LXC states*.

Ensuring a Container Is Present

To ensure the existence of a named container, use the *lxc.present* state. Here are some examples:

```
# Using a template
web1:
  lxc.present:
    - template: download
    - options:
        dist: centos
```

```
        release: 6
        arch: amd64

# Cloning
web2:
  lxc.present:
    - clone_from: web-base

# Using a rootfs image
web3:
  lxc.present:
    - image: salt://path/to/cent6.tar.gz

# Using profiles
web4:
  lxc.present:
    - profile: centos_web
    - network_profile: centos
```

Warning: The *lxc.present* state will not modify an existing container (in other words, it will not re-create the container). If an *lxc.present* state is run on an existing container, there will be no change and the state will return a `True` result.

The *lxc.present* state also includes an optional `running` parameter which can be used to ensure that a container is running/stopped. Note that there are standalone *lxc.running* and *lxc.stopped* states which can be used for this purpose.

Ensuring a Container Does Not Exist

To ensure that a named container is not present, use the *lxc.absent* state. For example:

```
web1:
  lxc.absent
```

Ensuring a Container is Running/Stopped/Frozen

Containers can be in one of three states:

- **running** - Container is running and active
- **frozen** - Container is running, but all process are blocked and the container is essentially non-active until the container is ``unfrozen''
- **stopped** - Container is not running

Salt has three states (*lxc.running*, *lxc.frozen*, and *lxc.stopped*) which can be used to ensure a container is in one of these states:

```
web1:
  lxc.running

# Restart the container if it was already running
web2:
```

```
  lxc.running:
    - restart: True

web3:
  lxc.stopped

# Explicitly kill all tasks in container instead of gracefully stopping
web4:
  lxc.stopped:
    - kill: True

web5:
  lxc.frozen

# If container is stopped, do not start it (in which case the state will fail)
web6:
  lxc.frozen:
    - start: False
```

4.9.12 Remote execution tutorial

Before continuing make sure you have a working Salt installation by following the *Installation* and the *configuration* instructions.

Stuck?

There are many ways to *get help from the Salt community* including our mailing list and our IRC channel #salt.

Order your minions around

Now that you have a *master* and at least one *minion* communicating with each other you can perform commands on the minion via the **salt** command. Salt calls are comprised of three main components:

```
salt '<target>' <function> [arguments]
```

See also:

salt manpage

target

The target component allows you to filter which minions should run the following function. The default filter is a glob on the minion id. For example:

```
salt '*' test.ping
salt '*.example.org' test.ping
```

Targets can be based on minion system information using the Grains system:

```
salt -G 'os:Ubuntu' test.ping
```

See also:

Grains system

Targets can be filtered by regular expression:

```
salt -E 'virtmach[0-9]' test.ping
```

Targets can be explicitly specified in a list:

```
salt -L 'foo,bar,baz,quo' test.ping
```

Or Multiple target types can be combined in one command:

```
salt -C 'G@os:Ubuntu and webser* or E@database.*' test.ping
```

function

A function is some functionality provided by a module. Salt ships with a large collection of available functions. List all available functions on your minions:

```
salt '*' sys.doc
```

Here are some examples:

Show all currently available minions:

```
salt '*' test.ping
```

Run an arbitrary shell command:

```
salt '*' cmd.run 'uname -a'
```

See also:

the full list of modules

arguments

Space-delimited arguments to the function:

```
salt '*' cmd.exec_code python 'import sys; print sys.version'
```

Optional, keyword arguments are also supported:

```
salt '*' pip.install salt timeout=5 upgrade=True
```

They are always in the form of `kwarg=argument`.

4.9.13 Multi Master Tutorial

As of Salt 0.16.0, the ability to connect minions to multiple masters has been made available. The multi-master system allows for redundancy of Salt masters and facilitates multiple points of communication out to minions. When using a multi-master setup, all masters are running hot, and any active master can be used to send commands out to the minions.

Note: If you need failover capabilities with multiple masters, there is also a MultiMaster-PKI setup available, that uses a different topology MultiMaster-PKI with Failover Tutorial

In 0.16.0, the masters do not share any information, keys need to be accepted on both masters, and shared files need to be shared manually or use tools like the git fileserver backend to ensure that the `file_roots` are kept consistent.

Beginning with Salt 2016.11.0, the *Pluggable Minion Data Cache* was introduced. The minion data cache contains the Salt Mine data, minion grains, and minion pillar information cached on the Salt Master. By default, Salt uses the `localfs` cache module, but other external data stores can be used instead.

Using a pluggable minion cache modules allows for the data stored on a Salt Master about Salt Minions to be replicated on other Salt Masters the Minion is connected to. Please see the *Minion Data Cache* documentation for more information and configuration examples.

Summary of Steps

1. Create a redundant master server
2. Copy primary master key to redundant master
3. Start redundant master
4. Configure minions to connect to redundant master
5. Restart minions
6. Accept keys on redundant master

Prepping a Redundant Master

The first task is to prepare the redundant master. If the redundant master is already running, stop it. There is only one requirement when preparing a redundant master, which is that masters share the same private key. When the first master was created, the master's identifying key pair was generated and placed in the master's `pki_dir`. The default location of the master's key pair is `/etc/salt/pki/master/`. Take the private key, `master.pem`, and copy it to the same location on the redundant master. Do the same for the master's public key, `master.pub`. Assuming that no minions have yet been connected to the new redundant master, it is safe to delete any existing key in this location and replace it.

Note: There is no logical limit to the number of redundant masters that can be used.

Once the new key is in place, the redundant master can be safely started.

Configure Minions

Since minions need to be master-aware, the new master needs to be added to the minion configurations. Simply update the minion configurations to list all connected masters:

```
master:
  - saltmaster1.example.com
  - saltmaster2.example.com
```

Now the minion can be safely restarted.

Note: If the ipc_mode for the minion is set to TCP (default in Windows), then each minion in the multi-minion setup (one per master) needs its own tcp_pub_port and tcp_pull_port.

If these settings are left as the default 4510/4511, each minion object will receive a port 2 higher than the previous. Thus the first minion will get 4510/4511, the second will get 4512/4513, and so on. If these port decisions are unacceptable, you must configure tcp_pub_port and tcp_pull_port with lists of ports for each master. The length of these lists should match the number of masters, and there should not be overlap in the lists.

Now the minions will check into the original master and also check into the new redundant master. Both masters are first-class and have rights to the minions.

Note: Minions can automatically detect failed masters and attempt to reconnect to reconnect to them quickly. To enable this functionality, set *master_alive_interval* in the minion config and specify a number of seconds to poll the masters for connection status.

If this option is not set, minions will still reconnect to failed masters but the first command sent after a master comes back up may be lost while the minion authenticates.

Sharing Files Between Masters

Salt does not automatically share files between multiple masters. A number of files should be shared or sharing of these files should be strongly considered.

Minion Keys

Minion keys can be accepted the normal way using **salt-key** on both masters. Keys accepted, deleted, or rejected on one master will NOT be automatically managed on redundant masters; this needs to be taken care of by running salt-key on both masters or sharing the `/etc/salt/pki/master/{minions,minions_pre,minions_rejected}` directories between masters.

Note: While sharing the **/etc/salt/pki/master** directory will work, it is strongly discouraged, since allowing access to the **master.pem** key outside of Salt creates a *SERIOUS* security risk.

File_Roots

The `file_roots` contents should be kept consistent between masters. Otherwise state runs will not always be consistent on minions since instructions managed by one master will not agree with other masters.

The recommended way to sync these is to use a fileserver backend like gitfs or to keep these files on shared storage.

Important: If using gitfs/git_pillar with the cachedir shared between masters using GlusterFS, nfs, or another network filesystem, and the masters are running Salt 2015.5.9 or later, it is strongly recommended not to turn off `gitfs_global_lock`/`git_pillar_global_lock` as doing so will cause lock files to be removed if they were created by a different master.

Pillar_Roots

Pillar roots should be given the same considerations as *file_roots*.

Master Configurations

While reasons may exist to maintain separate master configurations, it is wise to remember that each master maintains independent control over minions. Therefore, access controls should be in sync between masters unless a valid reason otherwise exists to keep them inconsistent.

These access control options include but are not limited to:

- external_auth
- publisher_acl
- peer
- peer_run

4.9.14 Multi-Master-PKI Tutorial With Failover

This tutorial will explain, how to run a salt-environment where a single minion can have multiple masters and fail-over between them if its current master fails.

The individual steps are

- setup the master(s) to sign its auth-replies
- setup minion(s) to verify master-public-keys
- enable multiple masters on minion(s)
- enable master-check on minion(s)

> Please note, that it is advised to have good knowledge of the salt- authentication and communication-process to understand this tutorial. All of the settings described here, go on top of the default authentication/communication process.

Motivation

The default behaviour of a salt-minion is to connect to a master and accept the masters public key. With each publication, the master sends his public-key for the minion to check and if this public-key ever changes, the minion complains and exits. Practically this means, that there can only be a single master at any given time.

Would it not be much nicer, if the minion could have any number of masters (1:n) and jump to the next master if its current master died because of a network or hardware failure?

Note: There is also a MultiMaster-Tutorial with a different approach and topology than this one, that might also suite your needs or might even be better suited Multi-Master Tutorial

---

It is also desirable, to add some sort of authenticity-check to the very first public key a minion receives from a master. Currently a minions takes the first masters public key for granted.

The Goal

Setup the master to sign the public key it sends to the minions and enable the minions to verify this signature for authenticity.

Prepping the master to sign its public key

For signing to work, both master and minion must have the signing and/or verification settings enabled. If the master signs the public key but the minion does not verify it, the minion will complain and exit. The same happens, when the master does not sign but the minion tries to verify.

The easiest way to have the master sign its public key is to set

```
master_sign_pubkey: True
```

After restarting the salt-master service, the master will automatically generate a new key-pair

```
master_sign.pem
master_sign.pub
```

A custom name can be set for the signing key-pair by setting

```
master_sign_key_name: <name_without_suffix>
```

The master will then generate that key-pair upon restart and use it for creating the public keys signature attached to the auth-reply.

The computation is done for every auth-request of a minion. If many minions auth very often, it is advised to use conf_master:*master_pubkey_signature* and conf_master:*master_use_pubkey_signature* settings described below.

If multiple masters are in use and should sign their auth-replies, the signing key-pair master_sign.* has to be copied to each master. Otherwise a minion will fail to verify the masters public when connecting to a different master than it did initially. That is because the public keys signature was created with a different signing key-pair.

Prepping the minion to verify received public keys

The minion must have the public key (and only that one!) available to be able to verify a signature it receives. That public key (defaults to master_sign.pub) must be copied from the master to the minions pki-directory.

```
/etc/salt/pki/minion/master_sign.pub
```

Important: DO NOT COPY THE master_sign.pem FILE. IT MUST STAY ON THE MASTER AND ONLY THERE!

When that is done, enable the signature checking in the minions configuration

```
verify_master_pubkey_sign: True
```

and restart the minion. For the first try, the minion should be run in manual debug mode.

```
salt-minion -l debug
```

Upon connecting to the master, the following lines should appear on the output:

```
[DEBUG   ] Attempting to authenticate with the Salt Master at 172.16.0.10
[DEBUG   ] Loaded minion key: /etc/salt/pki/minion/minion.pem
[DEBUG   ] salt.crypt.verify_signature: Loading public key
[DEBUG   ] salt.crypt.verify_signature: Verifying signature
[DEBUG   ] Successfully verified signature of master public key with verification
↪public key master_sign.pub
[INFO    ] Received signed and verified master pubkey from master 172.16.0.10
[DEBUG   ] Decrypting the current master AES key
```

If the signature verification fails, something went wrong and it will look like this

```
[DEBUG   ] Attempting to authenticate with the Salt Master at 172.16.0.10
[DEBUG   ] Loaded minion key: /etc/salt/pki/minion/minion.pem
[DEBUG   ] salt.crypt.verify_signature: Loading public key
[DEBUG   ] salt.crypt.verify_signature: Verifying signature
[DEBUG   ] Failed to verify signature of public key
[CRITICAL] The Salt Master server's public key did not authenticate!
```

In a case like this, it should be checked, that the verification pubkey (master_sign.pub) on the minion is the same as the one on the master.

Once the verification is successful, the minion can be started in daemon mode again.

For the paranoid among us, its also possible to verify the publication whenever it is received from the master. That is, for every single auth-attempt which can be quite frequent. For example just the start of the minion will force the signature to be checked 6 times for various things like auth, mine, *highstate*, etc.

If that is desired, enable the setting

```
always_verify_signature: True
```

Multiple Masters For A Minion

Configuring multiple masters on a minion is done by specifying two settings:

- a list of masters addresses
- what type of master is defined

```
master:
    - 172.16.0.10
    - 172.16.0.11
    - 172.16.0.12
```

```
master_type: failover
```

This tells the minion that all the master above are available for it to connect to. When started with this configuration, it will try the master in the order they are defined. To randomize that order, set

```
master_shuffle: True
```

The master-list will then be shuffled before the first connection attempt.

The first master that accepts the minion, is used by the minion. If the master does not yet know the minion, that counts as accepted and the minion stays on that master.

For the minion to be able to detect if its still connected to its current master enable the check for it

```
master_alive_interval: <seconds>
```

If the loss of the connection is detected, the minion will temporarily remove the failed master from the list and try one of the other masters defined (again shuffled if that is enabled).

Testing the setup

At least two running masters are needed to test the failover setup.

Both masters should be running and the minion should be running on the command line in debug mode

```
salt-minion -l debug
```

The minion will connect to the first master from its master list

```
[DEBUG   ] Attempting to authenticate with the Salt Master at 172.16.0.10
[DEBUG   ] Loaded minion key: /etc/salt/pki/minion/minion.pem
[DEBUG   ] salt.crypt.verify_signature: Loading public key
[DEBUG   ] salt.crypt.verify_signature: Verifying signature
[DEBUG   ] Successfully verified signature of master public key with verification
↪public key master_sign.pub
[INFO    ] Received signed and verified master pubkey from master 172.16.0.10
[DEBUG   ] Decrypting the current master AES key
```

A test.ping on the master the minion is currently connected to should be run to test connectivity.

If successful, that master should be turned off. A firewall-rule denying the minions packets will also do the trick.

Depending on the configured conf_minion:*master_alive_interval*, the minion will notice the loss of the connection and log it to its logfile.

```
[INFO    ] Connection to master 172.16.0.10 lost
[INFO    ] Trying to tune in to next master from master-list
```

The minion will then remove the current master from the list and try connecting to the next master

```
[INFO    ] Removing possibly failed master 172.16.0.10 from list of masters
[WARNING ] Master ip address changed from 172.16.0.10 to 172.16.0.11
[DEBUG   ] Attempting to authenticate with the Salt Master at 172.16.0.11
```

If everything is configured correctly, the new masters public key will be verified successfully

```
[DEBUG   ] Loaded minion key: /etc/salt/pki/minion/minion.pem
[DEBUG   ] salt.crypt.verify_signature: Loading public key
[DEBUG   ] salt.crypt.verify_signature: Verifying signature
[DEBUG   ] Successfully verified signature of master public key with verification
↪public key master_sign.pub
```

the authentication with the new master is successful

```
[INFO    ] Received signed and verified master pubkey from master 172.16.0.11
[DEBUG   ] Decrypting the current master AES key
[DEBUG   ] Loaded minion key: /etc/salt/pki/minion/minion.pem
[INFO    ] Authentication with master successful!
```

and the minion can be pinged again from its new master.

Performance Tuning

With the setup described above, the master computes a signature for every auth-request of a minion. With many minions and many auth-requests, that can chew up quite a bit of CPU-Power.

To avoid that, the master can use a pre-created signature of its public-key. The signature is saved as a base64 encoded string which the master reads once when starting and attaches only that string to auth-replies.

Enabling this also gives paranoid users the possibility, to have the signing key-pair on a different system than the actual salt-master and create the public keys signature there. Probably on a system with more restrictive firewall rules, without internet access, less users, etc.

That signature can be created with

```
salt-key --gen-signature
```

This will create a default signature file in the master pki-directory

```
/etc/salt/pki/master/master_pubkey_signature
```

It is a simple text-file with the binary-signature converted to base64.

If no signing-pair is present yet, this will auto-create the signing pair and the signature file in one call

```
salt-key --gen-signature --auto-create
```

Telling the master to use the pre-created signature is done with

```
master_use_pubkey_signature: True
```

That requires the file `master_pubkey_signature' to be present in the masters pki-directory with the correct signature.

If the signature file is named differently, its name can be set with

```
master_pubkey_signature: <filename>
```

With many masters and many public-keys (default and signing), it is advised to use the salt-masters hostname for the signature-files name. Signatures can be easily confused because they do not provide any information about the key the signature was created from.

Verifying that everything works is done the same way as above.

How the signing and verification works

The default key-pair of the salt-master is

```
/etc/salt/pki/master/master.pem
/etc/salt/pki/master/master.pub
```

To be able to create a signature of a message (in this case a public-key), another key-pair has to be added to the setup. Its default name is:

```
master_sign.pem
master_sign.pub
```

The combination of the master.* and master_sign.* key-pairs give the possibility of generating signatures. The signature of a given message is unique and can be verified, if the public-key of the signing-key-pair is available to the recipient (the minion).

The signature of the masters public-key in master.pub is computed with

```
master_sign.pem
master.pub
M2Crypto.EVP.sign_update()
```

This results in a binary signature which is converted to base64 and attached to the auth-reply send to the minion.

With the signing-pairs public-key available to the minion, the attached signature can be verified with

```
master_sign.pub
master.pub
M2Cryptos EVP.verify_update().
```

When running multiple masters, either the signing key-pair has to be present on all of them, or the master_pubkey_signature has to be pre-computed for each master individually (because they all have different public-keys).

> DO NOT PUT THE SAME master.pub ON ALL MASTERS FOR EASE OF USE.

4.9.15 Packaging External Modules for Salt

External Modules Setuptools Entry-Points Support

The salt loader was enhanced to look for external modules by looking at the *salt.loader* entry-point:

> https://pythonhosted.org/setuptools/setuptools.html#dynamic-discovery-of-services-and-plugins

pkg_resources should be installed, which is normally included in setuptools.

> https://pythonhosted.org/setuptools/pkg_resources.html

The package which has custom engines, minion modules, outputters, etc, should require setuptools and should define the following entry points in its setup function:

```
from setuptools import setup, find_packages

setup(name=<NAME>,
      version=<VERSION>,
      description=<DESC>,
      author=<AUTHOR>,
      author_email=<AUTHOR-EMAIL>,
      url=' ... ',
      packages=find_packages(),
      entry_points='''
        [salt.loader]
        engines_dirs = <package>.<loader-module>:engines_dirs
        fileserver_dirs = <package>.<loader-module>:fileserver_dirs
        pillar_dirs = <package>.<loader-module>:pillar_dirs
        returner_dirs = <package>.<loader-module>:returner_dirs
        roster_dirs = <package>.<loader-module>:roster_dirs
      ''')
```

The above setup script example mentions a loader module. here's an example of how *<package>/<loader-module>.py* it should look:

```
# -*- coding: utf-8 -*-

# Import python libs
```

```python
import os

PKG_DIR = os.path.abspath(os.path.dirname(__file__))


def engines_dirs():
    '''
    yield one path per parent directory of where engines can be found
    '''
    yield os.path.join(PKG_DIR, 'engines_1')
    yield os.path.join(PKG_DIR, 'engines_2')


def fileserver_dirs():
    '''
    yield one path per parent directory of where fileserver modules can be found
    '''
    yield os.path.join(PKG_DIR, 'fileserver')


def pillar_dirs():
    '''
    yield one path per parent directory of where external pillar modules can be found
    '''
    yield os.path.join(PKG_DIR, 'pillar')


def returner_dirs():
    '''
    yield one path per parent directory of where returner modules can be found
    '''
    yield os.path.join(PKG_DIR, 'returners')


def roster_dirs():
    '''
    yield one path per parent directory of where roster modules can be found
    '''
    yield os.path.join(PKG_DIR, 'roster')
```

4.9.16 How Do I Use Salt States?

Simplicity, Simplicity, Simplicity

Many of the most powerful and useful engineering solutions are founded on simple principles. Salt States strive to do just that: K.I.S.S. (Keep It Stupidly Simple)

The core of the Salt State system is the SLS, or **S**a**L**t **S**tate file. The SLS is a representation of the state in which a system should be in, and is set up to contain this data in a simple format. This is often called configuration management.

Note: This is just the beginning of using states, make sure to read up on pillar *Pillar* next.

It is All Just Data

Before delving into the particulars, it will help to understand that the SLS file is just a data structure under the hood. While understanding that the SLS is just a data structure isn't critical for understanding and making use of Salt States, it should help bolster knowledge of where the real power is.

SLS files are therefore, in reality, just dictionaries, lists, strings, and numbers. By using this approach Salt can be much more flexible. As one writes more state files, it becomes clearer exactly what is being written. The result is a system that is easy to understand, yet grows with the needs of the admin or developer.

The Top File

The example SLS files in the below sections can be assigned to hosts using a file called **top.sls**. This file is described in-depth *here*.

Default Data - YAML

By default Salt represents the SLS data in what is one of the simplest serialization formats available - YAML.

A typical SLS file will often look like this in YAML:

Note: These demos use some generic service and package names, different distributions often use different names for packages and services. For instance *apache* should be replaced with *httpd* on a Red Hat system. Salt uses the name of the init script, systemd name, upstart name etc. based on what the underlying service management for the platform. To get a list of the available service names on a platform execute the service.get_all salt function.

Information on how to make states work with multiple distributions is later in the tutorial.

```
apache:
  pkg.installed: []
  service.running:
    - require:
      - pkg: apache
```

This SLS data will ensure that the package named apache is installed, and that the apache service is running. The components can be explained in a simple way.

The first line is the ID for a set of data, and it is called the ID Declaration. This ID sets the name of the thing that needs to be manipulated.

The second and third lines contain the state module function to be run, in the format `<state_module>.<function>`. The `pkg.installed` state module function ensures that a software package is installed via the system's native package manager. The `service.running` state module function ensures that a given system daemon is running.

Finally, on line five, is the word `require`. This is called a Requisite Statement, and it makes sure that the Apache service is only started after a successful installation of the apache package.

Adding Configs and Users

When setting up a service like an Apache web server, many more components may need to be added. The Apache configuration file will most likely be managed, and a user and group may need to be set up.

```
apache:
  pkg.installed: []
  service.running:
    - watch:
      - pkg: apache
      - file: /etc/httpd/conf/httpd.conf
      - user: apache
  user.present:
    - uid: 87
    - gid: 87
    - home: /var/www/html
    - shell: /bin/nologin
    - require:
      - group: apache
  group.present:
    - gid: 87
    - require:
      - pkg: apache

/etc/httpd/conf/httpd.conf:
  file.managed:
    - source: salt://apache/httpd.conf
    - user: root
    - group: root
    - mode: 644
```

This SLS data greatly extends the first example, and includes a config file, a user, a group and new requisite statement: `watch`.

Adding more states is easy, since the new user and group states are under the Apache ID, the user and group will be the Apache user and group. The `require` statements will make sure that the user will only be made after the group, and that the group will be made only after the Apache package is installed.

Next, the `require` statement under service was changed to watch, and is now watching 3 states instead of just one. The watch statement does the same thing as require, making sure that the other states run before running the state with a watch, but it adds an extra component. The `watch` statement will run the state's watcher function for any changes to the watched states. So if the package was updated, the config file changed, or the user uid modified, then the service state's watcher will be run. The service state's watcher just restarts the service, so in this case, a change in the config file will also trigger a restart of the respective service.

Moving Beyond a Single SLS

When setting up Salt States in a scalable manner, more than one SLS will need to be used. The above examples were in a single SLS file, but two or more SLS files can be combined to build out a State Tree. The above example also references a file with a strange source - `salt://apache/httpd.conf`. That file will need to be available as well.

The SLS files are laid out in a directory structure on the Salt master; an SLS is just a file and files to download are just files.

The Apache example would be laid out in the root of the Salt file server like this:

```
apache/init.sls
apache/httpd.conf
```

So the httpd.conf is just a file in the apache directory, and is referenced directly.

Do not use dots in SLS file names or their directories

The initial implementation of *top.sls* and *Include declaration* followed the python import model where a slash is represented as a period. This means that a SLS file with a period in the name (besides the suffix period) can not be referenced. For example, webserver_1.0.sls is not referenceable because webserver_1.0 would refer to the directory/file webserver_1/0.sls

The same applies for any subdirectories, this is especially `tricky' when git repos are created. Another command that typically can't render it's output is `` `state.show_sls` `` of a file in a path that contains a dot.

But when using more than one single SLS file, more components can be added to the toolkit. Consider this SSH example:

ssh/init.sls:

```
openssh-client:
  pkg.installed

/etc/ssh/ssh_config:
  file.managed:
    - user: root
    - group: root
    - mode: 644
    - source: salt://ssh/ssh_config
    - require:
      - pkg: openssh-client
```

ssh/server.sls:

```
include:
  - ssh

openssh-server:
  pkg.installed

sshd:
  service.running:
    - require:
      - pkg: openssh-client
      - pkg: openssh-server
      - file: /etc/ssh/banner
      - file: /etc/ssh/sshd_config

/etc/ssh/sshd_config:
  file.managed:
    - user: root
    - group: root
    - mode: 644
    - source: salt://ssh/sshd_config
    - require:
      - pkg: openssh-server

/etc/ssh/banner:
  file:
    - managed
    - user: root
    - group: root
```

```
    - mode: 644
    - source: salt://ssh/banner
    - require:
      - pkg: openssh-server
```

Note: Notice that we use two similar ways of denoting that a file is managed by Salt. In the */etc/ssh/sshd_config* state section above, we use the *file.managed* state declaration whereas with the */etc/ssh/banner* state section, we use the *file* state declaration and add a *managed* attribute to that state declaration. Both ways produce an identical result; the first way -- using *file.managed* -- is merely a shortcut.

Now our State Tree looks like this:

```
apache/init.sls
apache/httpd.conf
ssh/init.sls
ssh/server.sls
ssh/banner
ssh/ssh_config
ssh/sshd_config
```

This example now introduces the `include` statement. The include statement includes another SLS file so that components found in it can be required, watched or as will soon be demonstrated - extended.

The include statement allows for states to be cross linked. When an SLS has an include statement it is literally extended to include the contents of the included SLS files.

Note that some of the SLS files are called init.sls, while others are not. More info on what this means can be found in the *States Tutorial*.

Extending Included SLS Data

Sometimes SLS data needs to be extended. Perhaps the apache service needs to watch additional resources, or under certain circumstances a different file needs to be placed.

In these examples, the first will add a custom banner to ssh and the second will add more watchers to apache to include mod_python.

`ssh/custom-server.sls:`

```
include:
  - ssh.server

extend:
  /etc/ssh/banner:
    file:
      - source: salt://ssh/custom-banner
```

`python/mod_python.sls:`

```
include:
  - apache

extend:
  apache:
    service:
      - watch:
```

```
      - pkg: mod_python

mod_python:
  pkg.installed
```

The `custom-server.sls` file uses the extend statement to overwrite where the banner is being downloaded from, and therefore changing what file is being used to configure the banner.

In the new mod_python SLS the mod_python package is added, but more importantly the apache service was extended to also watch the mod_python package.

Using extend with require or watch

The `extend` statement works differently for `require` or `watch`. It appends to, rather than replacing the requisite component.

Understanding the Render System

Since SLS data is simply that (data), it does not need to be represented with YAML. Salt defaults to YAML because it is very straightforward and easy to learn and use. But the SLS files can be rendered from almost any imaginable medium, so long as a renderer module is provided.

The default rendering system is the `yaml_jinja` renderer. The `yaml_jinja` renderer will first pass the template through the Jinja2 templating system, and then through the YAML parser. The benefit here is that full programming constructs are available when creating SLS files.

Other renderers available are `yaml_mako` and `yaml_wempy` which each use the Mako or Wempy templating system respectively rather than the jinja templating system, and more notably, the pure Python or `py`, `pydsl` & `pyobjects` renderers. The `py` renderer allows for SLS files to be written in pure Python, allowing for the utmost level of flexibility and power when preparing SLS data; while the *`pydsl`* renderer provides a flexible, domain-specific language for authoring SLS data in Python; and the *`pyobjects`* renderer gives you a ``Pythonic'' interface to building state data.

Note: The templating engines described above aren't just available in SLS files. They can also be used in *`file.managed`* states, making file management much more dynamic and flexible. Some examples for using templates in managed files can be found in the documentation for the *`file state`*, as well as the *MooseFS example* below.

Getting to Know the Default - yaml_jinja

The default renderer - `yaml_jinja`, allows for use of the jinja templating system. A guide to the Jinja templating system can be found here: http://jinja.pocoo.org/docs

When working with renderers a few very useful bits of data are passed in. In the case of templating engine based renderers, three critical components are available, `salt`, `grains`, and `pillar`. The `salt` object allows for any Salt function to be called from within the template, and `grains` allows for the Grains to be accessed from within the template. A few examples:

`apache/init.sls:`

```
apache:
  pkg.installed:
```

```
    {% if grains['os'] == 'RedHat'%}
    - name: httpd
    {% endif %}
  service.running:
    {% if grains['os'] == 'RedHat'%}
    - name: httpd
    {% endif %}
    - watch:
      - pkg: apache
      - file: /etc/httpd/conf/httpd.conf
      - user: apache
  user.present:
    - uid: 87
    - gid: 87
    - home: /var/www/html
    - shell: /bin/nologin
    - require:
      - group: apache
  group.present:
    - gid: 87
    - require:
      - pkg: apache

/etc/httpd/conf/httpd.conf:
  file.managed:
    - source: salt://apache/httpd.conf
    - user: root
    - group: root
    - mode: 644
```

This example is simple. If the `os` grain states that the operating system is Red Hat, then the name of the Apache package and service needs to be httpd. A more aggressive way to use Jinja can be found here, in a module to set up a MooseFS distributed filesystem chunkserver:

`moosefs/chunk.sls:`

```
include:
  - moosefs

{% for mnt in salt['cmd.run']('ls /dev/data/moose*').split() %}
/mnt/moose{{ mnt[-1] }}:
  mount.mounted:
    - device: {{ mnt }}
    - fstype: xfs
    - mkmnt: True
  file.directory:
    - user: mfs
    - group: mfs
    - require:
      - user: mfs
      - group: mfs
{% endfor %}

/etc/mfshdd.cfg:
  file.managed:
    - source: salt://moosefs/mfshdd.cfg
    - user: root
    - group: root
```

```
    - mode: 644
    - template: jinja
    - require:
      - pkg: mfs-chunkserver

/etc/mfschunkserver.cfg:
  file.managed:
    - source: salt://moosefs/mfschunkserver.cfg
    - user: root
    - group: root
    - mode: 644
    - template: jinja
    - require:
      - pkg: mfs-chunkserver

mfs-chunkserver:
  pkg.installed: []
mfschunkserver:
  service.running:
    - require:
{% for mnt in salt['cmd.run']('ls /dev/data/moose*') %}
      - mount: /mnt/moose{{ mnt[-1] }}
      - file: /mnt/moose{{ mnt[-1] }}
{% endfor %}
      - file: /etc/mfschunkserver.cfg
      - file: /etc/mfshdd.cfg
      - file: /var/lib/mfs
```

This example shows much more of the available power of Jinja. Multiple for loops are used to dynamically detect available hard drives and set them up to be mounted, and the `salt` object is used multiple times to call shell commands to gather data.

Introducing the Python, PyDSL, and the Pyobjects Renderers

Sometimes the chosen default renderer might not have enough logical power to accomplish the needed task. When this happens, the Python renderer can be used. Normally a YAML renderer should be used for the majority of SLS files, but an SLS file set to use another renderer can be easily added to the tree.

This example shows a very basic Python SLS file:

`python/django.sls:`

```
#!py

def run():
    '''
    Install the django package
    '''
    return {'include': ['python'],
            'django': {'pkg': ['installed']}}
```

This is a very simple example; the first line has an SLS shebang that tells Salt to not use the default renderer, but to use the `py` renderer. Then the run function is defined, the return value from the run function must be a Salt friendly data structure, or better known as a Salt *HighState data structure*.

Alternatively, using the *pydsl* renderer, the above example can be written more succinctly as:

```
#!pydsl

include('python', delayed=True)
state('django').pkg.installed()
```

The *pyobjects* renderer provides an ``Pythonic'' object based approach for building the state data. The above example could be written as:

```
#!pyobjects

include('python')
Pkg.installed("django")
```

These Python examples would look like this if they were written in YAML:

```
include:
  - python

django:
  pkg.installed
```

This example clearly illustrates that; one, using the YAML renderer by default is a wise decision and two, unbridled power can be obtained where needed by using a pure Python SLS.

Running and Debugging Salt States

Once the rules in an SLS are ready, they should be tested to ensure they work properly. To invoke these rules, simply execute `salt '*' state.apply` on the command line. If you get back only hostnames with a `:` after, but no return, chances are there is a problem with one or more of the sls files. On the minion, use the `salt-call` command to examine the output for errors:

```
salt-call state.apply -l debug
```

This should help troubleshoot the issue. The minion can also be started in the foreground in debug mode by running `salt-minion -l debug`.

Next Reading

With an understanding of states, the next recommendation is to become familiar with Salt's pillar interface:

Pillar Walkthrough

4.9.17 States tutorial, part 1 - Basic Usage

The purpose of this tutorial is to demonstrate how quickly you can configure a system to be managed by Salt States. For detailed information about the state system please refer to the full *states reference*.

This tutorial will walk you through using Salt to configure a minion to run the Apache HTTP server and to ensure the server is running.

Before continuing make sure you have a working Salt installation by following the *Installation* and the *configuration* instructions.

Stuck?

There are many ways to *get help from the Salt community* including our mailing list and our IRC channel #salt.

Setting up the Salt State Tree

States are stored in text files on the master and transferred to the minions on demand via the master's File Server. The collection of state files make up the `State Tree`.

To start using a central state system in Salt, the Salt File Server must first be set up. Edit the master config file (*file_roots*) and uncomment the following lines:

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

Note: If you are deploying on FreeBSD via ports, the `file_roots` path defaults to `/usr/local/etc/salt/states`.

Restart the Salt master in order to pick up this change:

```
pkill salt-master
salt-master -d
```

Preparing the Top File

On the master, in the directory uncommented in the previous step, (`/srv/salt` by default), create a new file called *top.sls* and add the following:

```
base:
  '*':
    - webserver
```

The *top file* is separated into environments (discussed later). The default environment is `base`. Under the `base` environment a collection of minion matches is defined; for now simply specify all hosts (`*`).

Targeting minions

The expressions can use any of the targeting mechanisms used by Salt — minions can be matched by glob, PCRE regular expression, or by *grains*. For example:

```
base:
  'os:Fedora':
    - match: grain
    - webserver
```

Create an `sls` file

In the same directory as the *top file*, create a file named `webserver.sls`, containing the following:

```
apache:                 # ID declaration
  pkg:                  # state declaration
    - installed         # function declaration
```

The first line, called the *ID declaration*, is an arbitrary identifier. In this case it defines the name of the package to be installed.

Note: The package name for the Apache httpd web server may differ depending on OS or distro — for example, on Fedora it is `httpd` but on Debian/Ubuntu it is `apache2`.

The second line, called the *State declaration*, defines which of the Salt States we are using. In this example, we are using the *pkg state* to ensure that a given package is installed.

The third line, called the *Function declaration*, defines which function in the *pkg state* module to call.

Renderers

States `sls` files can be written in many formats. Salt requires only a simple data structure and is not concerned with how that data structure is built. Templating languages and DSLs are a dime-a-dozen and everyone has a favorite.

Building the expected data structure is the job of Salt *Renderers* and they are dead-simple to write.

In this tutorial we will be using YAML in Jinja2 templates, which is the default format. The default can be changed by editing *renderer* in the master configuration file.

Install the package

Next, let's run the state we created. Open a terminal on the master and run:

```
salt '*' state.apply
```

Our master is instructing all targeted minions to run *state.apply*. When this function is executed without any SLS targets, a minion will download the *top file* and attempt to match the expressions within it. When the minion does match an expression the modules listed for it will be downloaded, compiled, and executed.

Note: This action is referred to as a ``highstate'', and can be run using the *state.highstate* function. However, to make the usage easier to understand (``highstate'' is not necessarily an intuitive name), a `state.apply` function was added in version 2015.5.0, which when invoked without any SLS names will trigger a highstate. *state.highstate* still exists and can be used, but the documentation (as can be seen above) has been updated to reference `state.apply`, so keep the following in mind as you read the documentation:

- `state.apply` invoked without any SLS names will run *state.highstate*
- `state.apply` invoked with SLS names will run *state.sls*

Once completed, the minion will report back with a summary of all actions taken and all changes made.

Warning: If you have created *custom grain modules*, they will not be available in the top file until after the first *highstate*. To make custom grains available on a minion's first *highstate*, it is recommended to use *this example* to ensure that the custom grains are synced when the minion starts.

SLS File Namespace

Note that in the *example* above, the SLS file `webserver.sls` was referred to simply as `webserver`. The namespace for SLS files when referenced in `top.sls` or an *Include declaration* follows a few simple rules:

1. The `.sls` is discarded (i.e. `webserver.sls` becomes `webserver`).
2. **Subdirectories can be used for better organization.**
   (a) Each subdirectory is represented with a dot (following the Python import model) in Salt states and on the command line . `webserver/dev.sls` on the filesystem is referred to as `webserver.dev` in Salt
   (b) Because slashes are represented as dots, SLS files can not contain dots in the name (other than the dot for the SLS suffix). The SLS file `webserver_1.0.sls` can not be matched, and `webserver_1.0` would match the directory/file `webserver_1/0.sls`
3. A file called `init.sls` in a subdirectory is referred to by the path of the directory. So, `webserver/init.sls` is referred to as `webserver`.
4. If both `webserver.sls` and `webserver/init.sls` happen to exist, `webserver/init.sls` will be ignored and `webserver.sls` will be the file referred to as `webserver`.

Troubleshooting Salt

If the expected output isn't seen, the following tips can help to narrow down the problem.

Turn up logging Salt can be quite chatty when you change the logging setting to `debug`:

```
salt-minion -l debug
```

Run the minion in the foreground By not starting the minion in daemon mode (*-d*) one can view any output from the minion as it works:

```
salt-minion
```

Increase the default timeout value when running **`salt`**. For example, to change the default timeout to 60 seconds:

```
salt -t 60
```

For best results, combine all three:

```
salt-minion -l debug        # On the minion
salt '*' state.apply -t 60  # On the master
```

Next steps

This tutorial focused on getting a simple Salt States configuration working. *Part 2* will build on this example to cover more advanced `sls` syntax and will explore more of the states that ship with Salt.

4.9.18 States tutorial, part 2 - More Complex States, Requisites

Note: This tutorial builds on topics covered in *part 1*. It is recommended that you begin there.

In the *last part* of the Salt States tutorial we covered the basics of installing a package. We will now modify our `webserver.sls` file to have requirements, and use even more Salt States.

Call multiple States

You can specify multiple *State declaration* under an *ID declaration*. For example, a quick modification to our `webserver.sls` to also start Apache if it is not running:

```
apache:
  pkg.installed: []
  service.running:
    - require:
      - pkg: apache
```

Try stopping Apache before running `state.apply` once again and observe the output.

Note: For those running RedhatOS derivatives (Centos, AWS), you will want to specify the service name to be httpd. More on state service here, *service state*. With the example above, just add ``- name: httpd'' above the require line and with the same spacing.

Require other states

We now have a working installation of Apache so let's add an HTML file to customize our website. It isn't exactly useful to have a website without a webserver so we don't want Salt to install our HTML file until Apache is installed and running. Include the following at the bottom of your `webserver/init.sls` file:

```
apache:
  pkg.installed: []
  service.running:
    - require:
      - pkg: apache

/var/www/index.html:                        # ID declaration
  file:                                     # state declaration
    - managed                               # function
    - source: salt://webserver/index.html   # function arg
    - require:                              # requisite declaration
      - pkg: apache                         # requisite reference
```

line 7 is the *ID declaration*. In this example it is the location we want to install our custom HTML file. (**Note**: the default location that Apache serves may differ from the above on your OS or distro. `/srv/www` could also be a likely place to look.)

Line 8 the *State declaration*. This example uses the Salt *file state*.

Line 9 is the *Function declaration*. The *managed function* will download a file from the master and install it in the location specified.

Line 10 is a *Function arg declaration* which, in this example, passes the `source` argument to the *managed function*.

Line 11 is a *Requisite declaration*.

Line 12 is a *Requisite reference* which refers to a state and an ID. In this example, it is referring to the `ID declaration` from our example in *part 1*. This declaration tells Salt not to install the HTML file until Apache is installed.

Next, create the `index.html` file and save it in the `webserver` directory:

```
<!DOCTYPE html>
<html>
    <head><title>Salt rocks</title></head>
    <body>
        <h1>This file brought to you by Salt</h1>
    </body>
</html>
```

Last, call `state.apply` again and the minion will fetch and execute the *highstate* as well as our HTML file from the master using Salt's File Server:

```
salt '*' state.apply
```

Verify that Apache is now serving your custom HTML.

`require` vs. `watch`

There are two *Requisite declaration*, "require", and "watch". Not every state supports "watch". The *`service state`* does support "watch" and will restart a service based on the watch condition.

For example, if you use Salt to install an Apache virtual host configuration file and want to restart Apache whenever that file is changed you could modify our Apache example from earlier as follows:

```
/etc/httpd/extra/httpd-vhosts.conf:
  file.managed:
    - source: salt://webserver/httpd-vhosts.conf

apache:
  pkg.installed: []
  service.running:
    - watch:
      - file: /etc/httpd/extra/httpd-vhosts.conf
    - require:
      - pkg: apache
```

If the pkg and service names differ on your OS or distro of choice you can specify each one separately using a *Name declaration* which explained in *Part 3*.

Next steps

In *part 3* we will discuss how to use includes, extends, and templating to make a more complete State Tree configuration.

4.9.19 States tutorial, part 3 - Templating, Includes, Extends

---

Note: This tutorial builds on topics covered in *part 1* and *part 2*. It is recommended that you begin there.

This part of the tutorial will cover more advanced templating and configuration techniques for `sls` files.

Templating SLS modules

SLS modules may require programming logic or inline execution. This is accomplished with module templating. The default module templating system used is Jinja2 and may be configured by changing the *renderer* value in the master config.

All states are passed through a templating system when they are initially read. To make use of the templating system, simply add some templating markup. An example of an sls module with templating markup may look like this:

```
{% for usr in ['moe','larry','curly'] %}
{{ usr }}:
  user.present
{% endfor %}
```

This templated sls file once generated will look like this:

```
moe:
  user.present
larry:
  user.present
curly:
  user.present
```

Here's a more complex example:

```
# Comments in yaml start with a hash symbol.
# Since jinja rendering occurs before yaml parsing, if you want to include jinja
# in the comments you may need to escape them using 'jinja' comments to prevent
# jinja from trying to render something which is not well-defined jinja.
# e.g.
# {# iterate over the Three Stooges using a {% for %}..{% endfor %} loop
# with the iterator variable {{ usr }} becoming the state ID. #}
{% for usr in 'moe','larry','curly' %}
{{ usr }}:
  group:
    - present
  user:
    - present
    - gid_from_name: True
    - require:
      - group: {{ usr }}
{% endfor %}
```

Using Grains in SLS modules

Often times a state will need to behave differently on different systems. *Salt grains* objects are made available in the template context. The *grains* can be used from within sls modules:

```
apache:
  pkg.installed:
    {% if grains['os'] == 'RedHat' %}
    - name: httpd
    {% elif grains['os'] == 'Ubuntu' %}
```

```
    - name: apache2
    {% endif %}
```

Using Environment Variables in SLS modules

You can use `salt['environ.get']('VARNAME')` to use an environment variable in a Salt state.

```
MYENVVAR="world" salt-call state.template test.sls
```

```
Create a file with contents from an environment variable:
  file.managed:
    - name: /tmp/hello
    - contents: {{ salt['environ.get']('MYENVVAR') }}
```

Error checking:

```
{% set myenvvar = salt['environ.get']('MYENVVAR') %}
{% if myenvvar %}

Create a file with contents from an environment variable:
  file.managed:
    - name: /tmp/hello
    - contents: {{ salt['environ.get']('MYENVVAR') }}

{% else %}

Fail - no environment passed in:
  test.fail_without_changes

{% endif %}
```

Calling Salt modules from templates

All of the Salt modules loaded by the minion are available within the templating system. This allows data to be gathered in real time on the target system. It also allows for shell commands to be run easily from within the sls modules.

The Salt module functions are also made available in the template context as `salt:`

The following example illustrates calling the `group_to_gid` function in the `file` execution module with a single positional argument called `some_group_that_exists`.

```
moe:
  user.present:
    - gid: {{ salt['file.group_to_gid']('some_group_that_exists') }}
```

One way to think about this might be that the `gid` key is being assigned a value equivelent to the following python pseudo-code:

```
import salt.modules.file
file.group_to_gid('some_group_that_exists')
```

Note that for the above example to work, `some_group_that_exists` must exist before the state file is processed by the templating engine.

Below is an example that uses the `network.hw_addr` function to retrieve the MAC address for eth0:

```
salt['network.hw_addr']('eth0')
```

To examine the possible arguments to each execution module function, one can examine the *module reference documentation </ref/modules/all>*:

Advanced SLS module syntax

Lastly, we will cover some incredibly useful techniques for more complex State trees.

Include declaration

A previous example showed how to spread a Salt tree across several files. Similarly, *Requisites and Other Global State Arguments* span multiple files by using an *Include declaration*. For example:

`python/python-libs.sls:`

```
python-dateutil:
  pkg.installed
```

`python/django.sls:`

```
include:
  - python.python-libs

django:
  pkg.installed:
    - require:
      - pkg: python-dateutil
```

Extend declaration

You can modify previous declarations by using an *Extend declaration*. For example the following modifies the Apache tree to also restart Apache when the vhosts file is changed:

`apache/apache.sls:`

```
apache:
  pkg.installed
```

`apache/mywebsite.sls:`

```
include:
  - apache.apache

extend:
  apache:
    service:
      - running
      - watch:
        - file: /etc/httpd/extra/httpd-vhosts.conf

/etc/httpd/extra/httpd-vhosts.conf:
```

```
  file.managed:
    - source: salt://apache/httpd-vhosts.conf
```

Using extend with require or watch

The `extend` statement works differently for `require` or `watch`. It appends to, rather than replacing the requisite component.

Name declaration

You can override the *ID declaration* by using a *Name declaration*. For example, the previous example is a bit more maintainable if rewritten as follows:

`apache/mywebsite.sls:`

```
include:
  - apache.apache

extend:
  apache:
    service:
      - running
      - watch:
        - file: mywebsite

mywebsite:
  file.managed:
    - name: /etc/httpd/extra/httpd-vhosts.conf
    - source: salt://apache/httpd-vhosts.conf
```

Names declaration

Even more powerful is using a *Names declaration* to override the *ID declaration* for multiple states at once. This often can remove the need for looping in a template. For example, the first example in this tutorial can be rewritten without the loop:

```
stooges:
  user.present:
    - names:
      - moe
      - larry
      - curly
```

Next steps

In *part 4* we will discuss how to use salt's *`file_roots`* to set up a workflow in which states can be ``promoted'' from dev, to QA, to production.

4.9.20 States tutorial, part 4

Note: This tutorial builds on topics covered in *part 1*, *part 2*, and *part 3*. It is recommended that you begin there.

This part of the tutorial will show how to use salt's *file_roots* to set up a workflow in which states can be ``promoted'' from dev, to QA, to production.

Salt fileserver path inheritance

Salt's fileserver allows for more than one root directory per environment, like in the below example, which uses both a local directory and a secondary location shared to the salt master via NFS:

```
# In the master config file (/etc/salt/master)
file_roots:
  base:
    - /srv/salt
    - /mnt/salt-nfs/base
```

Salt's fileserver collapses the list of root directories into a single virtual environment containing all files from each root. If the same file exists at the same relative path in more than one root, then the top-most match ``wins''. For example, if `/srv/salt/foo.txt` and `/mnt/salt-nfs/base/foo.txt` both exist, then `salt://foo.txt` will point to `/srv/salt/foo.txt`.

Note: When using multiple fileserver backends, the order in which they are listed in the *fileserver_backend* parameter also matters. If both `roots` and `git` backends contain a file with the same relative path, and `roots` appears before `git` in the *fileserver_backend* list, then the file in `roots` will ``win'', and the file in gitfs will be ignored.

A more thorough explanation of how Salt's modular fileserver works can be found *here*. We recommend reading this.

Environment configuration

Configure a multiple-environment setup like so:

```
file_roots:
  base:
    - /srv/salt/prod
  qa:
    - /srv/salt/qa
    - /srv/salt/prod
  dev:
    - /srv/salt/dev
    - /srv/salt/qa
    - /srv/salt/prod
```

Given the path inheritance described above, files within `/srv/salt/prod` would be available in all environments. Files within `/srv/salt/qa` would be available in both `qa`, and `dev`. Finally, the files within `/srv/salt/dev` would only be available within the `dev` environment.

Based on the order in which the roots are defined, new files/states can be placed within `/srv/salt/dev`, and pushed out to the dev hosts for testing.

Those files/states can then be moved to the same relative path within `/srv/salt/qa`, and they are now available only in the `dev` and `qa` environments, allowing them to be pushed to QA hosts and tested.

Finally, if moved to the same relative path within /srv/salt/prod, the files are now available in all three environments.

Requesting files from specific fileserver environments

See *here* for documentation on how to request files from specific environments.

Practical Example

As an example, consider a simple website, installed to /var/www/foobarcom. Below is a top.sls that can be used to deploy the website:

/srv/salt/prod/top.sls:

```
base:
  'web*prod*':
    - webserver.foobarcom
qa:
  'web*qa*':
    - webserver.foobarcom
dev:
  'web*dev*':
    - webserver.foobarcom
```

Using pillar, roles can be assigned to the hosts:

/srv/pillar/top.sls:

```
base:
  'web*prod*':
    - webserver.prod
  'web*qa*':
    - webserver.qa
  'web*dev*':
    - webserver.dev
```

/srv/pillar/webserver/prod.sls:

```
webserver_role: prod
```

/srv/pillar/webserver/qa.sls:

```
webserver_role: qa
```

/srv/pillar/webserver/dev.sls:

```
webserver_role: dev
```

And finally, the SLS to deploy the website:

/srv/salt/prod/webserver/foobarcom.sls:

```
{% if pillar.get('webserver_role', '') %}
/var/www/foobarcom:
  file.recurse:
    - source: salt://webserver/src/foobarcom
    - env: {{ pillar['webserver_role'] }}
```

```
    - user: www
    - group: www
    - dir_mode: 755
    - file_mode: 644
{% endif %}
```

Given the above SLS, the source for the website should initially be placed in `/srv/salt/dev/webserver/src/foobarcom`.

First, let's deploy to dev. Given the configuration in the top file, this can be done using `state.apply`:

```
salt --pillar 'webserver_role:dev' state.apply
```

However, in the event that it is not desirable to apply all states configured in the top file (which could be likely in more complex setups), it is possible to apply just the states for the `foobarcom` website, by invoking `state.apply` with the desired SLS target as an argument:

```
salt --pillar 'webserver_role:dev' state.apply webserver.foobarcom
```

Once the site has been tested in dev, then the files can be moved from `/srv/salt/dev/webserver/src/foobarcom` to `/srv/salt/qa/webserver/src/foobarcom`, and deployed using the following:

```
salt --pillar 'webserver_role:qa' state.apply webserver.foobarcom
```

Finally, once the site has been tested in qa, then the files can be moved from `/srv/salt/qa/webserver/src/foobarcom` to `/srv/salt/prod/webserver/src/foobarcom`, and deployed using the following:

```
salt --pillar 'webserver_role:prod' state.apply webserver.foobarcom
```

Thanks to Salt's fileserver inheritance, even though the files have been moved to within `/srv/salt/prod`, they are still available from the same `salt://` URI in both the qa and dev environments.

Continue Learning

The best way to continue learning about Salt States is to read through the *reference documentation* and to look through examples of existing state trees. Many pre-configured state trees can be found on GitHub in the saltstack-formulas collection of repositories.

If you have any questions, suggestions, or just want to chat with other people who are using Salt, we have a very *active community* and we'd love to hear from you.

In addition, by continuing to the *Orchestrate Runner* docs, you can learn about the powerful orchestration of which Salt is capable.

4.9.21 States Tutorial, Part 5 - Orchestration with Salt

This was moved to *Orchestrate Runner*.

4.9.22 Syslog-ng usage

Overview

Syslog_ng state module is for generating syslog-ng configurations. You can do the following things:

- generate syslog-ng configuration from YAML,
- use non-YAML configuration,
- start, stop or reload syslog-ng.

There is also an execution module, which can check the syntax of the configuration, get the version and other information about syslog-ng.

Configuration

Users can create syslog-ng configuration statements with the `syslog_ng.config` function. It requires a *name* and a *config* parameter. The *name* parameter determines the name of the generated statement and the *config* parameter holds a parsed YAML structure.

A statement can be declared in the following forms (both are equivalent):

```
source.s_localhost:
  syslog_ng.config:
    - config:
        - tcp:
          - ip: "127.0.0.1"
          - port: 1233
```

```
s_localhost:
  syslog_ng.config:
    - config:
        source:
          - tcp:
            - ip: "127.0.0.1"
            - port: 1233
```

The first one is called short form, because it needs less typing. Users can use lists and dictionaries to specify their configuration. The format is quite self describing and there are more examples [at the end](#examples) of this document.

Quotation

The quotation can be tricky sometimes but here are some rules to follow:

- when a string meant to be `"string"` in the generated configuration, it should be like `'"string"'` in the YAML document
- similarly, users should write `"'string'"` to get `'string'` in the generated configuration

Full example

The following configuration is an example, how a complete syslog-ng configuration looks like:

```
# Set the location of the configuration file
set_location:
  module.run:
    - name: syslog_ng.set_config_file
    - m_name: "/home/tibi/install/syslog-ng/etc/syslog-ng.conf"

# The syslog-ng and syslog-ng-ctl binaries are here. You needn't use
# this method if these binaries can be found in a directory in your PATH.
set_bin_path:
  module.run:
    - name: syslog_ng.set_binary_path
    - m_name: "/home/tibi/install/syslog-ng/sbin"

# Writes the first lines into the config file, also erases its previous
# content
write_version:
  module.run:
    - name: syslog_ng.write_version
    - m_name: "3.6"

# There is a shorter form to set the above variables
set_variables:
  module.run:
    - name: syslog_ng.set_parameters
    - version: "3.6"
    - binary_path: "/home/tibi/install/syslog-ng/sbin"
    - config_file: "/home/tibi/install/syslog-ng/etc/syslog-ng.conf"


# Some global options
options.global_options:
  syslog_ng.config:
    - config:
        - time_reap: 30
        - mark_freq: 10
        - keep_hostname: "yes"

source.s_localhost:
  syslog_ng.config:
    - config:
        - tcp:
          - ip: "127.0.0.1"
          - port: 1233

destination.d_log_server:
  syslog_ng.config:
    - config:
        - tcp:
          - "127.0.0.1"
          - port: 1234

log.l_log_to_central_server:
  syslog_ng.config:
    - config:
        - source: s_localhost
        - destination: d_log_server

some_comment:
```

```
  module.run:
    - name: syslog_ng.write_config
    - config: |
        # Multi line
        # comment

# Another mode to use comments or existing configuration snippets
config.other_comment_form:
  syslog_ng.config:
    - config: |
        # Multi line
        # comment
```

The `syslog_ng.reloaded` function can generate syslog-ng configuration from YAML. If the statement (source, destination, parser, etc.) has a name, this function uses the id as the name, otherwise (log statement) it's purpose is like a mandatory comment.

After execution this example the syslog_ng state will generate this file:

```
#Generated by Salt on 2014-08-18 00:11:11
@version: 3.6

options {
    time_reap(
        30
    );
    mark_freq(
        10
    );
    keep_hostname(
        yes
    );
};


source s_localhost {
    tcp(
        ip(
            127.0.0.1
        ),
        port(
            1233
        )
    );
};


destination d_log_server {
    tcp(
        127.0.0.1,
        port(
            1234
        )
    );
};


log {
```

```
    source(
        s_localhost
    );
    destination(
        d_log_server
    );
};


# Multi line
# comment


# Multi line
# comment
```

Users can include arbitrary texts in the generated configuration with using the `config` statement (see the example above).

Syslog_ng module functions

You can use *syslog_ng.set_binary_path* to set the directory which contains the syslog-ng and syslog-ng-ctl binaries. If this directory is in your PATH, you don't need to use this function. There is also a *syslog_ng.set_config_file* function to set the location of the configuration file.

Examples

Simple source

```
source s_tail {
 file(
   "/var/log/apache/access.log",
   follow_freq(1),
   flags(no-parse, validate-utf8)
 );
};
```

```
s_tail:
  # Salt will call the source function of syslog_ng module
  syslog_ng.config:
    - config:
        source:
          - file:
            - file: ''"/var/log/apache/access.log"''
            - follow_freq : 1
            - flags:
              - no-parse
              - validate-utf8
```

OR

```
s_tail:
  syslog_ng.config:
    - config:
```

```
        source:
            - file:
              - '''"/var/log/apache/access.log"'''
              - follow_freq : 1
              - flags:
                - no-parse
                - validate-utf8
```

OR

```
source.s_tail:
  syslog_ng.config:
    - config:
        - file:
          - '''"/var/log/apache/access.log"'''
          - follow_freq : 1
          - flags:
            - no-parse
            - validate-utf8
```

Complex source

```
source s_gsoc2014 {
 tcp(
   ip("0.0.0.0"),
   port(1234),
   flags(no-parse)
 );
};
```

```
s_gsoc2014:
  syslog_ng.config:
    - config:
        source:
          - tcp:
            - ip: 0.0.0.0
            - port: 1234
            - flags: no-parse
```

Filter

```
filter f_json {
 match(
   "@json:"
 );
};
```

```
f_json:
  syslog_ng.config:
    - config:
        filter:
          - match:
            - '''"@json:"'''
```

Template

```
template t_demo_filetemplate {
 template(
   "$ISODATE $HOST $MSG "
 );
 template_escape(
   no
 );
};
```

```
t_demo_filetemplate:
  syslog_ng.config:
    -config:
        template:
          - template:
            - '"$ISODATE $HOST $MSG\n"'
          - template_escape:
            - "no"
```

Rewrite

```
rewrite r_set_message_to_MESSAGE {
 set(
   "${.json.message}",
   value("$MESSAGE")
 );
};
```

```
r_set_message_to_MESSAGE:
  syslog_ng.config:
    - config:
        rewrite:
          - set:
            - '"${.json.message}"'
            - value : '"$MESSAGE"'
```

Global options

```
options {
   time_reap(30);
   mark_freq(10);
   keep_hostname(yes);
};
```

```
global_options:
  syslog_ng.config:
    - config:
        options:
          - time_reap: 30
          - mark_freq: 10
          - keep_hostname: "yes"
```

Log

```
log {
 source(s_gsoc2014);
 junction {
  channel {
   filter(f_json);
   parser(p_json);
   rewrite(r_set_json_tag);
   rewrite(r_set_message_to_MESSAGE);
   destination {
    file(
      "/tmp/json-input.log",
      template(t_gsoc2014)
    );
   };
   flags(final);
  };
  channel {
   filter(f_not_json);
   parser {
    syslog-parser(

    );
   };
   rewrite(r_set_syslog_tag);
   flags(final);
  };
 };
 destination {
  file(
    "/tmp/all.log",
    template(t_gsoc2014)
  );
 };
};
```

```
l_gsoc2014:
  syslog_ng.config:
    - config:
        log:
          - source: s_gsoc2014
          - junction:
            - channel:
              - filter: f_json
              - parser: p_json
              - rewrite: r_set_json_tag
              - rewrite: r_set_message_to_MESSAGE
              - destination:
                - file:
                  - '"/tmp/json-input.log"'
                  - template: t_gsoc2014
              - flags: final
            - channel:
              - filter: f_not_json
              - parser:
                - syslog-parser: []
```

```
              - rewrite: r_set_syslog_tag
              - flags: final
          - destination:
            - file:
              - "/tmp/all.log"
              - template: t_gsoc2014
```

4.9.23 Salt in 10 Minutes

Note: Welcome to SaltStack! I am excited that you are interested in Salt and starting down the path to better infrastructure management. I developed (and am continuing to develop) Salt with the goal of making the best software available to manage computers of almost any kind. I hope you enjoy working with Salt and that the software can solve your real world needs!

- Thomas S Hatch
- Salt creator and Chief Developer
- CTO of SaltStack, Inc.

Getting Started

What is Salt?

Salt is a different approach to infrastructure management, founded on the idea that high-speed communication with large numbers of systems can open up new capabilities. This approach makes Salt a powerful multitasking system that can solve many specific problems in an infrastructure.

The backbone of Salt is the remote execution engine, which creates a high-speed, secure and bi-directional communication net for groups of systems. On top of this communication system, Salt provides an extremely fast, flexible, and easy-to-use configuration management system called `Salt States`.

Installing Salt

SaltStack has been made to be very easy to install and get started. The *installation documents* contain instructions for all supported platforms.

Starting Salt

Salt functions on a master/minion topology. A master server acts as a central control bus for the clients, which are called `minions`. The minions connect back to the master.

Setting Up the Salt Master

Turning on the Salt Master is easy -- just turn it on! The default configuration is suitable for the vast majority of installations. The Salt Master can be controlled by the local Linux/Unix service manager:

On Systemd based platforms (newer Debian, openSUSE, Fedora):

```
systemctl start salt-master
```

On Upstart based systems (Ubuntu, older Fedora/RHEL):

```
service salt-master start
```

On SysV Init systems (Gentoo, older Debian etc.):

```
/etc/init.d/salt-master start
```

Alternatively, the Master can be started directly on the command-line:

```
salt-master -d
```

The Salt Master can also be started in the foreground in debug mode, thus greatly increasing the command output:

```
salt-master -l debug
```

The Salt Master needs to bind to two TCP network ports on the system. These ports are `4505` and `4506`. For more in depth information on firewalling these ports, the firewall tutorial is available *here*.

Finding the Salt Master

When a minion starts, by default it searches for a system that resolves to the `salt` hostname on the network. If found, the minion initiates the handshake and key authentication process with the Salt master. This means that the easiest configuration approach is to set internal DNS to resolve the name `salt` back to the Salt Master IP.

Otherwise, the minion configuration file will need to be edited so that the configuration option `master` points to the DNS name or the IP of the Salt Master:

Note: The default location of the configuration files is `/etc/salt`. Most platforms adhere to this convention, but platforms such as FreeBSD and Microsoft Windows place this file in different locations.

`/etc/salt/minion:`

```
master: saltmaster.example.com
```

Setting up a Salt Minion

Note: The Salt Minion can operate with or without a Salt Master. This walk-through assumes that the minion will be connected to the master, for information on how to run a master-less minion please see the master-less quick-start guide:

Masterless Minion Quickstart

Now that the master can be found, start the minion in the same way as the master; with the platform init system or via the command line directly:

As a daemon:

```
salt-minion -d
```

In the foreground in debug mode:

```
salt-minion -l debug
```

When the minion is started, it will generate an `id` value, unless it has been generated on a previous run and cached (in `/etc/salt/minion_id` by default). This is the name by which the minion will attempt to authenticate to the master. The following steps are attempted, in order to try to find a value that is not `localhost`:

1. The Python function `socket.getfqdn()` is run
2. `/etc/hostname` is checked (non-Windows only)
3. `/etc/hosts` (`%WINDIR%\system32\drivers\etc\hosts` on Windows hosts) is checked for hostnames that map to anything within **127.0.0.0/8**.

If none of the above are able to produce an id which is not `localhost`, then a sorted list of IP addresses on the minion (excluding any within **127.0.0.0/8**) is inspected. The first publicly-routable IP address is used, if there is one. Otherwise, the first privately-routable IP address is used.

If all else fails, then `localhost` is used as a fallback.

Note: Overriding the `id`

The minion id can be manually specified using the *id* parameter in the minion config file. If this configuration value is specified, it will override all other sources for the `id`.

Now that the minion is started, it will generate cryptographic keys and attempt to connect to the master. The next step is to venture back to the master server and accept the new minion's public key.

Using salt-key

Salt authenticates minions using public-key encryption and authentication. For a minion to start accepting commands from the master, the minion keys need to be accepted by the master.

The `salt-key` command is used to manage all of the keys on the master. To list the keys that are on the master:

```
salt-key -L
```

The keys that have been rejected, accepted, and pending acceptance are listed. The easiest way to accept the minion key is to accept all pending keys:

```
salt-key -A
```

Note: Keys should be verified! Print the master key fingerprint by running `salt-key -F master` on the Salt master. Copy the `master.pub` fingerprint from the Local Keys section, and then set this value as the *master_finger* in the minion configuration file. Restart the Salt minion.

On the master, run `salt-key -f minion-id` to print the fingerprint of the minion's public key that was received by the master. On the minion, run `salt-call key.finger --local` to print the fingerprint of the minion key.

On the master:

```
# salt-key -f foo.domain.com
Unaccepted Keys:
foo.domain.com:  39:f9:e4:8a:aa:74:8d:52:1a:ec:92:03:82:09:c8:f9
```

On the minion:

```
# salt-call key.finger --local
local:
    39:f9:e4:8a:aa:74:8d:52:1a:ec:92:03:82:09:c8:f9
```

If they match, approve the key with `salt-key -a foo.domain.com`.

Sending the First Commands

Now that the minion is connected to the master and authenticated, the master can start to command the minion.

Salt commands allow for a vast set of functions to be executed and for specific minions and groups of minions to be targeted for execution.

The `salt` command is comprised of command options, target specification, the function to execute, and arguments to the function.

A simple command to start with looks like this:

```
salt '*' test.ping
```

The `*` is the target, which specifies all minions.

`test.ping` tells the minion to run the *test.ping* function.

In the case of `test.ping`, `test` refers to a *execution module*. `ping` refers to the *ping* function contained in the aforementioned `test` module.

Note: Execution modules are the workhorses of Salt. They do the work on the system to perform various tasks, such as manipulating files and restarting services.

The result of running this command will be the master instructing all of the minions to execute *test.ping* in parallel and return the result.

This is not an actual ICMP ping, but rather a simple function which returns `True`. Using *test.ping* is a good way of confirming that a minion is connected.

Note: Each minion registers itself with a unique minion ID. This ID defaults to the minion's hostname, but can be explicitly defined in the minion config as well by using the *id* parameter.

Of course, there are hundreds of other modules that can be called just as `test.ping` can. For example, the following would return disk usage on all targeted minions:

```
salt '*' disk.usage
```

Getting to Know the Functions

Salt comes with a vast library of functions available for execution, and Salt functions are self-documenting. To see what functions are available on the minions execute the `sys.doc` function:

```
salt '*' sys.doc
```

This will display a very large list of available functions and documentation on them.

Note: Module documentation is also available *on the web*.

These functions cover everything from shelling out to package management to manipulating database servers. They comprise a powerful system management API which is the backbone to Salt configuration management and many other aspects of Salt.

Note: Salt comes with many plugin systems. The functions that are available via the `salt` command are called *Execution Modules*.

Helpful Functions to Know

The *cmd* module contains functions to shell out on minions, such as *cmd.run* and *cmd.run_all*:

```
salt '*' cmd.run 'ls -l /etc'
```

The `pkg` functions automatically map local system package managers to the same salt functions. This means that `pkg.install` will install packages via `yum` on Red Hat based systems, `apt` on Debian systems, etc.:

```
salt '*' pkg.install vim
```

Note: Some custom Linux spins and derivatives of other distributions are not properly detected by Salt. If the above command returns an error message saying that `pkg.install` is not available, then you may need to override the pkg provider. This process is explained *here*.

The *network.interfaces* function will list all interfaces on a minion, along with their IP addresses, netmasks, MAC addresses, etc:

```
salt '*' network.interfaces
```

Changing the Output Format

The default output format used for most Salt commands is called the `nested` outputter, but there are several other outputters that can be used to change the way the output is displayed. For instance, the `pprint` outputter can be used to display the return data using Python's `pprint` module:

```
root@saltmaster:~# salt myminion grains.item pythonpath --out=pprint
{'myminion': {'pythonpath': ['/usr/lib64/python2.7',
                             '/usr/lib/python2.7/plat-linux2',
                             '/usr/lib64/python2.7/lib-tk',
                             '/usr/lib/python2.7/lib-tk',
```

```
                              '/usr/lib/python2.7/site-packages',
                              '/usr/lib/python2.7/site-packages/gst-0.10',
                              '/usr/lib/python2.7/site-packages/gtk-2.0']}}
```

The full list of Salt outputters, as well as example output, can be found *here*.

salt-call

The examples so far have described running commands from the Master using the `salt` command, but when troubleshooting it can be more beneficial to login to the minion directly and use `salt-call`.

Doing so allows you to see the minion log messages specific to the command you are running (which are *not* part of the return data you see when running the command from the Master using `salt`), making it unnecessary to tail the minion log. More information on `salt-call` and how to use it can be found *here*.

Grains

Salt uses a system called *Grains* to build up static data about minions. This data includes information about the operating system that is running, CPU architecture and much more. The grains system is used throughout Salt to deliver platform data to many components and to users.

Grains can also be statically set, this makes it easy to assign values to minions for grouping and managing.

A common practice is to assign grains to minions to specify what the role or roles a minion might be. These static grains can be set in the minion configuration file or via the *grains.setval* function.

Targeting

Salt allows for minions to be targeted based on a wide range of criteria. The default targeting system uses globular expressions to match minions, hence if there are minions named `larry1`, `larry2`, `curly1`, and `curly2`, a glob of `larry*` will match `larry1` and `larry2`, and a glob of `*1` will match `larry1` and `curly1`.

Many other targeting systems can be used other than globs, these systems include:

Regular Expressions Target using PCRE-compliant regular expressions

Grains Target based on grains data: *Targeting with Grains*

Pillar Target based on pillar data: *Targeting with Pillar*

IP Target based on IP address/subnet/range

Compound Create logic to target based on multiple targets: *Targeting with Compound*

Nodegroup Target with nodegroups: *Targeting with Nodegroup*

The concepts of targets are used on the command line with Salt, but also function in many other areas as well, including the state system and the systems used for ACLs and user permissions.

Passing in Arguments

Many of the functions available accept arguments which can be passed in on the command line:

```
salt '*' pkg.install vim
```

This example passes the argument `vim` to the pkg.install function. Since many functions can accept more complex input than just a string, the arguments are parsed through YAML, allowing for more complex data to be sent on the command line:

```
salt '*' test.echo 'foo: bar'
```

In this case Salt translates the string `foo: bar' into the dictionary ``{`foo': `bar'}''

Note: Any line that contains a newline will not be parsed by YAML.

Salt States

Now that the basics are covered the time has come to evaluate `States`. Salt `States`, or the `State System` is the component of Salt made for configuration management.

The state system is already available with a basic Salt setup, no additional configuration is required. States can be set up immediately.

Note: Before diving into the state system, a brief overview of how states are constructed will make many of the concepts clearer. Salt states are based on data modeling and build on a low level data structure that is used to execute each state function. Then more logical layers are built on top of each other.

The high layers of the state system which this tutorial will cover consists of everything that needs to be known to use states, the two high layers covered here are the *sls* layer and the highest layer *highstate*.

Understanding the layers of data management in the State System will help with understanding states, but they never need to be used. Just as understanding how a compiler functions assists when learning a programming language, understanding what is going on under the hood of a configuration management system will also prove to be a valuable asset.

The First SLS Formula

The state system is built on SLS formulas. These formulas are built out in files on Salt's file server. To make a very basic SLS formula open up a file under /srv/salt named vim.sls. The following state ensures that vim is installed on a system to which that state has been applied.

`/srv/salt/vim.sls:`

```
vim:
  pkg.installed
```

Now install vim on the minions by calling the SLS directly:

```
salt '*' state.apply vim
```

This command will invoke the state system and run the `vim` SLS.

Now, to beef up the vim SLS formula, a `vimrc` can be added:

`/srv/salt/vim.sls:`

```
vim:
  pkg.installed: []

/etc/vimrc:
  file.managed:
    - source: salt://vimrc
    - mode: 644
    - user: root
    - group: root
```

Now the desired `vimrc` needs to be copied into the Salt file server to `/srv/salt/vimrc`. In Salt, everything is a file, so no path redirection needs to be accounted for. The `vimrc` file is placed right next to the `vim.sls` file. The same command as above can be executed to all the vim SLS formulas and now include managing the file.

Note: Salt does not need to be restarted/reloaded or have the master manipulated in any way when changing SLS formulas. They are instantly available.

Adding Some Depth

Obviously maintaining SLS formulas right in a single directory at the root of the file server will not scale out to reasonably sized deployments. This is why more depth is required. Start by making an nginx formula a better way, make an nginx subdirectory and add an init.sls file:

`/srv/salt/nginx/init.sls:`

```
nginx:
  pkg.installed: []
  service.running:
    - require:
      - pkg: nginx
```

A few concepts are introduced in this SLS formula.

First is the service statement which ensures that the `nginx` service is running.

Of course, the nginx service can't be started unless the package is installed -- hence the `require` statement which sets up a dependency between the two.

The `require` statement makes sure that the required component is executed before and that it results in success.

Note: The *require* option belongs to a family of options called *requisites*. Requisites are a powerful component of Salt States, for more information on how requisites work and what is available see: *Requisites*

Also evaluation ordering is available in Salt as well: *Ordering States*

This new sls formula has a special name -- `init.sls`. When an SLS formula is named `init.sls` it inherits the name of the directory path that contains it. This formula can be referenced via the following command:

```
salt '*' state.apply nginx
```

Note: `state.apply` is just another remote execution function, just like *`test.ping`* or *`disk.usage`*. It simply takes the name of an SLS file as an argument.

Now that subdirectories can be used, the `vim.sls` formula can be cleaned up. To make things more flexible, move the `vim.sls` and vimrc into a new subdirectory called `edit` and change the `vim.sls` file to reflect the change:

`/srv/salt/edit/vim.sls:`

```
vim:
  pkg.installed

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
    - mode: 644
    - user: root
    - group: root
```

Only the source path to the vimrc file has changed. Now the formula is referenced as `edit.vim` because it resides in the edit subdirectory. Now the edit subdirectory can contain formulas for emacs, nano, joe or any other editor that may need to be deployed.

Next Reading

Two walk-throughs are specifically recommended at this point. First, a deeper run through States, followed by an explanation of Pillar.

1. *Starting States*
2. *Pillar Walkthrough*

An understanding of Pillar is extremely helpful in using States.

Getting Deeper Into States

Two more in-depth States tutorials exist, which delve much more deeply into States functionality.

1. *How Do I Use Salt States?*, covers much more to get off the ground with States.
2. The *States Tutorial* also provides a fantastic introduction.

These tutorials include much more in-depth information including templating SLS formulas etc.

So Much More!

This concludes the initial Salt walk-through, but there are many more things still to learn! These documents will cover important core aspects of Salt:

- *Pillar*
- *Job Management*

A few more tutorials are also available:

- *Remote Execution Tutorial*
- *Standalone Minion*

This still is only scratching the surface, many components such as the reactor and event systems, extending Salt, modular components and more are not covered here. For an overview of all Salt features and documentation, look at the *Table of Contents*.

4.9.24 Salt's Test Suite: An Introduction

Note: This tutorial makes a couple of assumptions. The first assumption is that you have a basic knowledge of Salt. To get up to speed, check out the *Salt Walkthrough*.

The second assumption is that your Salt development environment is already configured and that you have a basic understanding of contributing to the Salt codebase. If you're unfamiliar with either of these topics, please refer to the *Installing Salt for Development* and the *Contributing* pages, respectively.

Salt comes with a powerful integration and unit test suite. The test suite allows for the fully automated run of integration and/or unit tests from a single interface.

Salt's test suite is located under the `tests` directory in the root of Salt's code base and is divided into two main types of tests: *unit tests and integration tests*. The `unit` and `integration` sub-test-suites are located in the `tests` directory, which is where the majority of Salt's test cases are housed.

Getting Set Up For Tests

There are a couple of requirements, in addition to Salt's requirements, that need to be installed in order to run Salt's test suite. You can install these additional requirements using the files located in the `salt/requirements` directory, depending on your relevant version of Python:

```
pip install -r requirements/dev_python27.txt
pip install -r requirements/dev_python34.txt
```

To be able to run integration tests which utilizes ZeroMQ transport, you also need to install additional requirements for it. Make sure you have installed the C/C++ compiler and development libraries and header files needed for your Python version.

This is an example for RedHat-based operating systems:

```
yum install gcc gcc-c++ python-devel
pip install -r requirements/zeromq.txt
```

On Debian, Ubuntu or their derivatives run the following commands:

```
apt-get install build-essential python-dev
pip install -r requirements/zeromq.txt
```

This will install the latest `pycrypto` and `pyzmq` (with bundled `libzmq`) Python modules required for running integration tests suite.

Test Directory Structure

As noted in the introduction to this tutorial, Salt's test suite is located in the `tests` directory in the root of Salt's code base. From there, the tests are divided into two groups `integration` and `unit`. Within each of these directories, the directory structure roughly mirrors the directory structure of Salt's own codebase. For example, the files inside `tests/integration/modules` contains tests for the files located within `salt/modules`.

Note: `tests/integration` and `tests/unit` are the only directories discussed in this tutorial. With the exception of the `tests/runtests.py` file, which is used below in the *Running the Test Suite* section, the other directories and files located in `tests` are outside the scope of this tutorial.

Integration vs. Unit

Given that Salt's test suite contains two powerful, though very different, testing approaches, when should you write integration tests and when should you write unit tests?

Integration tests use Salt masters, minions, and a syndic to test salt functionality directly and focus on testing the interaction of these components. Salt's integration test runner includes functionality to run Salt execution modules, runners, states, shell commands, salt-ssh commands, salt-api commands, and more. This provides a tremendous ability to use Salt to test itself and makes writing such tests a breeze. Integration tests are the preferred method of testing Salt functionality when possible.

Unit tests do not spin up any Salt daemons, but instead find their value in testing singular implementations of individual functions. Instead of testing against specific interactions, unit tests should be used to test a function's logic. Unit tests should be used to test a function's exit point(s) such as any `return` or `raises` statements.

Unit tests are also useful in cases where writing an integration test might not be possible. While the integration test suite is extremely powerful, unfortunately at this time, it does not cover all functional areas of Salt's ecosystem. For example, at the time of this writing, there is not a way to write integration tests for Proxy Minions. Since the test runner will need to be adjusted to account for Proxy Minion processes, unit tests can still provide some testing support in the interim by testing the logic contained inside Proxy Minion functions.

Running the Test Suite

Once all of the *requirements* are installed, the `runtests.py` file in the `salt/tests` directory is used to instantiate Salt's test suite:

```
python tests/runtests.py [OPTIONS]
```

The command above, if executed without any options, will run the entire suite of integration and unit tests. Some tests require certain flags to run, such as destructive tests. If these flags are not included, then the test suite will only perform the tests that don't require special attention.

At the end of the test run, you will see a summary output of the tests that passed, failed, or were skipped.

The test runner also includes a `--help` option that lists all of the various command line options:

```
python tests/runtests.py --help
```

You can also call the test runner as an executable:

```
./tests/runtests.py --help
```

Running Integration Tests

Salt's set of integration tests use Salt to test itself. The integration portion of the test suite includes some built-in Salt daemons that will spin up in preparation of the test run. This list of Salt daemon processes includes:

- 2 Salt Masters
- 2 Salt Minions
- 1 Salt Syndic

These various daemons are used to execute Salt commands and functionality within the test suite, allowing you to write tests to assert against expected or unexpected behaviors.

A simple example of a test utilizing a typical master/minion execution module command is the test for the `test_ping` function in the `tests/integration/modules/test_test.py` file:

```
def test_ping(self):
    '''
    test.ping
    '''
    self.assertTrue(self.run_function('test.ping'))
```

The test above is a very simple example where the `test.ping` function is executed by Salt's test suite runner and is asserting that the minion returned with a `True` response.

Test Selection Options

If you look in the output of the `--help` command of the test runner, you will see a section called `Tests Selection Options`. The options under this section contain various subsections of the integration test suite such as `--modules`, `--ssh`, or `--states`. By selecting any one of these options, the test daemons will spin up and the integration tests in the named subsection will run.

```
./tests/runtests.py --modules
```

Note: The testing subsections listed in the `Tests Selection Options` of the `--help` output *only* apply to the integration tests. They do not run unit tests.

Running Unit Tests

While `./tests/runtests.py` executes the *entire* test suite (barring any tests requiring special flags), the `--unit` flag can be used to run *only* Salt's unit tests. Salt's unit tests include the tests located in the `tests/unit` directory.

The unit tests do not spin up any Salt testing daemons as the integration tests do and execute very quickly compared to the integration tests.

```
./tests/runtests.py --unit
```

Running Specific Tests

There are times when a specific test file, test class, or even a single, individual test need to be executed, such as when writing new tests. In these situations, the `--name` option should be used.

For running a single test file, such as the pillar module test file in the integration test directory, you must provide the file path using `.` instead of `/` as separators and no file extension:

```
./tests/runtests.py --name=integration.modules.test_pillar
./tests/runtests.py -n integration.modules.test_pillar
```

Some test files contain only one test class while other test files contain multiple test classes. To run a specific test class within the file, append the name of the test class to the end of the file path:

```
./tests/runtests.py --name=integration.modules.test_pillar.PillarModuleTest
./tests/runtests.py -n integration.modules.test_pillar.PillarModuleTest
```

To run a single test within a file, append both the name of the test class the individual test belongs to, as well as the name of the test itself:

```
./tests/runtests.py \
  --name=integration.modules.test_pillar.PillarModuleTest.test_data
./tests/runtests.py \
  -n integration.modules.test_pillar.PillarModuleTest.test_data
```

The `--name` and `-n` options can be used for unit tests as well as integration tests. The following command is an example of how to execute a single test found in the `tests/unit/modules/test_cp.py` file:

```
./tests/runtests.py \
  -n unit.modules.test_cp.CpTestCase.test_get_template_success
```

Writing Tests for Salt

Once you're comfortable running tests, you can now start writing them! Be sure to review the *Integration vs. Unit* section of this tutorial to determine what type of test makes the most sense for the code you're testing.

Note: There are many decorators, naming conventions, and code specifications required for Salt test files. We will not be covering all of the these specifics in this tutorial. Please refer to the testing documentation links listed below in the *Additional Testing Documentation* section to learn more about these requirements.

In the following sections, the test examples assume the ``new'' test is added to a test file that is already present and regularly running in the test suite and is written with the correct requirements.

Writing Integration Tests

Since integration tests validate against a running environment, as explained in the *Running Integration Tests* section of this tutorial, integration tests are very easy to write and are generally the preferred method of writing Salt tests.

The following integration test is an example taken from the `test.py` file in the `tests/integration/modules` directory. This test uses the `run_function` method to test the functionality of a traditional execution module command.

The `run_function` method uses the integration test daemons to execute a `module.function` command as you would with Salt. The minion runs the function and returns. The test also uses Python's Assert Functions to test that the minion's return is expected.

```python
def test_ping(self):
    '''
    test.ping
    '''
    self.assertTrue(self.run_function('test.ping'))
```

Args can be passed in to the `run_function` method as well:

```python
def test_echo(self):
    '''
    test.echo
    '''
    self.assertEqual(self.run_function('test.echo', ['text']), 'text')
```

The next example is taken from the tests/integration/modules/test_aliases.py file and demonstrates how to pass kwargs to the run_function call. Also note that this test uses another salt function to ensure the correct data is present (via the aliases.set_target call) before attempting to assert what the aliases.get_target call should return.

```
def test_set_target(self):
    '''
    aliases.set_target and aliases.get_target
    '''
    set_ret = self.run_function(
            'aliases.set_target',
            alias='fred',
            target='bob')
    self.assertTrue(set_ret)
    tgt_ret = self.run_function(
            'aliases.get_target',
            alias='fred')
    self.assertEqual(tgt_ret, 'bob')
```

Using multiple Salt commands in this manner provides two useful benefits. The first is that it provides some additional coverage for the aliases.set_target function. The second benefit is the call to aliases.get_target is not dependent on the presence of any aliases set outside of this test. Tests should not be dependent on the previous execution, success, or failure of other tests. They should be isolated from other tests as much as possible.

While it might be tempting to build out a test file where tests depend on one another before running, this should be avoided. SaltStack recommends that each test should test a single functionality and not rely on other tests. Therefore, when possible, individual tests should also be broken up into singular pieces. These are not hard-and-fast rules, but serve more as recommendations to keep the test suite simple. This helps with debugging code and related tests when failures occur and problems are exposed. There may be instances where large tests use many asserts to set up a use case that protects against potential regressions.

Note: The examples above all use the run_function option to test execution module functions in a traditional master/minion environment. To see examples of how to test other common Salt components such as runners, salt-api, and more, please refer to the *Integration Test Class Examples* documentation.

Destructive vs Non-destructive Tests

Since Salt is used to change the settings and behavior of systems, often, the best approach to run tests is to make actual changes to an underlying system. This is where the concept of destructive integration tests comes into play. Tests can be written to alter the system they are running on. This capability is what fills in the gap needed to properly test aspects of system management like package installation.

To write a destructive test, import and use the destructiveTest decorator for the test method:

```
import integration
from tests.support.helpers import destructiveTest

class PkgTest(integration.ModuleCase):
    @destructiveTest
    def test_pkg_install(self):
        ret = self.run_function('pkg.install', name='finch')
        self.assertSaltTrueReturn(ret)
        ret = self.run_function('pkg.purge', name='finch')
        self.assertSaltTrueReturn(ret)
```

Writing Unit Tests

As explained in the *Integration vs. Unit* section above, unit tests should be written to test the *logic* of a function. This includes focusing on testing `return` and `raises` statements. Substantial effort should be made to mock external resources that are used in the code being tested.

External resources that should be mocked include, but are not limited to, APIs, function calls, external data either globally available or passed in through function arguments, file data, etc. This practice helps to isolate unit tests to test Salt logic. One handy way to think about writing unit tests is to ``block all of the exits''. More information about how to properly mock external resources can be found in Salt's *Unit Test* documentation.

Salt's unit tests utilize Python's mock class as well as MagicMock. The `@patch` decorator is also heavily used when ``blocking all the exits''.

A simple example of a unit test currently in use in Salt is the `test_get_file_not_found` test in the `tests/unit/modules/test_cp.py` file. This test uses the `@patch` decorator and `MagicMock` to mock the return of the call to Salt's `cp.hash_file` execution module function. This ensures that we're testing the `cp.get_file` function directly, instead of inadvertently testing the call to `cp.hash_file`, which is used in `cp.get_file`.

```
def test_get_file_not_found(self):
    '''
    Test if get_file can't find the file.
    '''
    with patch('salt.modules.cp.hash_file', MagicMock(return_value=False)):
        path = 'salt://saltines'
        dest = '/srv/salt/cheese'
        ret = ''
        self.assertEqual(cp.get_file(path, dest), ret)
```

Note that Salt's `cp` module is imported at the top of the file, along with all of the other necessary testing imports. The `get_file` function is then called directed in the testing function, instead of using the `run_function` method as the integration test examples do above.

The call to `cp.get_file` returns an empty string when a `hash_file` isn't found. Therefore, the example above is a good illustration of a unit test ``blocking the exits'' via the `@patch` decorator, as well as testing logic via asserting against the `return` statement in the `if` clause.

There are more examples of writing unit tests of varying complexities available in the following docs:

- *Simple Unit Test Example*
- *Complete Unit Test Example*
- *Complex Unit Test Example*

Note: Considerable care should be made to ensure that you're testing something useful in your test functions. It is very easy to fall into a situation where you have mocked so much of the original function that the test results in only asserting against the data you have provided. This results in a poor and fragile unit test.

Checking for Log Messages

To test to see if a given log message has been emitted, the following pattern can be used

```python
# Import logging handler
from tests.support.helpers import TestsLoggingHandler

# .. inside test
with TestsLoggingHandler() as handler:
    for message in handler.messages:
        if message.startswith('ERROR: This is the error message we seek'):
            break
        else:
            raise AssertionError('Did not find error message')
```

Automated Test Runs

SaltStack maintains a Jenkins server which can be viewed at https://jenkins.saltstack.com. The tests executed from this Jenkins server create fresh virtual machines for each test run, then execute the destructive tests on the new, clean virtual machine. This allows for the execution of tests across supported platforms.

Additional Testing Documentation

In addition to this tutorial, there are some other helpful resources and documentation that go into more depth on Salt's test runner, writing tests for Salt code, and general Python testing documentation. Please see the follow references for more information:

- *Salt's Test Suite Documentation*
- *Integration Tests*
- *Unit Tests*
- MagicMock
- Python Unittest
- Python's Assert Functions

4.10 Troubleshooting

The intent of the troubleshooting section is to introduce solutions to a number of common issues encountered by users and the tools that are available to aid in developing States and Salt code.

4.10.1 Troubleshooting the Salt Master

If your Salt master is having issues such as minions not returning data, slow execution times, or a variety of other issues, the following links contain details on troubleshooting the most common issues encountered:

Troubleshooting the Salt Master

Running in the Foreground

A great deal of information is available via the debug logging system, if you are having issues with minions connecting or not starting run the master in the foreground:

```
# salt-master -l debug
```

Anyone wanting to run Salt daemons via a process supervisor such as monit, runit, or supervisord, should omit the `-d` argument to the daemons and run them in the foreground.

What Ports does the Master Need Open?

For the master, TCP ports 4505 and 4506 need to be open. If you've put both your Salt master and minion in debug mode and don't see an acknowledgment that your minion has connected, it could very well be a firewall interfering with the connection. See our *firewall configuration* page for help opening the firewall on various platforms.

If you've opened the correct TCP ports and still aren't seeing connections, check that no additional access control system such as SELinux or AppArmor is blocking Salt.

Too many open files

The salt-master needs at least 2 sockets per host that connects to it, one for the Publisher and one for response port. Thus, large installations may, upon scaling up the number of minions accessing a given master, encounter:

```
12:45:29,289 [salt.master    ][INFO    ] Starting Salt worker process 38
Too many open files
sock != -1 (tcp_listener.cpp:335)
```

The solution to this would be to check the number of files allowed to be opened by the user running salt-master (root by default):

```
[root@salt-master ~]# ulimit -n
1024
```

If this value is not equal to at least twice the number of minions, then it will need to be raised. For example, in an environment with 1800 minions, the `nofile` limit should be set to no less than 3600. This can be done by creating the file `/etc/security/limits.d/99-salt.conf`, with the following contents:

```
root        hard    nofile        4096
root        soft    nofile        4096
```

Replace `root` with the user under which the master runs, if different.

If your master does not have an `/etc/security/limits.d` directory, the lines can simply be appended to `/etc/security/limits.conf`.

As with any change to resource limits, it is best to stay logged into your current shell and open another shell to run `ulimit -n` again and verify that the changes were applied correctly. Additionally, if your master is running upstart, it may be necessary to specify the `nofile` limit in `/etc/default/salt-master` if upstart isn't respecting your resource limits:

```
limit nofile 4096 4096
```

Note: The above is simply an example of how to set these values, and you may wish to increase them even further if your Salt master is doing more than just running Salt.

Salt Master Stops Responding

There are known bugs with ZeroMQ versions less than 2.1.11 which can cause the Salt master to not respond properly. If you're running a ZeroMQ version greater than or equal to 2.1.9, you can work around the bug by setting the sysctls `net.core.rmem_max` and `net.core.wmem_max` to 16777216. Next, set the third field in `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem` to at least 16777216.

You can do it manually with something like:

```
# echo 16777216 > /proc/sys/net/core/rmem_max
# echo 16777216 > /proc/sys/net/core/wmem_max
# echo "4096 87380 16777216" > /proc/sys/net/ipv4/tcp_rmem
# echo "4096 87380 16777216" > /proc/sys/net/ipv4/tcp_wmem
```

Or with the following Salt state:

```
net.core.rmem_max:
  sysctl:
    - present
    - value: 16777216

net.core.wmem_max:
  sysctl:
    - present
    - value: 16777216

net.ipv4.tcp_rmem:
  sysctl:
    - present
    - value: 4096 87380 16777216

net.ipv4.tcp_wmem:
  sysctl:
    - present
    - value: 4096 87380 16777216
```

Live Python Debug Output

If the master seems to be unresponsive, a SIGUSR1 can be passed to the salt-master threads to display what piece of code is executing. This debug information can be invaluable in tracking down bugs.

To pass a SIGUSR1 to the master, first make sure the master is running in the foreground. Stop the service if it is running as a daemon, and start it in the foreground like so:

```
# salt-master -l debug
```

Then pass the signal to the master when it seems to be unresponsive:

```
# killall -SIGUSR1 salt-master
```

When filing an issue or sending questions to the mailing list for a problem with an unresponsive daemon, be sure to include this information if possible.

Live Salt-Master Profiling

When faced with performance problems one can turn on master process profiling by sending it SIGUSR2.

```
# killall -SIGUSR2 salt-master
```

This will activate `yappi` profiler inside salt-master code, then after some time one must send SIGUSR2 again to stop profiling and save results to file. If run in foreground salt-master will report filename for the results, which are usually located under `/tmp` on Unix-based OSes and `c:\temp` on windows.

Results can then be analyzed with kcachegrind or similar tool.

Commands Time Out or Do Not Return Output

Depending on your OS (this is most common on Ubuntu due to apt-get) you may sometimes encounter times where a `state.apply`, or other long running commands do not return output.

By default the timeout is set to 5 seconds. The timeout value can easily be increased by modifying the `timeout` line within your `/etc/salt/master` configuration file.

Having keys accepted for Salt minions that no longer exist or are not reachable also increases the possibility of timeouts, since the Salt master waits for those systems to return command results.

Passing the -c Option to Salt Returns a Permissions Error

Using the `-c` option with the Salt command modifies the configuration directory. When the configuration file is read it will still base data off of the `root_dir` setting. This can result in unintended behavior if you are expecting files such as `/etc/salt/pki` to be pulled from the location specified with `-c`. Modify the `root_dir` setting to address this behavior.

Salt Master Doesn't Return Anything While Running jobs

When a command being run via Salt takes a very long time to return (package installations, certain scripts, etc.) the master may drop you back to the shell. In most situations the job is still running but Salt has exceeded the set timeout before returning. Querying the job queue will provide the data of the job but is inconvenient. This can be resolved by either manually using the `-t` option to set a longer timeout when running commands (by default it is 5 seconds) or by modifying the master configuration file: `/etc/salt/master` and setting the `timeout` value to change the default timeout for all commands, and then restarting the salt-master service.

Salt Master Auth Flooding

In large installations, care must be taken not to overwhealm the master with authentication requests. Several options can be set on the master which mitigate the chances of an authentication flood from causing an interruption in service.

Note: recon_default:

The average number of seconds to wait between reconnection attempts.

recon_max: The maximum number of seconds to wait between reconnection attempts.

recon_randomize: A flag to indicate whether the recon_default value should be randomized.

acceptance_wait_time: The number of seconds to wait for a reply to each authentication request.

random_reauth_delay: The range of seconds across which the minions should attempt to randomize authentication attempts.

auth_timeout: The total time to wait for the authentication process to complete, regardless of the number of attempts.

Running states locally

To debug the states, you can use call locally.

```
salt-call -l trace --local state.highstate
```

The top.sls file is used to map what SLS modules get loaded onto what minions via the state system.

It is located in the file defined in the `file_roots` variable of the salt master configuration file which is defined by found in `CONFIG_DIR/master`, normally `/etc/salt/master`

The default configuration for the `file_roots` is:

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

So the top file is defaulted to the location `/srv/salt/top.sls`

Salt Master Umask

The salt master uses a cache to track jobs as they are published and returns come back. The recommended umask for a salt-master is *022*, which is the default for most users on a system. Incorrect umasks can result in permission-denied errors when the master tries to access files in its cache.

4.10.2 Troubleshooting the Salt Minion

In the event that your Salt minion is having issues, a variety of solutions and suggestions are available. Please refer to the following links for more information:

Troubleshooting the Salt Minion

Running in the Foreground

A great deal of information is available via the debug logging system, if you are having issues with minions connecting or not starting run the minion in the foreground:

```
# salt-minion -l debug
```

Anyone wanting to run Salt daemons via a process supervisor such as monit, runit, or supervisord, should omit the `-d` argument to the daemons and run them in the foreground.

What Ports does the Minion Need Open?

No ports need to be opened on the minion, as it makes outbound connections to the master. If you've put both your Salt master and minion in debug mode and don't see an acknowledgment that your minion has connected, it could very well be a firewall interfering with the connection. See our *firewall configuration* page for help opening the firewall on various platforms.

If you have netcat installed, you can check port connectivity from the minion with the `nc` command:

```
$ nc -v -z salt.master.ip.addr 4505
Connection to salt.master.ip.addr 4505 port [tcp/unknown] succeeded!
$ nc -v -z salt.master.ip.addr 4506
Connection to salt.master.ip.addr 4506 port [tcp/unknown] succeeded!
```

The Nmap utility can also be used to check if these ports are open:

```
# nmap -sS -q -p 4505-4506 salt.master.ip.addr

Starting Nmap 6.40 ( http://nmap.org ) at 2013-12-29 19:44 CST
Nmap scan report for salt.master.ip.addr (10.0.0.10)
Host is up (0.0026s latency).
PORT     STATE  SERVICE
4505/tcp open   unknown
4506/tcp open   unknown
MAC Address: 00:11:22:AA:BB:CC (Intel)

Nmap done: 1 IP address (1 host up) scanned in 1.64 seconds
```

If you've opened the correct TCP ports and still aren't seeing connections, check that no additional access control system such as SELinux or AppArmor is blocking Salt. Tools like tcptraceroute can also be used to determine if an intermediate device or firewall is blocking the needed TCP ports.

Using salt-call

The `salt-call` command was originally developed for aiding in the development of new Salt modules. Since then, many applications have been developed for running any Salt module locally on a minion. These range from the original intent of salt-call (development assistance), to gathering more verbose output from calls like `state.apply`.

When initially creating your state tree, it is generally recommended to invoke highstates by running `state.apply` directly from the minion with `salt-call`, rather than remotely from the master. This displays far more information about the execution than calling it remotely. For even more verbosity, increase the loglevel using the `-l` argument:

```
# salt-call -l debug state.apply
```

The main difference between using `salt` and using `salt-call` is that `salt-call` is run from the minion, and it only runs the selected function on that minion. By contrast, `salt` is run from the master, and requires you to specify the minions on which to run the command using salt's *targeting system*.

Live Python Debug Output

If the minion seems to be unresponsive, a SIGUSR1 can be passed to the process to display what piece of code is executing. This debug information can be invaluable in tracking down bugs.

To pass a SIGUSR1 to the minion, first make sure the minion is running in the foreground. Stop the service if it is running as a daemon, and start it in the foreground like so:

```
# salt-minion -l debug
```

Then pass the signal to the minion when it seems to be unresponsive:

```
# killall -SIGUSR1 salt-minion
```

When filing an issue or sending questions to the mailing list for a problem with an unresponsive daemon, be sure to include this information if possible.

Multiprocessing in Execution Modules

As is outlined in github issue #6300, Salt cannot use python's multiprocessing pipes and queues from execution modules. Multiprocessing from the execution modules is perfectly viable, it is just necessary to use Salt's event system to communicate back with the process.

The reason for this difficulty is that python attempts to pickle all objects in memory when communicating, and it cannot pickle function objects. Since the Salt loader system creates and manages function objects this causes the pickle operation to fail.

Salt Minion Doesn't Return Anything While Running Jobs Locally

When a command being run via Salt takes a very long time to return (package installations, certain scripts, etc.) the minion may drop you back to the shell. In most situations the job is still running but Salt has exceeded the set timeout before returning. Querying the job queue will provide the data of the job but is inconvenient. This can be resolved by either manually using the `-t` option to set a longer timeout when running commands (by default it is 5 seconds) or by modifying the minion configuration file: `/etc/salt/minion` and setting the `timeout` value to change the default timeout for all commands, and then restarting the salt-minion service.

Note: Modifying the minion timeout value is not required when running commands from a Salt Master. It is only required when running commands locally on the minion.

4.10.3 Running in the Foreground

A great deal of information is available via the debug logging system, if you are having issues with minions connecting or not starting run the minion and/or master in the foreground:

```
salt-master -l debug
salt-minion -l debug
```

Anyone wanting to run Salt daemons via a process supervisor such as monit, runit, or supervisord, should omit the `-d` argument to the daemons and run them in the foreground.

4.10.4 What Ports do the Master and Minion Need Open?

No ports need to be opened up on each minion. For the master, TCP ports 4505 and 4506 need to be open. If you've put both your Salt master and minion in debug mode and don't see an acknowledgment that your minion has connected, it could very well be a firewall.

You can check port connectivity from the minion with the nc command:

```
nc -v -z salt.master.ip 4505
nc -v -z salt.master.ip 4506
```

There is also a *firewall configuration* document that might help as well.

If you've enabled the right TCP ports on your operating system or Linux distribution's firewall and still aren't seeing connections, check that no additional access control system such as SELinux or AppArmor is blocking Salt.

4.10.5 Using salt-call

The `salt-call` command was originally developed for aiding in the development of new Salt modules. Since then, many applications have been developed for running any Salt module locally on a minion. These range from the original intent of salt-call, development assistance, to gathering more verbose output from calls like `state.apply`.

When initially creating your state tree, it is generally recommended to invoke `state.apply` directly from the minion with `salt-call`, rather than remotely from the master. This displays far more information about the execution than calling it remotely. For even more verbosity, increase the loglevel using the `-l` argument:

```
salt-call -l debug state.apply
```

The main difference between using `salt` and using `salt-call` is that `salt-call` is run from the minion, and it only runs the selected function on that minion. By contrast, `salt` is run from the master, and requires you to specify the minions on which to run the command using salt's *targeting system*.

4.10.6 Too many open files

The salt-master needs at least 2 sockets per host that connects to it, one for the Publisher and one for response port. Thus, large installations may, upon scaling up the number of minions accessing a given master, encounter:

```
12:45:29,289 [salt.master    ][INFO    ] Starting Salt worker process 38
Too many open files
sock != -1 (tcp_listener.cpp:335)
```

The solution to this would be to check the number of files allowed to be opened by the user running salt-master (root by default):

```
[root@salt-master ~]# ulimit -n
1024
```

And modify that value to be at least equal to the number of minions x 2. This setting can be changed in limits.conf as the nofile value(s), and activated upon new a login of the specified user.

So, an environment with 1800 minions, would need 1800 x 2 = 3600 as a minimum.

4.10.7 Salt Master Stops Responding

There are known bugs with ZeroMQ versions less than 2.1.11 which can cause the Salt master to not respond properly. If you're running a ZeroMQ version greater than or equal to 2.1.9, you can work around the bug by setting the sysctls `net.core.rmem_max` and `net.core.wmem_max` to 16777216. Next, set the third field in `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem` to at least 16777216.

You can do it manually with something like:

```
# echo 16777216 > /proc/sys/net/core/rmem_max
# echo 16777216 > /proc/sys/net/core/wmem_max
# echo "4096 87380 16777216" > /proc/sys/net/ipv4/tcp_rmem
# echo "4096 87380 16777216" > /proc/sys/net/ipv4/tcp_wmem
```

Or with the following Salt state:

```
net.core.rmem_max:
  sysctl:
    - present
    - value: 16777216

net.core.wmem_max:
  sysctl:
    - present
    - value: 16777216

net.ipv4.tcp_rmem:
  sysctl:
    - present
    - value: 4096 87380 16777216

net.ipv4.tcp_wmem:
  sysctl:
    - present
    - value: 4096 87380 16777216
```

4.10.8 Salt and SELinux

Currently there are no SELinux policies for Salt. For the most part Salt runs without issue when SELinux is running in Enforcing mode. This is because when the minion executes as a daemon the type context is changed to `initrc_t`. The problem with SELinux arises when using salt-call or running the minion in the foreground, since the type context stays `unconfined_t`.

This problem is generally manifest in the rpm install scripts when using the pkg module. Until a full SELinux Policy is available for Salt the solution to this issue is to set the execution context of `salt-call` and `salt-minion` to rpm_exec_t:

```
# CentOS 5 and RHEL 5:
chcon -t system_u:system_r:rpm_exec_t:s0 /usr/bin/salt-minion
chcon -t system_u:system_r:rpm_exec_t:s0 /usr/bin/salt-call

# CentOS 6 and RHEL 6:
chcon system_u:object_r:rpm_exec_t:s0 /usr/bin/salt-minion
chcon system_u:object_r:rpm_exec_t:s0 /usr/bin/salt-call
```

This works well, because the `rpm_exec_t` context has very broad control over other types.

4.10.9 Red Hat Enterprise Linux 5

Salt requires Python 2.6 or 2.7. Red Hat Enterprise Linux 5 and its variants come with Python 2.4 installed by default. When installing on RHEL 5 from the EPEL repository this is handled for you. But, if you run Salt from git, be advised that its dependencies need to be installed from EPEL and that Salt needs to be run with the `python26` executable.

4.10.10 Common YAML Gotchas

An extensive list of YAML idiosyncrasies has been compiled:

YAML Idiosyncrasies

One of Salt's strengths, the use of existing serialization systems for representing SLS data, can also backfire. YAML is a general purpose system and there are a number of things that would seem to make sense in an sls file that cause YAML issues. It is wise to be aware of these issues. While reports or running into them are generally rare they can still crop up at unexpected times.

Spaces vs Tabs

YAML uses spaces, period. Do not use tabs in your SLS files! If strange errors are coming up in rendering SLS files, make sure to check that no tabs have crept in! In Vim, after enabling search highlighting with: `:set hlsearch`, you can check with the following key sequence in normal mode(you can hit *ESC* twice to be sure): `/`, *Ctrl-v*, *Tab*, then hit *Enter*. Also, you can convert tabs to 2 spaces by these commands in Vim: `:set tabstop=2 expandtab` and then `:retab`.

Indentation

The suggested syntax for YAML files is to use 2 spaces for indentation, but YAML will follow whatever indentation system that the individual file uses. Indentation of two spaces works very well for SLS files given the fact that the data is uniform and not deeply nested.

Nested Dictionaries

When dictionaries are nested within other data structures (particularly lists), the indentation logic sometimes changes. Examples of where this might happen include `context` and `default` options from the *file.managed* state:

```
/etc/http/conf/http.conf:
  file:
    - managed
    - source: salt://apache/http.conf
    - user: root
    - group: root
    - mode: 644
    - template: jinja
    - context:
        custom_var: "override"
    - defaults:
        custom_var: "default value"
        other_var: 123
```

Notice that while the indentation is two spaces per level, for the values under the `context` and `defaults` options there is a four-space indent. If only two spaces are used to indent, then those keys will be considered part of the same dictionary that contains the `context` key, and so the data will not be loaded correctly. If using a double indent is not desirable, then a deeply-nested dict can be declared with curly braces:

```
/etc/http/conf/http.conf:
  file:
    - managed
    - source: salt://apache/http.conf
    - user: root
    - group: root
    - mode: 644
    - template: jinja
    - context:
      custom_var: "override"
    - defaults:
        custom_var: "default value"
        other_var: 123
```

Here is a more concrete example of how YAML actually handles these indentations, using the Python interpreter on the command line:

```
>>> import yaml
>>> yaml.safe_load('''mystate:
...   file.managed:
...     - context:
...         some: var''')
{'mystate': {'file.managed': [{'context': {'some': 'var'}}]}}
>>> yaml.safe_load('''mystate:
...   file.managed:
...     - context:
...       some: var''')
{'mystate': {'file.managed': [{'some': 'var', 'context': None}]}}
```

Note that in the second example, `some` is added as another key in the same dictionary, whereas in the first example, it's the start of a new dictionary. That's the distinction. `context` is a common example because it is a keyword arg for many functions, and should contain a dictionary.

True/False, Yes/No, On/Off

PyYAML will load these values as boolean `True` or `False`. Un-capitalized versions will also be loaded as booleans (`true`, `false`, `yes`, `no`, `on`, and `off`). This can be especially problematic when constructing Pillar data. Make sure that your Pillars which need to use the string versions of these values are enclosed in quotes. Pillars will be parsed twice by salt, so you'll need to wrap your values in multiple quotes, including double quotation marks (`" "`) and single quotation marks (`' '`). Note that spaces are included in the quotation type examples for clarity.

Multiple quoting examples looks like this:

```
- '"false"'
- "'True'"
- "'YES'"
- '"No"'
```

Note: When using multiple quotes in this manner, they must be different. Using `"" ""` or `'' ''` won't work in this case (spaces are included in examples for clarity).

The `%' Sign

The % symbol has a special meaning in YAML, it needs to be passed as a string literal:

```
cheese:
  ssh_auth.present:
    - user: tbortels
    - source: salt://ssh_keys/chease.pub
    - config: '%h/.ssh/authorized_keys'
```

Time Expressions

PyYAML will load a time expression as the integer value of that, assuming HH:MM. So for example, 12:00 is loaded by PyYAML as 720. An excellent explanation for why can be found here.

To keep time expressions like this from being loaded as integers, always quote them.

Note: When using a jinja load_yaml map, items must be quoted twice. For example:

```
{% load_yaml as wsus_schedule %}

FRI_10:
  time: '"23:00"'
  day: 6 - Every Friday
SAT_10:
  time: '"06:00"'
  day: 7 - Every Saturday
SAT_20:
  time: '"14:00"'
  day: 7 - Every Saturday
SAT_30:
  time: '"22:00"'
  day: 7 - Every Saturday
SUN_10:
  time: '"06:00"'
  day: 1 - Every Sunday
{% endload %}
```

YAML does not like ``Double Short Decs''

If I can find a way to make YAML accept ``Double Short Decs'' then I will, since I think that double short decs would be awesome. So what is a ``Double Short Dec''? It is when you declare a multiple short decs in one ID. Here is a standard short dec, it works great:

```
vim:
  pkg.installed
```

The short dec means that there are no arguments to pass, so it is not required to add any arguments, and it can save space.

YAML though, gets upset when declaring multiple short decs, for the record…

THIS DOES NOT WORK:

```
vim:
  pkg.installed
  user.present
```

Similarly declaring a short dec in the same ID dec as a standard dec does not work either...

ALSO DOES NOT WORK:

```
fred:
  user.present
  ssh_auth.present:
    - name: AAAAB3NzaC...
    - user: fred
    - enc: ssh-dss
    - require:
      - user: fred
```

The correct way is to define them like this:

```
vim:
  pkg.installed: []
  user.present: []

fred:
  user.present: []
  ssh_auth.present:
    - name: AAAAB3NzaC...
    - user: fred
    - enc: ssh-dss
    - require:
      - user: fred
```

Alternatively, they can be defined the ``old way'', or with multiple ``full decs'':

```
vim:
  pkg:
    - installed
  user:
    - present

fred:
  user:
    - present
  ssh_auth:
    - present
    - name: AAAAB3NzaC...
    - user: fred
    - enc: ssh-dss
    - require:
      - user: fred
```

YAML supports only plain ASCII

According to YAML specification, only ASCII characters can be used.

Within double-quotes, special characters may be represented with C-style escape sequences starting with a backslash (\ ).

Examples:

```
- micro: "\u00b5"
- copyright: "\u00A9"
- A: "\x41"
- alpha: "\u0251"
- Alef: "\u05d0"
```

List of usable Unicode characters will help you to identify correct numbers.

Python can also be used to discover the Unicode number for a character:

```
repr(u"Text with wrong characters i need to figure out")
```

This shell command can find wrong characters in your SLS files:

```
find . -name '*.sls'  -exec  grep --color='auto' -P -n '[^\x00-\x7F]' \{} \;
```

Alternatively you can toggle the *yaml_utf8* setting in your master configuration file. This is still an experimental setting but it should manage the right encoding conversion in salt after yaml states compilations.

Underscores stripped in Integer Definitions

If a definition only includes numbers and underscores, it is parsed by YAML as an integer and all underscores are stripped. To ensure the object becomes a string, it should be surrounded by quotes. More information here.

Here's an example:

```
>>> import yaml
>>> yaml.safe_load('2013_05_10')
20130510
>>> yaml.safe_load('"2013_05_10"')
'2013_05_10'
```

Automatic `datetime` conversion

If there is a value in a YAML file formatted `2014-01-20 14:23:23` or similar, YAML will automatically convert this to a Python `datetime` object. These objects are not msgpack serializable, and so may break core salt functionality. If values such as these are needed in a salt YAML file (specifically a configuration file), they should be formatted with surrounding strings to force YAML to serialize them as strings:

```
>>> import yaml
>>> yaml.safe_load('2014-01-20 14:23:23')
datetime.datetime(2014, 1, 20, 14, 23, 23)
>>> yaml.safe_load('"2014-01-20 14:23:23"')
'2014-01-20 14:23:23'
```

Additionally, numbers formatted like `XXXX-XX-XX` will also be converted (or YAML will attempt to convert them, and error out if it doesn't think the date is a real one). Thus, for example, if a minion were to have an ID of `4017-16-20` the minion would not start because YAML would complain that the date was out of range. The workaround is the same, surround the offending string with quotes:

```
>>> import yaml
>>> yaml.safe_load('4017-16-20')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/yaml/__init__.py", line 93, in safe_load
    return load(stream, SafeLoader)
  File "/usr/local/lib/python2.7/site-packages/yaml/__init__.py", line 71, in load
    return loader.get_single_data()
  File "/usr/local/lib/python2.7/site-packages/yaml/constructor.py", line 39, in get_
↪single_data
    return self.construct_document(node)
  File "/usr/local/lib/python2.7/site-packages/yaml/constructor.py", line 43, in
↪construct_document
    data = self.construct_object(node)
  File "/usr/local/lib/python2.7/site-packages/yaml/constructor.py", line 88, in
↪construct_object
    data = constructor(self, node)
  File "/usr/local/lib/python2.7/site-packages/yaml/constructor.py", line 312, in
↪construct_yaml_timestamp
    return datetime.date(year, month, day)
ValueError: month must be in 1..12
>>> yaml.safe_load('"4017-16-20"')
'4017-16-20'
```

Keys Limited to 1024 Characters

Simple keys are limited to a single line and cannot be longer that 1024 characters. This is a limitation from PyYaml, as seen in a comment in PyYAML's code, and applies to anything parsed by YAML in Salt.

4.10.11 Live Python Debug Output

If the minion or master seems to be unresponsive, a SIGUSR1 can be passed to the processes to display where in the code they are running. If encountering a situation like this, this debug information can be invaluable. First make sure the master of minion are running in the foreground:

```
salt-master -l debug
salt-minion -l debug
```

Then pass the signal to the master or minion when it seems to be unresponsive:

```
killall -SIGUSR1 salt-master
killall -SIGUSR1 salt-minion
```

Also under BSD and macOS in addition to SIGUSR1 signal, debug subroutine set up for SIGINFO which has an advantage of being sent by Ctrl+T shortcut.

When filing an issue or sending questions to the mailing list for a problem with an unresponsive daemon this information can be invaluable.

4.10.12 Salt 0.16.x minions cannot communicate with a 0.17.x master

As of release 0.17.1 you can no longer run different versions of Salt on your Master and Minion servers. This is due to a protocol change for security purposes. The Salt team will continue to attempt to ensure versions are as backwards compatible as possible.

4.10.13 Debugging the Master and Minion

A list of common *master* and *minion* troubleshooting steps provide a starting point for resolving issues you may encounter.

4.11 Frequently Asked Questions

FAQ

- *Frequently Asked Questions*
  - *Is Salt open-core?*
  - *I think I found a bug! What should I do?*
  - *What ports should I open on my firewall?*
  - *I'm seeing weird behavior (including but not limited to packages not installing their users properly)*
  - *My script runs every time I run a* state.apply*. Why?*
  - *When I run* test.ping*, why don't the Minions that aren't responding return anything? Returning* `False` *would be helpful.*
  - *How does Salt determine the Minion's id?*
  - *I'm trying to manage packages/services but I get an error saying that the state is not available. Why?*
  - *Why aren't my custom modules/states/etc. available on my Minions?*
  - *Module* `X` *isn't available, even though the shell command it uses is installed. Why?*
  - *Can I run different versions of Salt on my Master and Minion?*
  - *Does Salt support backing up managed files?*
  - *Is it possible to deploy a file to a specific minion, without other minions having access to it?*
  - *What is the best way to restart a Salt Minion daemon using Salt after upgrade?*
    * *Upgrade without automatic restart*
    * *Restart using states*
    * *Restart using remote executions*
  - *Salting the Salt Master*
  - *Is Targeting using Grain Data Secure?*
  - *Why Did the Value for a Grain Change on Its Own?*

4.11.1 Is Salt open-core?

No. Salt is 100% committed to being open-source, including all of our APIs. It is developed under the Apache 2.0 license, allowing it to be used in both open and proprietary projects.

To expand on this a little:

There is much argument over the actual definition of ``open core''. From our standpoint, Salt is open source because

1. It is a standalone product that anyone is free to use.
2. It is developed in the open with contributions accepted from the community for the good of the project.
3. There are no features of Salt itself that are restricted to separate proprietary products distributed by SaltStack, Inc.
4. Because of our Apache 2.0 license, Salt can be used as the foundation for a project or even a proprietary tool.
5. Our APIs are open and documented (any lack of documentation is an oversight as opposed to an intentional decision by SaltStack the company) and available for use by anyone.

SaltStack the company does make proprietary products which use Salt and its libraries, like company is free to do, but we do so via the APIs, NOT by forking Salt and creating a different, closed-source version of it for paying customers.

4.11.2 I think I found a bug! What should I do?

The salt-users mailing list as well as the salt IRC channel can both be helpful resources to confirm if others are seeing the issue and to assist with immediate debugging.

To report a bug to the Salt project, please follow the instructions in *reporting a bug*.

4.11.3 What ports should I open on my firewall?

Minions need to be able to connect to the Master on TCP ports 4505 and 4506. Minions do not need any inbound ports open. More detailed information on firewall settings can be found *here*.

4.11.4 I'm seeing weird behavior (including but not limited to packages not installing their users properly)

This is often caused by SELinux. Try disabling SELinux or putting it in permissive mode and see if the weird behavior goes away.

4.11.5 My script runs every time I run a *state.apply*. Why?

You are probably using `cmd.run` rather than `cmd.wait`. A `cmd.wait` state will only run when there has been a change in a state that it is watching.

A `cmd.run` state will run the corresponding command *every time* (unless it is prevented from running by the `unless` or `onlyif` arguments).

More details can be found in the documentation for the `cmd` states.

4.11.6 When I run *test.ping*, why don't the Minions that aren't responding return anything? Returning `False` would be helpful.

When you run *test.ping* the Master tells Minions to run commands/functions, and listens for the return data, printing it to the screen when it is received. If it doesn't receive anything back, it doesn't have anything to display for that Minion.

There are a couple options for getting information on Minions that are not responding. One is to use the verbose (`-v`) option when you run salt commands, as it will display ``Minion did not return'' for any Minions which time out.

```
salt -v '*' pkg.install zsh
```

Another option is to use the *manage.down* runner:

```
salt-run manage.down
```

Also, if the Master is under heavy load, it is possible that the CLI will exit without displaying return data for all targeted Minions. However, this doesn't mean that the Minions did not return; this only means that the Salt CLI timed out waiting for a response. Minions will still send their return data back to the Master once the job completes. If any expected Minions are missing from the CLI output, the *jobs.list_jobs* runner can be used to show the job IDs of the jobs that have been run, and the *jobs.lookup_jid* runner can be used to get the return data for that job.

```
salt-run jobs.list_jobs
salt-run jobs.lookup_jid 20130916125524463507
```

If you find that you are often missing Minion return data on the CLI, only to find it with the jobs runners, then this may be a sign that the *worker_threads* value may need to be increased in the master config file. Additionally, running your Salt CLI commands with the `-t` option will make Salt wait longer for the return data before the CLI command exits. For instance, the below command will wait up to 60 seconds for the Minions to return:

```
salt -t 60 '*' test.ping
```

4.11.7 How does Salt determine the Minion's id?

If the Minion id is not configured explicitly (using the *id* parameter), Salt will determine the id based on the host-name. Exactly how this is determined varies a little between operating systems and is described in detail *here*.

4.11.8 I'm trying to manage packages/services but I get an error saying that the state is not available. Why?

Salt detects the Minion's operating system and assigns the correct package or service management module based on what is detected. However, for certain custom spins and OS derivatives this detection fails. In cases like this, an issue should be opened on our tracker, with the following information:

1. The output of the following command:

   ```
   salt <minion_id> grains.items | grep os
   ```

2. The contents of `/etc/lsb-release`, if present on the Minion.

4.11.9 Why aren't my custom modules/states/etc. available on my Minions?

Custom modules are synced to Minions when *saltutil.sync_modules*, or *saltutil.sync_all* is run.

Similarly, custom states are synced to Minions when *saltutil.sync_states*, or *saltutil.sync_all* is run.

They are both also synced when a *highstate* is triggered.

As of the Fluorine release, as well as 2017.7.7 and 2018.3.2 in their respective release cycles, the `sync` argument to `state.apply`/*state.sls* can be used to sync custom types when running individual SLS files.

Other custom types (renderers, outputters, etc.) have similar behavior, see the documentation for the `saltutil` module for more information.

This reactor example can be used to automatically sync custom types when the minion connects to the master, to help with this chicken-and-egg issue.

4.11.10 Module X isn't available, even though the shell command it uses is installed. Why?

This is most likely a PATH issue. Did you custom-compile the software which the module requires? RHEL/CentOS/etc. in particular override the root user's path in `/etc/init.d/functions`, setting it to `/sbin:/usr/sbin:/bin:/usr/bin`, making software installed into `/usr/local/bin` unavailable to Salt when the Minion is started using the initscript. In version 2014.1.0, Salt will have a better solution for these sort of PATH-related issues, but recompiling the software to install it into a location within the PATH should resolve the issue in the meantime. Alternatively, you can create a symbolic link within the PATH using a `file.symlink` state.

```
/usr/bin/foo:
  file.symlink:
    - target: /usr/local/bin/foo
```

4.11.11 Can I run different versions of Salt on my Master and Minion?

This depends on the versions. In general, it is recommended that Master and Minion versions match.

When upgrading Salt, the master(s) should always be upgraded first. Backwards compatibility for minions running newer versions of salt than their masters is not guaranteed.

Whenever possible, backwards compatibility between new masters and old minions will be preserved. Generally, the only exception to this policy is in case of a security vulnerability.

Recent examples of backwards compatibility breakage include the 0.17.1 release (where all backwards compatibility was broken due to a security fix), and the 2014.1.0 release (which retained compatibility between 2014.1.0 masters and 0.17 minions, but broke compatibility for 2014.1.0 minions and older masters).

4.11.12 Does Salt support backing up managed files?

Yes. Salt provides an easy to use addition to your file.managed states that allow you to back up files via *backup_mode*, backup_mode can be configured on a per state basis, or in the minion config (note that if set in the minion config this would simply be the default method to use, you still need to specify that the file should be backed up!).

4.11.13 Is it possible to deploy a file to a specific minion, without other minions having access to it?

The Salt fileserver does not yet support access control, but it is still possible to do this. As of Salt 2015.5.0, the `file_tree` external pillar is available, and allows the contents of a file to be loaded as Pillar data. This external pillar is capable of assigning Pillar values both to individual minions, and to *nodegroups*. See the `documentation` for details on how to set this up.

Once the external pillar has been set up, the data can be pushed to a minion via a `file.managed` state, using the `contents_pillar` argument:

```
/etc/my_super_secret_file:
  file.managed:
    - user: secret
    - group: secret
    - mode: 600
    - contents_pillar: secret_files:my_super_secret_file
```

In this example, the source file would be located in a directory called `secret_files` underneath the file_tree path for the minion. The syntax for specifying the pillar variable is the same one used for *pillar.get*, with a colon representing a nested dictionary.

Warning: Deploying binary contents using the *file.managed* state is only supported in Salt 2015.8.4 and newer.

4.11.14 What is the best way to restart a Salt Minion daemon using Salt after upgrade?

Updating the `salt-minion` package requires a restart of the `salt-minion` service. But restarting the service while in the middle of a state run interrupts the process of the Minion running states and sending results back to the Master. A common way to workaround that is to schedule restarting the Minion service in the background by issuing a `salt-call` command calling `service.restart` function. This prevents the Minion being disconnected from the Master immediately. Otherwise you would get `Minion did not return. [Not connected]` message as the result of a state run.

Upgrade without automatic restart

Doing the Minion upgrade seems to be a simplest state in your SLS file at first. But the operating systems such as Debian GNU/Linux, Ubuntu and their derivatives start the service after the package installation by default. To prevent this, we need to create policy layer which will prevent the Minion service to restart right after the upgrade:

```
{%- if grains['os_family'] == 'Debian' %}

Disable starting services:
  file.managed:
    - name: /usr/sbin/policy-rc.d
    - user: root
    - group: root
    - mode: 0755
    - contents:
      - '#!/bin/sh'
      - exit 101
    # do not touch if already exists
    - replace: False
    - prereq:
      - pkg: Upgrade Salt Minion

{%- endif %}

Upgrade Salt Minion:
  pkg.installed:
    - name: salt-minion
    - version: 2016.11.3{% if grains['os_family'] == 'Debian' %}+ds-1{% endif %}
    - order: last
```

```
Enable Salt Minion:
  service.enabled:
    - name: salt-minion
    - require:
      - pkg: Upgrade Salt Minion

{%- if grains['os_family'] == 'Debian' %}

Enable starting services:
  file.absent:
    - name: /usr/sbin/policy-rc.d
    - onchanges:
      - pkg: Upgrade Salt Minion

{%- endif %}
```

Restart using states

Now we can apply the workaround to restart the Minion in reliable way. The following example works on UNIX-like operating systems:

```
{%- if grains['os'] != 'Windows' %}
Restart Salt Minion:
  cmd.run:
    - name: 'salt-call service.restart salt-minion'
    - bg: True
    - onchanges:
      - pkg: Upgrade Salt Minion
{%- endif %}
```

Note that restarting the `salt-minion` service on Windows operating systems is not always necessary when performing an upgrade. The installer stops the `salt-minion` service, removes it, deletes the contents of the `\salt\bin` directory, installs the new code, re-creates the `salt-minion` service, and starts it (by default). The restart step **would** be necessary during the upgrade process, however, if the minion config was edited after the upgrade or installation. If a minion restart is necessary, the state above can be edited as follows:

```
Restart Salt Minion:
  cmd.run:
{%- if grains['kernel'] == 'Windows' %}
    - name: 'C:\salt\salt-call.bat service.restart salt-minion'
{%- else %}
    - name: 'salt-call service.restart salt-minion'
{%- endif %}
    - bg: True
    - onchanges:
      - pkg: Upgrade Salt Minion
```

However, it requires more advanced tricks to upgrade from legacy version of Salt (before `2016.3.0`) on UNIX-like operating systems, where executing commands in the background is not supported. You also may need to schedule restarting the Minion service using *masterless mode* after all other states have been applied for Salt versions earlier than `2016.11.0`. This allows the Minion to keep the connection to the Master alive for being able to report the final results back to the Master, while the service is restarting in the background. This state should run last or watch for the `pkg` state changes:

```
Restart Salt Minion:
  cmd.run:
{%- if grains['kernel'] == 'Windows' %}
    - name: 'start powershell "Restart-Service -Name salt-minion"'
{%- else %}
    # fork and disown the process
    - name: |-
        exec 0>&- # close stdin
        exec 1>&- # close stdout
        exec 2>&- # close stderr
        nohup salt-call --local service.restart salt-minion &
{%- endif %}
```

Restart using remote executions

Restart the Minion from the command line:

```
salt -G kernel:Windows cmd.run_bg 'C:\salt\salt-call.bat service.restart salt-minion'
salt -C 'not G@kernel:Windows' cmd.run_bg 'salt-call service.restart salt-minion'
```

4.11.15 Salting the Salt Master

In order to configure a master server via states, the Salt master can also be ``salted'' in order to enforce state on the Salt master as well as the Salt minions. Salting the Salt master requires a Salt minion to be installed on the same machine as the Salt master. Once the Salt minion is installed, the minion configuration file must be pointed to the local Salt master:

```
master: 127.0.0.1
```

Once the Salt master has been ``salted'' with a Salt minion, it can be targeted just like any other minion. If the minion on the salted master is running, the minion can be targeted via any usual `salt` command. Additionally, the `salt-call` command can execute operations to enforce state on the salted master without requiring the minion to be running.

More information about salting the Salt master can be found in the salt-formula for salt itself:

https://github.com/saltstack-formulas/salt-formula

Restarting the `salt-master` service using execution module or application of state could be done the same way as for the Salt minion described *above*.

4.11.16 Is Targeting using Grain Data Secure?

Because grains can be set by users that have access to the minion configuration files on the local system, grains are considered less secure than other identifiers in Salt. Use caution when targeting sensitive operations or setting pillar values based on grain data.

The only grain which can be safely used is `grains['id']` which contains the Minion ID.

When possible, you should target sensitive operations and data using the Minion ID. If the Minion ID of a system changes, the Salt Minion's public key must be re-accepted by an administrator on the Salt Master, making it less vulnerable to impersonation attacks.

4.11.17 Why Did the Value for a Grain Change on Its Own?

This is usually the result of an upstream change in an OS distribution that replaces or removes something that Salt was using to detect the grain. Fortunately, when this occurs, you can use Salt to fix it with a command similar to the following:

```
salt -G 'grain:ChangedValue' grains.setvals "{'grain': 'OldValue'}"
```

(Replacing *grain*, *ChangedValue*, and *OldValue* with the grain and values that you want to change / set.)

You should also file an issue describing the change so it can be fixed in Salt.

4.12 Salt Best Practices

Salt's extreme flexibility leads to many questions concerning the structure of configuration files.

This document exists to clarify these points through examples and code.

4.12.1 General rules

1. Modularity and clarity should be emphasized whenever possible.
2. Create clear relations between pillars and states.
3. Use variables when it makes sense but don't overuse them.
4. Store sensitive data in pillar.
5. Don't use grains for matching in your pillar top file for any sensitive pillars.

4.12.2 Structuring States and Formulas

When structuring Salt States and Formulas it is important to begin with the directory structure. A proper directory structure clearly defines the functionality of each state to the user via visual inspection of the state's name.

Reviewing the MySQL Salt Formula it is clear to see the benefits to the end-user when reviewing a sample of the available states:

```
/srv/salt/mysql/files/
/srv/salt/mysql/client.sls
/srv/salt/mysql/map.jinja
/srv/salt/mysql/python.sls
/srv/salt/mysql/server.sls
```

This directory structure would lead to these states being referenced in a top file in the following way:

```
base:
  'web*':
    - mysql.client
    - mysql.python
  'db*':
    - mysql.server
```

This clear definition ensures that the user is properly informed of what each state will do.

Another example comes from the vim-formula:

```
/srv/salt/vim/files/
/srv/salt/vim/absent.sls
/srv/salt/vim/init.sls
/srv/salt/vim/map.jinja
/srv/salt/vim/nerdtree.sls
/srv/salt/vim/pyflakes.sls
/srv/salt/vim/salt.sls
```

Once again viewing how this would look in a top file:

/srv/salt/top.sls:

```
base:
  'web*':
    - vim
    - vim.nerdtree
    - vim.pyflakes
    - vim.salt
  'db*':
    - vim.absent
```

The usage of a clear top-level directory as well as properly named states reduces the overall complexity and leads a user to both understand what will be included at a glance and where it is located.

In addition *Formulas* should be used as often as possible.

Note: Formulas repositories on the saltstack-formulas GitHub organization should not be pointed to directly from systems that automatically fetch new updates such as GitFS or similar tooling. Instead formulas repositories should be forked on GitHub or cloned locally, where unintended, automatic changes will not take place.

4.12.3 Structuring Pillar Files

Pillars are used to store secure and insecure data pertaining to minions. When designing the structure of the `/srv/pillar` directory, the pillars contained within should once again be focused on clear and concise data which users can easily review, modify, and understand.

The `/srv/pillar/` directory is primarily controlled by `top.sls`. It should be noted that the pillar `top.sls` is not used as a location to declare variables and their values. The `top.sls` is used as a way to include other pillar files and organize the way they are matched based on environments or grains.

An example `top.sls` may be as simple as the following:

/srv/pillar/top.sls:

```
base:
  '*':
    - packages
```

Any number of matchers can be added to the base environment. For example, here is an expanded version of the Pillar top file stated above:

/srv/pillar/top.sls:

```
base:
  '*':
    - packages
```

```
  'web*':
    - apache
    - vim
```

Or an even more complicated example, using a variety of matchers in numerous environments:

/srv/pillar/top.sls:

```
base:
  '*':
    - apache
dev:
  'os:Debian':
    - match: grain
    - vim
test:
  '* and not G@os: Debian':
    - match: compound
    - emacs
```

It is clear to see through these examples how the top file provides users with power but when used incorrectly it can lead to confusing configurations. This is why it is important to understand that the top file for pillar is not used for variable definitions.

Each SLS file within the `/srv/pillar/` directory should correspond to the states which it matches.

This would mean that the `apache` pillar file should contain data relevant to Apache. Structuring files in this way once again ensures modularity, and creates a consistent understanding throughout our Salt environment. Users can expect that pillar variables found in an Apache state will live inside of an Apache pillar:

`/srv/pillar/apache.sls`:

```
apache:
  lookup:
    name: httpd
    config:
      tmpl: /etc/httpd/httpd.conf
```

While this pillar file is simple, it shows how a pillar file explicitly relates to the state it is associated with.

4.12.4 Variable Flexibility

Salt allows users to define variables in SLS files. When creating a state variables should provide users with as much flexibility as possible. This means that variables should be clearly defined and easy to manipulate, and that sane defaults should exist in the event a variable is not properly defined. Looking at several examples shows how these different items can lead to extensive flexibility.

Although it is possible to set variables locally, this is generally not preferred:

`/srv/salt/apache/conf.sls`:

```
{% set name = 'httpd' %}
{% set tmpl = 'salt://apache/files/httpd.conf' %}

include:
  - apache

apache_conf:
```

```
  file.managed:
    - name: {{ name }}
    - source: {{ tmpl }}
    - template: jinja
    - user: root
    - watch_in:
      - service: apache
```

When generating this information it can be easily transitioned to the pillar where data can be overwritten, modified, and applied to multiple states, or locations within a single state:

`/srv/pillar/apache.sls`:

```
apache:
  lookup:
    name: httpd
    config:
      tmpl: salt://apache/files/httpd.conf
```

`/srv/salt/apache/conf.sls`:

```
{% from "apache/map.jinja" import apache with context %}

include:
  - apache

apache_conf:
  file.managed:
    - name: {{ salt['pillar.get']('apache:lookup:name') }}
    - source: {{ salt['pillar.get']('apache:lookup:config:tmpl') }}
    - template: jinja
    - user: root
    - watch_in:
      - service: apache
```

This flexibility provides users with a centralized location to modify variables, which is extremely important as an environment grows.

4.12.5 Modularity Within States

Ensuring that states are modular is one of the key concepts to understand within Salt. When creating a state a user must consider how many times the state could be re-used, and what it relies on to operate. Below are several examples which will iteratively explain how a user can go from a state which is not very modular to one that is:

`/srv/salt/apache/init.sls`:

```
httpd:
  pkg:
    - installed
  service.running:
    - enable: True

/etc/httpd/httpd.conf:
  file.managed:
    - source: salt://apache/files/httpd.conf
    - template: jinja
```

```
    - watch_in:
      - service: httpd
```

The example above is probably the worst-case scenario when writing a state. There is a clear lack of focus by naming both the pkg/service, and managed file directly as the state ID. This would lead to changing multiple requires within this state, as well as others that may depend upon the state.

Imagine if a require was used for the `httpd` package in another state, and then suddenly it's a custom package. Now changes need to be made in multiple locations which increases the complexity and leads to a more error prone configuration.

There is also the issue of having the configuration file located in the init, as a user would be unable to simply install the service and use the default conf file.

Our second revision begins to address the referencing by using `-name`, as opposed to direct ID references:

`/srv/salt/apache/init.sls`:

```
apache:
  pkg.installed:
    - name: httpd
  service.running:
    - name: httpd
    - enable: True

apache_conf:
  file.managed:
    - name: /etc/httpd/httpd.conf
    - source: salt://apache/files/httpd.conf
    - template: jinja
    - watch_in:
      - service: apache
```

The above init file is better than our original, yet it has several issues which lead to a lack of modularity. The first of these problems is the usage of static values for items such as the name of the service, the name of the managed file, and the source of the managed file. When these items are hard coded they become difficult to modify and the opportunity to make mistakes arises. It also leads to multiple edits that need to occur when changing these items (imagine if there were dozens of these occurrences throughout the state!). There is also still the concern of the configuration file data living in the same state as the service and package.

In the next example steps will be taken to begin addressing these issues. Starting with the addition of a map.jinja file (as noted in the *Formula documentation*), and modification of static values:

`/srv/salt/apache/map.jinja`:

```
{% set apache = salt['grains.filter_by']({
    'Debian': {
        'server': 'apache2',
        'service': 'apache2',
        'conf': '/etc/apache2/apache.conf',
    },
    'RedHat': {
        'server': 'httpd',
        'service': 'httpd',
        'conf': '/etc/httpd/httpd.conf',
    },
}, merge=salt['pillar.get']('apache:lookup')) %}
```

/srv/pillar/apache.sls:

```
apache:
  lookup:
    config:
      tmpl: salt://apache/files/httpd.conf
```

/srv/salt/apache/init.sls:

```
{% from "apache/map.jinja" import apache with context %}

apache:
  pkg.installed:
    - name: {{ apache.server }}
  service.running:
    - name: {{ apache.service }}
    - enable: True

apache_conf:
  file.managed:
    - name: {{ apache.conf }}
    - source: {{ salt['pillar.get']('apache:lookup:config:tmpl') }}
    - template: jinja
    - user: root
    - watch_in:
      - service: apache
```

The changes to this state now allow us to easily identify the location of the variables, as well as ensuring they are flexible and easy to modify. While this takes another step in the right direction, it is not yet complete. Suppose the user did not want to use the provided conf file, or even their own configuration file, but the default apache conf. With the current state setup this is not possible. To attain this level of modularity this state will need to be broken into two states.

/srv/salt/apache/map.jinja:

```
{% set apache = salt['grains.filter_by']({
    'Debian': {
        'server': 'apache2',
        'service': 'apache2',
        'conf': '/etc/apache2/apache.conf',
    },
    'RedHat': {
        'server': 'httpd',
        'service': 'httpd',
        'conf': '/etc/httpd/httpd.conf',
    },
}, merge=salt['pillar.get']('apache:lookup')) %}
```

/srv/pillar/apache.sls:

```
apache:
  lookup:
    config:
      tmpl: salt://apache/files/httpd.conf
```

/srv/salt/apache/init.sls:

```
{% from "apache/map.jinja" import apache with context %}

apache:
```

```
  pkg.installed:
    - name: {{ apache.server }}
  service.running:
    - name: {{ apache.service }}
    - enable: True
```

`/srv/salt/apache/conf.sls`:

```
{% from "apache/map.jinja" import apache with context %}

include:
  - apache

apache_conf:
  file.managed:
    - name: {{ apache.conf }}
    - source: {{ salt['pillar.get']('apache:lookup:config:tmpl') }}
    - template: jinja
    - user: root
    - watch_in:
      - service: apache
```

This new structure now allows users to choose whether they only wish to install the default Apache, or if they wish, overwrite the default package, service, configuration file location, or the configuration file itself. In addition to this the data has been broken between multiple files allowing for users to identify where they need to change the associated data.

4.12.6 Storing Secure Data

Secure data refers to any information that you would not wish to share with anyone accessing a server. This could include data such as passwords, keys, or other information.

As all data within a state is accessible by EVERY server that is connected it is important to store secure data within pillar. This will ensure that only those servers which require this secure data have access to it. In this example a use can go from an insecure configuration to one which is only accessible by the appropriate hosts:

`/srv/salt/mysql/testerdb.sls`:

```
testdb:
  mysql_database.present:
    - name: testerdb
```

`/srv/salt/mysql/user.sls`:

```
include:
  - mysql.testerdb

testdb_user:
  mysql_user.present:
    - name: frank
    - password: "test3rdb"
    - host: localhost
    - require:
      - sls: mysql.testerdb
```

Many users would review this state and see that the password is there in plain text, which is quite problematic. It results in several issues which may not be immediately visible.

The first of these issues is clear to most users -- the password being visible in this state. This means that any minion will have a copy of this, and therefore the password which is a major security concern as minions may not be locked down as tightly as the master server.

The other issue that can be encountered is access by users on the master. If everyone has access to the states (or their repository), then they are able to review this password. Keeping your password data accessible by only a few users is critical for both security and peace of mind.

There is also the issue of portability. When a state is configured this way it results in multiple changes needing to be made. This was discussed in the sections above but it is a critical idea to drive home. If states are not portable it may result in more work later!

Fixing this issue is relatively simple, the content just needs to be moved to the associated pillar:

`/srv/pillar/mysql.sls`:

```
mysql:
  lookup:
    name: testerdb
    password: test3rdb
    user: frank
    host: localhost
```

`/srv/salt/mysql/testerdb.sls`:

```
testdb:
  mysql_database.present:
    - name: {{ salt['pillar.get']('mysql:lookup:name') }}
```

`/srv/salt/mysql/user.sls`:

```
include:
  - mysql.testerdb

testdb_user:
  mysql_user.present:
    - name: {{ salt['pillar.get']('mysql:lookup:user') }}
    - password: {{ salt['pillar.get']('mysql:lookup:password') }}
    - host: {{ salt['pillar.get']('mysql:lookup:host') }}
    - require:
      - sls: mysql.testerdb
```

Now that the database details have been moved to the associated pillar file, only machines which are targeted via pillar will have access to these details. Access to users who should not be able to review these details can also be prevented while ensuring that they are still able to write states which take advantage of this information.

CHAPTER 5

Remote Execution

Running pre-defined or arbitrary commands on remote hosts, also known as remote execution, is the core function of Salt. The following links explore modules and returners, which are two key elements of remote execution.

Salt Execution Modules

Salt execution modules are called by the remote execution system to perform a wide variety of tasks. These modules provide functionality such as installing packages, restarting a service, running a remote command, transferring files, and so on.

Full list of execution modules Contains: a list of core modules that ship with Salt.

Writing execution modules Contains: a guide on how to write Salt modules.

5.1 Running Commands on Salt Minions

Salt can be controlled by a command line client by the root user on the Salt master. The Salt command line client uses the Salt client API to communicate with the Salt master server. The Salt client is straightforward and simple to use.

Using the Salt client commands can be easily sent to the minions.

Each of these commands accepts an explicit *--config* option to point to either the master or minion configuration file. If this option is not provided and the default configuration file does not exist then Salt falls back to use the environment variables `SALT_MASTER_CONFIG` and `SALT_MINION_CONFIG`.

See also:

Configuration

5.1.1 Using the Salt Command

The Salt command needs a few components to send information to the Salt minions. The target minions need to be defined, the function to call and any arguments the function requires.

Defining the Target Minions

The first argument passed to salt, defines the target minions, the target minions are accessed via their hostname. The default target type is a bash glob:

```
salt '*foo.com' sys.doc
```

Salt can also define the target minions with regular expressions:

```
salt -E '.*' cmd.run 'ls -l | grep foo'
```

Or to explicitly list hosts, salt can take a list:

```
salt -L foo.bar.baz,quo.qux cmd.run 'ps aux | grep foo'
```

More Powerful Targets

See *Targeting*.

Calling the Function

The function to call on the specified target is placed after the target specification.

New in version 0.9.8.

Functions may also accept arguments, space-delimited:

```
salt '*' cmd.exec_code python 'import sys; print sys.version'
```

Optional, keyword arguments are also supported:

```
salt '*' pip.install salt timeout=5 upgrade=True
```

They are always in the form of `kwarg=argument`.

Arguments are formatted as YAML:

```
salt '*' cmd.run 'echo "Hello: $FIRST_NAME"' saltenv='{FIRST_NAME: "Joe"}'
```

Note: dictionaries must have curly braces around them (like the `saltenv` keyword argument above). This was changed in 0.15.1: in the above example, the first argument used to be parsed as the dictionary `{'echo "Hello': '$FIRST_NAME"'}`. This was generally not the expected behavior.

If you want to test what parameters are actually passed to a module, use the `test.arg_repr` command:

```
salt '*' test.arg_repr 'echo "Hello: $FIRST_NAME"' saltenv='{FIRST_NAME: "Joe"}'
```

Finding available minion functions

The Salt functions are self documenting, all of the function documentation can be retried from the minions via the `sys.doc()` function:

```
salt '*' sys.doc
```

Compound Command Execution

If a series of commands needs to be sent to a single target specification then the commands can be sent in a single publish. This can make gathering groups of information faster, and lowers the stress on the network for repeated commands.

Compound command execution works by sending a list of functions and arguments instead of sending a single function and argument. The functions are executed on the minion in the order they are defined on the command line, and then the data from all of the commands are returned in a dictionary. This means that the set of commands are called in a predictable way, and the returned data can be easily interpreted.

Executing compound commands if done by passing a comma delimited list of functions, followed by a comma delimited list of arguments:

```
salt '*' cmd.run,test.ping,test.echo 'cat /proc/cpuinfo',,foo
```

The trick to look out for here, is that if a function is being passed no arguments, then there needs to be a placeholder for the absent arguments. This is why in the above example, there are two commas right next to each other. `test.ping` takes no arguments, so we need to add another comma, otherwise Salt would attempt to pass ``foo'' to `test.ping`.

If you need to pass arguments that include commas, then make sure you add spaces around the commas that separate arguments. For example:

```
salt '*' cmd.run,test.ping,test.echo 'echo "1,2,3"' , , foo
```

You may change the arguments separator using the `--args-separator` option:

```
salt --args-separator=:: '*' some.fun,test.echo params with , comma :: foo
```

5.1.2 CLI Completion

Shell completion scripts for the Salt CLI are available in the `pkg` Salt source directory.

5.2 Writing Execution Modules

Salt execution modules are the functions called by the **salt** command.

5.2.1 Modules Are Easy to Write!

Writing Salt execution modules is straightforward.

A Salt execution module is a Python or Cython module placed in a directory called `_modules/` at the root of the Salt fileserver. When using the default fileserver backend (i.e. *roots*), unless environments are otherwise defined in the *file_roots* config option, the `_modules/` directory would be located in `/srv/salt/_modules` on most systems.

Modules placed in `_modules/` will be synced to the minions when any of the following Salt functions are called:

- `state.apply`
- *saltutil.sync_modules*
- *saltutil.sync_all*

Note that a module's default name is its filename (i.e. `foo.py` becomes module `foo`), but that its name can be overridden by using a *__virtual__ function*.

If a Salt module has errors and cannot be imported, the Salt minion will continue to load without issue and the module with errors will simply be omitted.

If adding a Cython module the file must be named `<modulename>.pyx` so that the loader knows that the module needs to be imported as a Cython module. The compilation of the Cython module is automatic and happens when the minion starts, so only the `*.pyx` file is required.

5.2.2 Zip Archives as Modules

Python 2.3 and higher allows developers to directly import zip archives containing Python code. By setting *enable_zip_modules* to `True` in the minion config, the Salt loader will be able to import `.zip` files in this fashion. This allows Salt module developers to package dependencies with their modules for ease of deployment, isolation, etc.

For a user, Zip Archive modules behave just like other modules. When executing a function from a module provided as the file `my_module.zip`, a user would call a function within that module as `my_module.<function>`.

Creating a Zip Archive Module

A Zip Archive module is structured similarly to a simple Python package. The `.zip` file contains a single directory with the same name as the module. The module code traditionally in `<module_name>.py` goes in `<module_name>/__init__.py`. The dependency packages are subdirectories of `<module_name>/`.

Here is an example directory structure for the `lumberjack` module, which has two library dependencies (`sleep` and `work`) to be included.

```
modules $ ls -R lumberjack
__init__.py     sleep           work

lumberjack/sleep:
__init__.py

lumberjack/work:
__init__.py
```

The contents of `lumberjack/__init__.py` show how to import and use these included libraries.

```
# Libraries included in lumberjack.zip
from lumberjack import sleep, work


def is_ok(person):
    ''' Checks whether a person is really a lumberjack '''
    return sleep.all_night(person) and work.all_day(person)
```

Then, create the zip:

```
modules $ zip -r lumberjack lumberjack
  adding: lumberjack/ (stored 0%)
  adding: lumberjack/__init__.py (deflated 39%)
  adding: lumberjack/sleep/ (stored 0%)
  adding: lumberjack/sleep/__init__.py (deflated 7%)
  adding: lumberjack/work/ (stored 0%)
  adding: lumberjack/work/__init__.py (deflated 7%)
```

```
modules $ unzip -l lumberjack.zip
Archive:  lumberjack.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
        0  08-21-15 20:08   lumberjack/
      348  08-21-15 20:08   lumberjack/__init__.py
        0  08-21-15 19:53   lumberjack/sleep/
       83  08-21-15 19:53   lumberjack/sleep/__init__.py
        0  08-21-15 19:53   lumberjack/work/
       81  08-21-15 19:21   lumberjack/work/__init__.py
 --------                   -------
      512                   6 files
```

Once placed in *file_roots*, Salt users can distribute and use `lumberjack.zip` like any other module.

```
$ sudo salt minion1 saltutil.sync_modules
minion1:
  - modules.lumberjack
$ sudo salt minion1 lumberjack.is_ok 'Michael Palin'
minion1:
  True
```

5.2.3 Cross Calling Execution Modules

All of the Salt execution modules are available to each other and modules can call functions available in other execution modules.

The variable `__salt__` is packed into the modules after they are loaded into the Salt minion.

The `__salt__` variable is a Python dictionary containing all of the Salt functions. Dictionary keys are strings representing the names of the modules and the values are the functions themselves.

Salt modules can be cross-called by accessing the value in the `__salt__` dict:

```
def foo(bar):
    return __salt__['cmd.run'](bar)
```

This code will call the *run* function in the *cmd* module and pass the argument `bar` to it.

5.2.4 Calling Execution Modules on the Salt Master

New in version 2016.11.0.

Execution modules can now also be called via the **salt-run** command using the *salt runner*.

5.2.5 Preloaded Execution Module Data

When interacting with execution modules often it is nice to be able to read information dynamically about the minion or to load in configuration parameters for a module.

Salt allows for different types of data to be loaded into the modules by the minion.

Grains Data

The values detected by the Salt Grains on the minion are available in a Python dictionary named `__grains__` and can be accessed from within callable objects in the Python modules.

To see the contents of the grains dictionary for a given system in your deployment run the `grains.items()` function:

```
salt 'hostname' grains.items --output=pprint
```

Any value in a grains dictionary can be accessed as any other Python dictionary. For example, the grain representing the minion ID is stored in the `id` key and from an execution module, the value would be stored in `__grains__['id']`.

Module Configuration

Since parameters for configuring a module may be desired, Salt allows for configuration information from the minion configuration file to be passed to execution modules.

Since the minion configuration file is a YAML document, arbitrary configuration data can be passed in the minion config that is read by the modules. It is therefore **strongly** recommended that the values passed in the configuration file match the module name. A value intended for the `test` execution module should be named `test.<value>`.

The test execution module contains usage of the module configuration and the default configuration file for the minion contains the information and format used to pass data to the modules. *salt.modules.test*, `conf/minion`.

`__init__` Function

If you want your module to have different execution modes based on minion configuration, you can use the `__init__(opts)` function to perform initial module setup. The parameter `opts` is the complete minion configuration, as also available in the `__opts__` dict.

```
'''
Cheese module initialization example
'''
def __init__(opts):
    '''
    Allow foreign imports if configured to do so
    '''
    if opts.get('cheese.allow_foreign', False):
        _enable_foreign_products()
```

5.2.6 Strings and Unicode

An execution module author should always assume that strings fed to the module have already decoded from strings into Unicode. In Python 2, these will be of type `Unicode' and in Python 3 they will be of type `str`. Calling from a state to other Salt sub-systems, should pass Unicode (or bytes if passing binary data). In the rare event that a state needs to write directly to disk, Unicode should be encoded to a string immediately before writing to disk. An author may use `__salt_system_encoding__` to learn what the encoding type of the system is. For example, *`my_string'.encode(__salt_system_encoding__')*.

5.2.7 Outputter Configuration

Since execution module functions can return different data, and the way the data is printed can greatly change the presentation, Salt allows for a specific outputter to be set on a function-by-function basis.

This is done be declaring an `__outputter__` dictionary in the global scope of the module. The `__outputter__` dictionary contains a mapping of function names to Salt *outputters*.

```
__outputter__ = {
    'run': 'txt'
}
```

This will ensure that the `txt` outputter is used to display output from the `run` function.

5.2.8 Virtual Modules

Virtual modules let you override the name of a module in order to use the same name to refer to one of several similar modules. The specific module that is loaded for a virtual name is selected based on the current platform or environment.

For example, packages are managed across platforms using the `pkg` module. `pkg` is a virtual module name that is an alias for the specific package manager module that is loaded on a specific system (for example, *yumpkg* on RHEL/CentOS systems , and *aptpkg* on Ubuntu).

Virtual module names are set using the `__virtual__` function and the *virtual name*.

5.2.9 `__virtual__` Function

The `__virtual__` function returns either a string, `True`, `False`, or `False` with an *error string*. If a string is returned then the module is loaded using the name of the string as the virtual name. If `True` is returned the module is loaded using the current module name. If `False` is returned the module is not loaded. `False` lets the module perform system checks and prevent loading if dependencies are not met.

Since `__virtual__` is called before the module is loaded, `__salt__` will be unreliable as not all modules will be available at this point in time. The `__pillar` and `__grains__` *``dunder'' dictionaries* are available however.

Note: Modules which return a string from `__virtual__` that is already used by a module that ships with Salt will _override_ the stock module.

Returning Error Information from `__virtual__`

Optionally, Salt plugin modules, such as execution, state, returner, beacon, etc. modules may additionally return a string containing the reason that a module could not be loaded. For example, an execution module called `cheese` and a corresponding state module also called `cheese`, both depending on a utility called `enzymes` should have `__virtual__` functions that handle the case when the dependency is unavailable.

```
'''
Cheese execution (or returner/beacon/etc.) module
'''
try:
    import enzymes
    HAS_ENZYMES = True
except ImportError:
```

```
    HAS_ENZYMES = False


def __virtual__():
    '''
    only load cheese if enzymes are available
    '''
    if HAS_ENZYMES:
        return 'cheese'
    else:
        return False, 'The cheese execution module cannot be loaded: enzymes␣
↪unavailable.'

def slice():
    pass
```

```
'''
Cheese state module. Note that this works in state modules because it is
guaranteed that execution modules are loaded first
'''

def __virtual__():
    '''
    only load cheese if enzymes are available
    '''
    # predicate loading of the cheese state on the corresponding execution module
    if 'cheese.slice' in __salt__:
        return 'cheese'
    else:
        return False, 'The cheese state module cannot be loaded: enzymes unavailable.'
```

Examples

The package manager modules are among the best examples of using the `__virtual__` function. A table of all the virtual `pkg` modules can be found *here*.

Overriding Virtual Module Providers

Salt often uses OS grains (`os`, `osrelease`, `os_family`, etc.) to determine which module should be loaded as the virtual module for `pkg`, `service`, etc. Sometimes this OS detection is incomplete, with new distros popping up, existing distros changing init systems, etc. The virtual modules likely to be affected by this are in the list below (click each item for more information):

- *pkg*
- *service*
- *user*
- *shadow*
- *group*

If Salt is using the wrong module for one of these, first of all, please report it on the issue tracker, so that this issue can be resolved for a future release. To make it easier to troubleshoot, please also provide the *grains.items* output, taking care to redact any sensitive information.

Then, while waiting for the SaltStack development team to fix the issue, Salt can be made to use the correct module using the `providers` option in the minion config file:

```
providers:
  service: systemd
  pkg: aptpkg
```

The above example will force the minion to use the *systemd* module to provide service management, and the *aptpkg* module to provide package management.

Logging Restrictions

As a rule, logging should not be done anywhere in a Salt module before it is loaded. This rule apples to all code that would run before the `__virtual__()` function, as well as the code within the `__virtual__()` function itself.

If logging statements are made before the virtual function determines if the module should be loaded, then those logging statements will be called repeatedly. This clutters up log files unnecessarily.

Exceptions may be considered for logging statements made at the `trace` level. However, it is better to provide the necessary information by another means. One method is to *return error information* in the `__virtual__()` function.

5.2.10 `__virtualname__`

`__virtualname__` is a variable that is used by the documentation build system to know the virtual name of a module without calling the `__virtual__` function. Modules that return a string from the `__virtual__` function must also set the `__virtualname__` variable.

To avoid setting the virtual name string twice, you can implement `__virtual__` to return the value set for `__virtualname__` using a pattern similar to the following:

```
# Define the module's virtual name
__virtualname__ = 'pkg'


def __virtual__():
    '''
    Confine this module to Mac OS with Homebrew.
    '''

    if salt.utils.path.which('brew') and __grains__['os'] == 'MacOS':
        return __virtualname__
    return False
```

The `__virtual__()` function can return a `True` or `False` boolean, a tuple, or a string. If it returns a `True` value, this `__virtualname__` module-level attribute can be set as seen in the above example. This is the string that the module should be referred to as.

When `__virtual__()` returns a tuple, the first item should be a boolean and the second should be a string. This is typically done when the module should not load. The first value of the tuple is `False` and the second is the error message to display for why the module did not load.

For example:

```
def __virtual__():
    '''
    Only load if git exists on the system
```

```
    '''
    if salt.utils.path.which('git') is None:
        return (False,
                'The git execution module cannot be loaded: git unavailable.')
    else:
        return True
```

5.2.11 Documentation

Salt execution modules are documented. The `sys.doc()` function will return the documentation for all available modules:

```
salt '*' sys.doc
```

The `sys.doc` function simply prints out the docstrings found in the modules; when writing Salt execution modules, please follow the formatting conventions for docstrings as they appear in the other modules.

Adding Documentation to Salt Modules

It is strongly suggested that all Salt modules have documentation added.

To add documentation add a Python docstring to the function.

```
def spam(eggs):
    '''
    A function to make some spam with eggs!

    CLI Example::

        salt '*' test.spam eggs
    '''
    return eggs
```

Now when the sys.doc call is executed the docstring will be cleanly returned to the calling terminal.

Documentation added to execution modules in docstrings will automatically be added to the online web-based documentation.

Add Execution Module Metadata

When writing a Python docstring for an execution module, add information about the module using the following field lists:

```
:maintainer:    Thomas Hatch <thatch@saltstack.com, Seth House <shouse@saltstack.com>
:maturity:      new
:depends:       python-mysqldb
:platform:      all
```

The maintainer field is a comma-delimited list of developers who help maintain this module.

The maturity field indicates the level of quality and testing for this module. Standard labels will be determined.

The depends field is a comma-delimited list of modules that this module depends on.

The platform field is a comma-delimited list of platforms that this module is known to run on.

5.2.12 Log Output

You can call the logger from custom modules to write messages to the minion logs. The following code snippet demonstrates writing log messages:

```
import logging

log = logging.getLogger(__name__)

log.info('Here is Some Information')
log.warning('You Should Not Do That')
log.error('It Is Busted')
```

5.2.13 Aliasing Functions

Sometimes one wishes to use a function name that would shadow a python built-in. A common example would be `set()`. To support this, append an underscore to the function definition, `def set_():`, and use the `__func_alias__` feature to provide an alias to the function.

`__func_alias__` is a dictionary where each key is the name of a function in the module, and each value is a string representing the alias for that function. When calling an aliased function from a different execution module, state module, or from the cli, the alias name should be used.

```
__func_alias__ = {
    'set_': 'set',
    'list_': 'list',
}
```

5.2.14 Private Functions

In Salt, Python callable objects contained within an execution module are made available to the Salt minion for use. The only exception to this rule is a callable object with a name starting with an underscore `_`.

Objects Loaded Into the Salt Minion

```
def foo(bar):
    return bar
```

Objects NOT Loaded into the Salt Minion

```
def _foobar(baz): # Preceded with an _
    return baz

cheese = {} # Not a callable Python object
```

5.2.15 Useful Decorators for Modules

Depends Decorator

When writing execution modules there are many times where some of the module will work on all hosts but some functions have an external dependency, such as a service that needs to be installed or a binary that needs to be present on the system.

Instead of trying to wrap much of the code in large try/except blocks, a decorator can be used.

If the dependencies passed to the decorator don't exist, then the salt minion will remove those functions from the module on that host.

If a `fallback_function` is defined, it will replace the function instead of removing it

```
import logging

from salt.utils.decorators import depends

log = logging.getLogger(__name__)

try:
    import dependency_that_sometimes_exists
except ImportError as e:
    log.trace('Failed to import dependency_that_sometimes_exists: {0}'.format(e))

@depends('dependency_that_sometimes_exists')
def foo():
    '''
    Function with a dependency on the "dependency_that_sometimes_exists" module,
    if the "dependency_that_sometimes_exists" is missing this function will not exist
    '''
    return True

def _fallback():
    '''
    Fallback function for the depends decorator to replace a function with
    '''
    return '"dependency_that_sometimes_exists" needs to be installed for this function
 ↪to exist'

@depends('dependency_that_sometimes_exists', fallback_function=_fallback)
def foo():
    '''
    Function with a dependency on the "dependency_that_sometimes_exists" module.
    If the "dependency_that_sometimes_exists" is missing this function will be
    replaced with "_fallback"
    '''
    return True
```

In addition to global dependencies the depends decorator also supports raw booleans.

```
from salt.utils.decorators import depends

HAS_DEP = False
try:
    import dependency_that_sometimes_exists
    HAS_DEP = True
except ImportError:
```

```
    pass

@depends(HAS_DEP)
def foo():
    return True
```

5.3 Executors

Executors are used by minion to execute module functions. Executors can be used to modify the functions behavior, do any pre-execution steps or execute in a specific way like sudo executor.

Executors could be passed as a list and they will be used one-by-one in the order. If an executor returns `None` the next one will be called. If an executor returns non-`None` the execution sequence is terminated and the returned value is used as a result. It's a way executor could control modules execution working as a filter. Note that executor could actually not execute the function but just do something else and return `None` like `splay` executor does. In this case some other executor have to be used as a final executor that will actually execute the function. See examples below.

Executors list could be passed by minion config file in the following way:

```
module_executors:
  - splay
  - direct_call
splaytime: 30
```

The same could be done by command line:

```
salt -t 40 --module-executors='[splay, direct_call]' --executor-opts='{splaytime: 30}'
↪'*' test.ping
```

And the same command called via netapi will look like this:

```
curl -sSk https://localhost:8000 \
    -H 'Accept: application/x-yaml' \
    -H 'X-Auth-Token: 697adbdc8fe971d09ae4c2a3add7248859c87079' \
    -H 'Content-type: application/json' \
    -d '[{
        "client": "local",
        "tgt": "*",
        "fun": "test.ping",
        "module_executors": ["splay", "direct_call"],
        "executor_opts": {"splaytime": 10}
        }]'
```

See also:

The full list of executors

5.3.1 Writing Salt Executors

A Salt executor is written in a similar manner to a Salt execution module. Executor is a python module placed into the `executors` folder and containing the `execute` function with the following signature:

```
def execute(opts, data, func, args, kwargs)
```

Where the args are:

opts: Dictionary containing the minion configuration options

data: Dictionary containing the load data including `executor_opts` passed via cmdline/API.

func, **args**, **kwargs**: Execution module function to be executed and it's arguments. For instance the simplest `direct_call` executor just runs it as `func(*args,**kwargs)`.

Returns: `None` if the execution sequence must be continued with the next executor. Error string or execution result if the job is done and execution must be stopped.

Specific options could be passed to the executor via minion config or via `executor_opts` argument. For instance to access `splaytime` option set by minion config executor should access `opts.get('splaytime')`. To access the option set by commandline or API `data.get('executor_opts',{}).get('splaytime')` should be used. So if an option is safe and must be accessible by user executor should check it in both places, but if an option is unsafe it should be read from the only config ignoring the passed request data.

Configuration Management

Salt contains a robust and flexible configuration management framework, which is built on the remote execution core. This framework executes on the minions, allowing effortless, simultaneous configuration of tens of thousands of hosts, by rendering language specific state files. The following links provide resources to learn more about state and renderers.

States Express the state of a host using small, easy to read, easy to understand configuration files. *No programming required.*

- ***Full list of states*** Contains: list of install packages, create users, transfer files, start services, and so on.
- ***Pillar System*** Contains: description of Salt's Pillar system.
- ***Highstate data structure*** Contains: a dry vocabulary and technical representation of the configuration format that states represent.
- ***Writing states*** Contains: a guide on how to write Salt state modules, easily extending Salt to directly manage more software.

Note: Salt execution modules are different from state modules and cannot be called as a state in an SLS file. In other words, this will not work:

```
moe:
  user.rename:
    - new_name: larry
    - onlyif: id moe
```

You must use the `module` states to call execution modules directly. Here's an example:

```
rename_moe:
  module.run:
    - m_name: moe
    - new_name: larry
    - onlyif: id moe
```

Renderers Renderers use state configuration files written in a variety of languages, templating engines, or files. Salt's configuration management system is, under the hood, language agnostic.

- ***Full list of renderers*** Contains: a list of renderers. YAML is one choice, but many systems are available, from alternative templating engines to the PyDSL language for rendering sls formulas.
- ***Renderers*** Contains: more information about renderers. Salt states are only concerned with the ultimate highstate data structure, not how the data structure was created.

6.1 State System Reference

Salt offers an interface to manage the configuration or ``state'' of the Salt minions. This interface is a fully capable mechanism used to enforce the state of systems from a central manager.

6.1.1 Mod Aggregate State Runtime Modifications

New in version 2014.7.0.

The mod_aggregate system was added in the 2014.7.0 release of Salt and allows for runtime modification of the executing state data. Simply put, it allows for the data used by Salt's state system to be changed on the fly at runtime, kind of like a configuration management JIT compiler or a runtime import system. All in all, it makes Salt much more dynamic.

How it Works

The best example is the `pkg` state. One of the major requests in Salt has long been adding the ability to install all packages defined at the same time. The mod_aggregate system makes this a reality. While executing Salt's state system, when a `pkg` state is reached the `mod_aggregate` function in the state module is called. For `pkg` this function scans all of the other states that are slated to run, and picks up the references to `name` and `pkgs`, then adds them to `pkgs` in the first state. The result is a single call to yum, apt-get, pacman, etc as part of the first package install.

How to Use it

Note: Since this option changes the basic behavior of the state runtime, after it is enabled states should be executed using *test=True* to ensure that the desired behavior is preserved.

In config files

The first way to enable aggregation is with a configuration option in either the master or minion configuration files. Salt will invoke `mod_aggregate` the first time it encounters a state module that has aggregate support.

If this option is set in the master config it will apply to all state runs on all minions, if set in the minion config it will only apply to said minion.

Enable for all states:

```
state_aggregate: True
```

Enable for only specific state modules:

```
state_aggregate:
  - pkg
```

In states

The second way to enable aggregation is with the state-level `aggregate` keyword. In this configuration, Salt will invoke the `mod_aggregate` function the first time it encounters this keyword. Any additional occurrences of the keyword will be ignored as the aggregation has already taken place.

The following example will trigger `mod_aggregate` when the `lamp_stack` state is processed resulting in a single call to the underlying package manager.

```
lamp_stack:
  pkg.installed:
    - pkgs:
      - php
      - mysql-client
    - aggregate: True

memcached:
  pkg.installed:
    - name: memcached
```

Adding mod_aggregate to a State Module

Adding a mod_aggregate routine to an existing state module only requires adding an additional function to the state module called mod_aggregate.

The mod_aggregate function just needs to accept three parameters and return the low data to use. Since mod_aggregate is working on the state runtime level it does need to manipulate *low data*.

The three parameters are *low*, *chunks*, and *running*. The *low* option is the low data for the state execution which is about to be called. The *chunks* is the list of all of the low data dictionaries which are being executed by the runtime and the *running* dictionary is the return data from all of the state executions which have already be executed.

This example, simplified from the pkg state, shows how to create mod_aggregate functions:

```
def mod_aggregate(low, chunks, running):
    '''
    The mod_aggregate function which looks up all packages in the available
    low chunks and merges them into a single pkgs ref in the present low data
    '''
    pkgs = []
    # What functions should we aggregate?
    agg_enabled = [
            'installed',
            'latest',
            'removed',
            'purged',
            ]
    # The `low` data is just a dict with the state, function (fun) and
    # arguments passed in from the sls
    if low.get('fun') not in agg_enabled:
        return low
    # Now look into what other things are set to execute
    for chunk in chunks:
        # The state runtime uses "tags" to track completed jobs, it may
        # look familiar with the _|-
        tag = __utils__['state.gen_tag'](chunk)
        if tag in running:
```

```
                # Already ran the pkg state, skip aggregation
                continue
            if chunk.get('state') == 'pkg':
                if '__agg__' in chunk:
                    continue
                # Check for the same function
                if chunk.get('fun') != low.get('fun'):
                    continue
                # Pull out the pkg names!
                if 'pkgs' in chunk:
                    pkgs.extend(chunk['pkgs'])
                    chunk['__agg__'] = True
                elif 'name' in chunk:
                    pkgs.append(chunk['name'])
                    chunk['__agg__'] = True
    if pkgs:
        if 'pkgs' in low:
            low['pkgs'].extend(pkgs)
        else:
            low['pkgs'] = pkgs
    # The low has been modified and needs to be returned to the state
    # runtime for execution
    return low
```

6.1.2 Altering States

Note: This documentation has been moved *here*.

6.1.3 File State Backups

In 0.10.2 a new feature was added for backing up files that are replaced by the file.managed and file.recurse states. The new feature is called the backup mode. Setting the backup mode is easy, but it can be set in a number of places.

The backup_mode can be set in the minion config file:

```
backup_mode: minion
```

Or it can be set for each file:

```
/etc/ssh/sshd_config:
  file.managed:
    - source: salt://ssh/sshd_config
    - backup: minion
```

Backed-up Files

The files will be saved in the minion cachedir under the directory named `file_backup`. The files will be in the location relative to where they were under the root filesystem and be appended with a timestamp. This should make them easy to browse.

Interacting with Backups

Starting with version 0.17.0, it will be possible to list, restore, and delete previously-created backups.

Listing

The backups for a given file can be listed using *file.list_backups*:

```
# salt foo.bar.com file.list_backups /tmp/foo.txt
foo.bar.com:
    ----------
    0:
        ----------
        Backup Time:
            Sat Jul 27 2013 17:48:41.738027
        Location:
            /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_17:48:41_738027_
↪2013
        Size:
            13
    1:
        ----------
        Backup Time:
            Sat Jul 27 2013 17:48:28.369804
        Location:
            /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_17:48:28_369804_
↪2013
        Size:
            35
```

Restoring

Restoring is easy using *file.restore_backup*, just pass the path and the numeric id found with *file.list_backups*:

```
# salt foo.bar.com file.restore_backup /tmp/foo.txt 1
foo.bar.com:
    ----------
    comment:
        Successfully restored /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_
↪27_17:48:28_369804_2013 to /tmp/foo.txt
    result:
        True
```

The existing file will be backed up, just in case, as can be seen if *file.list_backups* is run again:

```
# salt foo.bar.com file.list_backups /tmp/foo.txt
foo.bar.com:
    ----------
    0:
        ----------
        Backup Time:
            Sat Jul 27 2013 18:00:19.822550
        Location:
            /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_18:00:19_822550_
↪2013
```

```
        Size:
            53
    1:
        ----------
        Backup Time:
            Sat Jul 27 2013 17:48:41.738027
        Location:
            /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_17:48:41_738027_
↪2013
        Size:
            13
    2:
        ----------
        Backup Time:
            Sat Jul 27 2013 17:48:28.369804
        Location:
            /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_17:48:28_369804_
↪2013
        Size:
            35
```

Note: Since no state is being run, restoring a file will not trigger any watches for the file. So, if you are restoring a config file for a service, it will likely still be necessary to run a `service.restart`.

Deleting

Deleting backups can be done using *file.delete_backup*:

```
# salt foo.bar.com file.delete_backup /tmp/foo.txt 0
foo.bar.com:
    ----------
    comment:
        Successfully removed /var/cache/salt/minion/file_backup/tmp/foo.txt_Sat_Jul_27_
↪18:00:19_822550_2013
    result:
        True
```

6.1.4 Understanding State Compiler Ordering

Note: This tutorial is an intermediate level tutorial. Some basic understanding of the state system and writing Salt Formulas is assumed.

Salt's state system is built to deliver all of the power of configuration management systems without sacrificing simplicity. This tutorial is made to help users understand in detail just how the order is defined for state executions in Salt.

This tutorial is written to represent the behavior of Salt as of version 0.17.0.

Compiler Basics

To understand ordering in depth some very basic knowledge about the state compiler is very helpful. No need to worry though, this is very high level!

High Data and Low Data

When defining Salt Formulas in YAML the data that is being represented is referred to by the compiler as High Data. When the data is initially loaded into the compiler it is a single large python dictionary, this dictionary can be viewed raw by running:

```
salt '*' state.show_highstate
```

This ``High Data'' structure is then compiled down to ``Low Data''. The Low Data is what is matched up to create individual executions in Salt's configuration management system. The low data is an ordered list of single state calls to execute. Once the low data is compiled the evaluation order can be seen.

The low data can be viewed by running:

```
salt '*' state.show_lowstate
```

Note: The state execution module contains MANY functions for evaluating the state system and is well worth a read! These routines can be very useful when debugging states or to help deepen one's understanding of Salt's state system.

As an example, a state written thusly:

```
apache:
  pkg.installed:
    - name: httpd
  service.running:
    - name: httpd
    - watch:
      - file: apache_conf
      - pkg: apache

apache_conf:
  file.managed:
    - name: /etc/httpd/conf.d/httpd.conf
    - source: salt://apache/httpd.conf
```

Will have High Data which looks like this represented in json:

```
{
    "apache": {
        "pkg": [
            {
                "name": "httpd"
            },
            "installed",
            {
                "order": 10000
            }
        ],
        "service": [
```

```
            {
                "name": "httpd"
            },
            {
                "watch": [
                    {
                        "file": "apache_conf"
                    },
                    {
                        "pkg": "apache"
                    }
                ]
            },
            "running",
            {
                "order": 10001
            }
        ],
        "__sls__": "blah",
        "__env__": "base"
    },
    "apache_conf": {
        "file": [
            {
                "name": "/etc/httpd/conf.d/httpd.conf"
            },
            {
                "source": "salt://apache/httpd.conf"
            },
            "managed",
            {
                "order": 10002
            }
        ],
        "__sls__": "blah",
        "__env__": "base"
    }
}
```

The subsequent Low Data will look like this:

```
[
    {
        "name": "httpd",
        "state": "pkg",
        "__id__": "apache",
        "fun": "installed",
        "__env__": "base",
        "__sls__": "blah",
        "order": 10000
    },
    {
        "name": "httpd",
        "watch": [
            {
                "file": "apache_conf"
            },
            {
```

```
                "pkg": "apache"
            }
        ],
        "state": "service",
        "__id__": "apache",
        "fun": "running",
        "__env__": "base",
        "__sls__": "blah",
        "order": 10001
    },
    {
        "name": "/etc/httpd/conf.d/httpd.conf",
        "source": "salt://apache/httpd.conf",
        "state": "file",
        "__id__": "apache_conf",
        "fun": "managed",
        "__env__": "base",
        "__sls__": "blah",
        "order": 10002
    }
]
```

This tutorial discusses the Low Data evaluation and the state runtime.

Ordering Layers

Salt defines 2 order interfaces which are evaluated in the state runtime and defines these orders in a number of passes.

Definition Order

Note: The Definition Order system can be disabled by turning the option `state_auto_order` to `False` in the master configuration file.

The top level of ordering is the *Definition Order*. The *Definition Order* is the order in which states are defined in salt formulas. This is very straightforward on basic states which do not contain `include` statements or a `top` file, as the states are just ordered from the top of the file, but the include system starts to bring in some simple rules for how the *Definition Order* is defined.

Looking back at the ``Low Data'' and ``High Data'' shown above, the order key has been transparently added to the data to enable the *Definition Order*.

The Include Statement

Basically, if there is an include statement in a formula, then the formulas which are included will be run BEFORE the contents of the formula which is including them. Also, the include statement is a list, so they will be loaded in the order in which they are included.

In the following case:

`foo.sls`

```
include:
  - bar
  - baz
```

bar.sls

```
include:
  - quo
```

baz.sls

```
include:
  - qux
```

In the above case if `state.apply foo` were called then the formulas will be loaded in the following order:

1. quo
2. bar
3. qux
4. baz
5. foo

The *order* Flag

The *Definition Order* happens transparently in the background, but the ordering can be explicitly overridden using the `order` flag in states:

```
apache:
  pkg.installed:
    - name: httpd
    - order: 1
```

This order flag will over ride the definition order, this makes it very simple to create states that are always executed first, last or in specific stages, a great example is defining a number of package repositories that need to be set up before anything else, or final checks that need to be run at the end of a state run by using `order: last` or `order: -1`.

When the order flag is explicitly set the *Definition Order* system will omit setting an order for that state and directly use the order flag defined.

Lexicographical Fall-back

Salt states were written to ALWAYS execute in the same order. Before the introduction of *Definition Order* in version 0.17.0 everything was ordered lexicographically according to the name of the state, then function then id.

This is the way Salt has always ensured that states always run in the same order regardless of where they are deployed, the addition of the *Definition Order* method mealy makes this finite ordering easier to follow.

The lexicographical ordering is still applied but it only has any effect when two order statements collide. This means that if multiple states are assigned the same order number that they will fall back to lexicographical ordering to ensure that every execution still happens in a finite order.

Note: If running with `state_auto_order: False` the `order` key is not set automatically, since the Lexicographical order can be derived from other keys.

Requisite Ordering

Salt states are fully declarative, in that they are written to declare the state in which a system should be. This means that components can require that other components have been set up successfully. Unlike the other ordering systems, the *Requisite* system in Salt is evaluated at runtime.

The requisite system is also built to ensure that the ordering of execution never changes, but is always the same for a given set of states. This is accomplished by using a runtime that processes states in a completely predictable order instead of using an event loop based system like other declarative configuration management systems.

Runtime Requisite Evaluation

The requisite system is evaluated as the components are found, and the requisites are always evaluated in the same order. This explanation will be followed by an example, as the raw explanation may be a little dizzying at first as it creates a linear dependency evaluation sequence.

The ``Low Data'' is an ordered list or dictionaries, the state runtime evaluates each dictionary in the order in which they are arranged in the list. When evaluating a single dictionary it is checked for requisites, requisites are evaluated in order, `require` then `watch` then `prereq`.

Note: If using requisite in statements like require_in and watch_in these will be compiled down to require and watch statements before runtime evaluation.

Each requisite contains an ordered list of requisites, these requisites are looked up in the list of dictionaries and then executed. Once all requisites have been evaluated and executed then the requiring state can safely be run (or not run if requisites have not been met).

This means that the requisites are always evaluated in the same order, again ensuring one of the core design principals of Salt's State system to ensure that execution is always finite is intact.

Simple Runtime Evaluation Example

Given the above ``Low Data'' the states will be evaluated in the following order:

1. The pkg.installed is executed ensuring that the apache package is installed, it contains no requisites and is therefore the first defined state to execute.
2. The service.running state is evaluated but NOT executed, a watch requisite is found, therefore they are read in order, the runtime first checks for the file, sees that it has not been executed and calls for the file state to be evaluated.
3. The file state is evaluated AND executed, since it, like the pkg state does not contain any requisites.
4. The evaluation of the service state continues, it next checks the pkg requisite and sees that it is met, with all requisites met the service state is now executed.

Best Practice

The best practice in Salt is to choose a method and stick with it, official states are written using requisites for all associations since requisites create clean, traceable dependency trails and make for the most portable formulas. To accomplish something similar to how classical imperative systems function all requisites can be omitted and the `failhard` option then set to `True` in the master configuration, this will stop all state runs at the first instance of a failure.

In the end, using requisites creates very tight and fine grained states, not using requisites makes full sequence runs and while slightly easier to write, and gives much less control over the executions.

6.1.5 Extending External SLS Data

Sometimes a state defined in one SLS file will need to be modified from a separate SLS file. A good example of this is when an argument needs to be overwritten or when a service needs to watch an additional state.

The Extend Declaration

The standard way to extend is via the extend declaration. The extend declaration is a top level declaration like `include` and encapsulates ID declaration data included from other SLS files. A standard extend looks like this:

```
include:
  - http
  - ssh

extend:
  apache:
    file:
      - name: /etc/httpd/conf/httpd.conf
      - source: salt://http/httpd2.conf
  ssh-server:
    service:
      - watch:
        - file: /etc/ssh/banner

/etc/ssh/banner:
  file.managed:
    - source: salt://ssh/banner
```

A few critical things happened here, first off the SLS files that are going to be extended are included, then the extend dec is defined. Under the extend dec 2 IDs are extended, the apache ID's file state is overwritten with a new name and source. Then the ssh server is extended to watch the banner file in addition to anything it is already watching.

Extend is a Top Level Declaration

This means that `extend` can only be called once in an sls, if it is used twice then only one of the extend blocks will be read. So this is WRONG:

```
include:
  - http
  - ssh

extend:
  apache:
```

```
    file:
      - name: /etc/httpd/conf/httpd.conf
      - source: salt://http/httpd2.conf
# Second extend will overwrite the first!! Only make one
extend:
  ssh-server:
    service:
      - watch:
        - file: /etc/ssh/banner
```

The Requisite ``in'' Statement

Since one of the most common things to do when extending another SLS is to add states for a service to watch, or anything for a watcher to watch, the requisite in statement was added to 0.9.8 to make extending the watch and require lists easier. The ssh-server extend statement above could be more cleanly defined like so:

```
include:
  - ssh

/etc/ssh/banner:
  file.managed:
    - source: salt://ssh/banner
    - watch_in:
      - service: ssh-server
```

Rules to Extend By

There are a few rules to remember when extending states:

1. Always include the SLS being extended with an include declaration
2. Requisites (watch and require) are appended to, everything else is overwritten
3. extend is a top level declaration, like an ID declaration, cannot be declared twice in a single SLS
4. Many IDs can be extended under the extend declaration

6.1.6 Failhard Global Option

Normally, when a state fails Salt continues to execute the remainder of the defined states and will only refuse to execute states that require the failed state.

But the situation may exist, where you would want all state execution to stop if a single state execution fails. The capability to do this is called `failing hard`.

State Level Failhard

A single state can have a failhard set, this means that if this individual state fails that all state execution will immediately stop. This is a great thing to do if there is a state that sets up a critical config file and setting a require for each state that reads the config would be cumbersome. A good example of this would be setting up a package manager early on:

```
/etc/yum.repos.d/company.repo:
  file.managed:
    - source: salt://company/yumrepo.conf
    - user: root
    - group: root
    - mode: 644
    - order: 1
    - failhard: True
```

In this situation, the yum repo is going to be configured before other states, and if it fails to lay down the config file, than no other states will be executed.

Global Failhard

It may be desired to have failhard be applied to every state that is executed, if this is the case, then failhard can be set in the master configuration file. Setting failhard in the master configuration file will result in failing hard when any minion gathering states from the master have a state fail.

This is NOT the default behavior, normally Salt will only fail states that require a failed state.

Using the global failhard is generally not recommended, since it can result in states not being executed or even checked. It can also be confusing to see states failhard if an admin is not actively aware that the failhard has been set.

To use the global failhard set failhard: True in the master configuration file.

6.1.7 Global State Arguments

Note: This documentation has been moved *here*.

6.1.8 Highstate data structure definitions

The Salt State Tree

A state tree is a collection of `SLS` files and directories that live under the directory specified in *file_roots*.

Note: Directory names or filenames in the state tree cannot contain a period, with the exception of the period in the .sls file suffix.

Top file

The main state file that instructs minions what environment and modules to use during state execution.

Configurable via *state_top*.

See also:

A detailed description of the top file

Include declaration

Defines a list of *Module reference* strings to include in this `SLS`.

Occurs only in the top level of the SLS data structure.

Example:

```
include:
  - edit.vim
  - http.server
```

Module reference

The name of a SLS module defined by a separate SLS file and residing on the Salt Master. A module named `edit.vim` is a reference to the SLS file `salt://edit/vim.sls`.

ID declaration

Defines an individual *highstate* component. Always references a value of a dictionary containing keys referencing *State declaration* and *Requisite declaration*. Can be overridden by a *Name declaration* or a *Names declaration*.

Occurs on the top level or under the *Extend declaration*.

Must be unique across entire state tree. If the same ID declaration is used twice, only the first one matched will be used. All subsequent ID declarations with the same name will be ignored.

Note: Naming gotchas

In Salt versions earlier than 0.9.7, ID declarations containing dots would result in unpredictable output.

Extend declaration

Extends a *Name declaration* from an included `SLS module`. The keys of the extend declaration always refer to an existing *ID declaration* which have been defined in included `SLS modules`.

Occurs only in the top level and defines a dictionary.

States cannot be extended more than once in a single state run.

Extend declarations are useful for adding-to or overriding parts of a *State declaration* that is defined in another `SLS` file. In the following contrived example, the shown `mywebsite.sls` file is `include` -ing and `extend` -ing the `apache.sls` module in order to add a `watch` declaration that will restart Apache whenever the Apache configuration file, `mywebsite` changes.

```
include:
  - apache

extend:
  apache:
    service:
      - watch:
        - file: mywebsite
```

```
mywebsite:
  file.managed:
    - name: /var/www/mysite
```

See also:

watch_in and require_in

Sometimes it is more convenient to use the *watch_in* or *require_in* syntax instead of extending another SLS file.

State Requisites

State declaration

A list which contains one string defining the *Function declaration* and any number of *Function arg declaration* dictionaries.

Can, optionally, contain a number of additional components like the name override components — *name* and *names*. Can also contain *requisite declarations*.

Occurs under an *ID declaration*.

Requisite declaration

A list containing *requisite references*.

Used to build the action dependency tree. While Salt states are made to execute in a deterministic order, this order is managed by requiring and watching other Salt states.

Occurs as a list component under a *State declaration* or as a key under an *ID declaration*.

Requisite reference

A single key dictionary. The key is the name of the referenced *State declaration* and the value is the ID of the referenced *ID declaration*.

Occurs as a single index in a *Requisite declaration* list.

Function declaration

The name of the function to call within the state. A state declaration can contain only a single function declaration.

For example, the following state declaration calls the `installed` function in the `pkg` state module:

```
httpd:
  pkg.installed: []
```

The function can be declared inline with the state as a shortcut. The actual data structure is compiled to this form:

```
httpd:
  pkg:
    - installed
```

Where the function is a string in the body of the state declaration. Technically when the function is declared in dot notation the compiler converts it to be a string in the state declaration list. Note that the use of the first example more than once in an ID declaration is invalid yaml.

INVALID:

```
httpd:
  pkg.installed
  service.running
```

When passing a function without arguments and another state declaration within a single ID declaration, then the long or ``standard'' format needs to be used since otherwise it does not represent a valid data structure.

VALID:

```
httpd:
  pkg.installed: []
  service.running: []
```

Occurs as the only index in the *State declaration* list.

Function arg declaration

A single key dictionary referencing a Python type which is to be passed to the named *Function declaration* as a parameter. The type must be the data type expected by the function.

Occurs under a *Function declaration*.

For example in the following state declaration `user`, `group`, and `mode` are passed as arguments to the *managed* function in the `file` state module:

```
/etc/http/conf/http.conf:
  file.managed:
    - user: root
    - group: root
    - mode: 644
```

Name declaration

Overrides the `name` argument of a *State declaration*. If `name` is not specified the *ID declaration* satisfies the `name` argument.

The name is always a single key dictionary referencing a string.

Overriding `name` is useful for a variety of scenarios.

For example, avoiding clashing ID declarations. The following two state declarations cannot both have `/etc/motd` as the ID declaration:

```
motd_perms:
  file.managed:
    - name: /etc/motd
    - mode: 644

motd_quote:
  file.append:
```

```
    - name: /etc/motd
    - text: "Of all smells, bread; of all tastes, salt."
```

Another common reason to override `name` is if the ID declaration is long and needs to be referenced in multiple places. In the example below it is much easier to specify `mywebsite` than to specify `/etc/apache2/sites-available/mywebsite.com` multiple times:

```
mywebsite:
  file.managed:
    - name: /etc/apache2/sites-available/mywebsite.com
    - source: salt://mywebsite.com

a2ensite mywebsite.com:
  cmd.wait:
    - unless: test -L /etc/apache2/sites-enabled/mywebsite.com
    - watch:
      - file: mywebsite

apache2:
  service.running:
    - watch:
      - file: mywebsite
```

Names declaration

Expands the contents of the containing *State declaration* into multiple state declarations, each with its own name.

For example, given the following state declaration:

```
python-pkgs:
  pkg.installed:
    - names:
      - python-django
      - python-crypto
      - python-yaml
```

Once converted into the lowstate data structure the above state declaration will be expanded into the following three state declarations:

```
python-django:
  pkg.installed

python-crypto:
  pkg.installed

python-yaml:
  pkg.installed
```

Other values can be overridden during the expansion by providing an additional dictionary level.

New in version 2014.7.0.

```
ius:
  pkgrepo.managed:
    - humanname: IUS Community Packages for Enterprise Linux 6 - $basearch
    - gpgcheck: 1
```

```
    - baseurl: http://mirror.rackspace.com/ius/stable/CentOS/6/$basearch
    - gpgkey: http://dl.iuscommunity.org/pub/ius/IUS-COMMUNITY-GPG-KEY
    - names:
        - ius
        - ius-devel:
            - baseurl: http://mirror.rackspace.com/ius/development/CentOS/6/$basearch
```

Large example

Here is the layout in yaml using the names of the highdata structure components.

```
<Include Declaration>:
  - <Module Reference>
  - <Module Reference>

<Extend Declaration>:
  <ID Declaration>:
    [<overrides>]


# standard declaration

<ID Declaration>:
  <State Module>:
    - <Function>
    - <Function Arg>
    - <Function Arg>
    - <Function Arg>
    - <Name>: <name>
    - <Requisite Declaration>:
      - <Requisite Reference>
      - <Requisite Reference>


# inline function and names

<ID Declaration>:
  <State Module>.<Function>:
    - <Function Arg>
    - <Function Arg>
    - <Function Arg>
    - <Names>:
      - <name>
      - <name>
      - <name>
    - <Requisite Declaration>:
      - <Requisite Reference>
      - <Requisite Reference>


# multiple states for single id

<ID Declaration>:
  <State Module>:
    - <Function>
    - <Function Arg>
    - <Name>: <name>
```

```
    - <Requisite Declaration>:
      - <Requisite Reference>
  <State Module>:
    - <Function>
    - <Function Arg>
    - <Names>:
      - <name>
      - <name>
    - <Requisite Declaration>:
      - <Requisite Reference>
```

6.1.9 Include and Exclude

Salt SLS files can include other SLS files and exclude SLS files that have been otherwise included. This allows for an SLS file to easily extend or manipulate other SLS files.

Include

When other SLS files are included, everything defined in the included SLS file will be added to the state run. When including define a list of SLS formulas to include:

```
include:
  - http
  - libvirt
```

The include statement will include SLS formulas from the same environment that the including SLS formula is in. But the environment can be explicitly defined in the configuration to override the running environment, therefore if an SLS formula needs to be included from an external environment named ``dev'' the following syntax is used:

```
include:
  - dev: http
```

NOTE: `include` does not simply inject the states where you place it in the SLS file. If you need to guarantee order of execution, consider using requisites.

Do not use dots in SLS file names or their directories

The initial implementation of `top.sls` and *Include declaration* followed the python import model where a slash is represented as a period. This means that a SLS file with a period in the name (besides the suffix period) can not be referenced. For example, webserver_1.0.sls is not referenceable because webserver_1.0 would refer to the directory/file webserver_1/0.sls

The same applies for any subdirectories, this is especially `tricky' when git repos are created. Another command that typically can't render it's output is ``state.show_sls`` of a file in a path that contains a dot.

Relative Include

In Salt 0.16.0, the capability to include SLS formulas which are relative to the running SLS formula was added. Simply precede the formula name with a `.`:

```
include:
  - .virt
  - .virt.hyper
```

In Salt 2015.8, the ability to include SLS formulas which are relative to the parents of the running SLS formula was added. In order to achieve this, precede the formula name with more than one `.` (dot). Much like Python's relative import abilities, two or more leading dots represent a relative include of the parent or parents of the current package, with each `.` representing one level after the first.

The following SLS configuration, if placed within `example.dev.virtual`, would result in `example.http` and `base` being included respectively:

```
include:
  - ..http
  - ...base
```

Exclude

The exclude statement, added in Salt 0.10.3, allows an SLS to hard exclude another SLS file or a specific id. The component is excluded after the high data has been compiled, so nothing should be able to override an exclude.

Since the exclude can remove an id or an sls the type of component to exclude needs to be defined. An exclude statement that verifies that the running *highstate* does not contain the `http` sls and the `/etc/vimrc` id would look like this:

```
exclude:
  - sls: http
  - id: /etc/vimrc
```

Note: The current state processing flow checks for duplicate IDs before processing excludes. An error occurs if duplicate IDs are present even if one of the IDs is targeted by an `exclude`.

6.1.10 State System Layers

The Salt state system is comprised of multiple layers. While using Salt does not require an understanding of the state layers, a deeper understanding of how Salt compiles and manages states can be very beneficial.

Function Call

The lowest layer of functionality in the state system is the direct state function call. State executions are executions of single state functions at the core. These individual functions are defined in state modules and can be called directly via the `state.single` command.

```
salt '*' state.single pkg.installed name='vim'
```

Low Chunk

The low chunk is the bottom of the Salt state compiler. This is a data representation of a single function call. The low chunk is sent to the state caller and used to execute a single state function.

A single low chunk can be executed manually via the `state.low` command.

```
salt '*' state.low '{name: vim, state: pkg, fun: installed}'
```

The passed data reflects what the state execution system gets after compiling the data down from sls formulas.

Low State

The *Low State* layer is the list of low chunks ``evaluated'' in order. To see what the low state looks like for a *highstate*, run:

```
salt '*' state.show_lowstate
```

This will display the raw lowstate in the order which each low chunk will be evaluated. The order of evaluation is not necessarily the order of execution, since requisites are evaluated at runtime. Requisite execution and evaluation is finite; this means that the order of execution can be ascertained with 100% certainty based on the order of the low state.

High Data

High data is the data structure represented in YAML via SLS files. The High data structure is created by merging the data components rendered inside sls files (or other render systems). The High data can be easily viewed by executing the `state.show_highstate` or `state.show_sls` functions. Since this data is a somewhat complex data structure, it may be easier to read using the json, yaml, or pprint outputters:

```
salt '*' state.show_highstate --out yaml
salt '*' state.show_sls edit.vim --out pprint
```

SLS

Above ``High Data'', the logical layers are no longer technically required to be executed, or to be executed in a hierarchy. This means that how the High data is generated is optional and very flexible. The SLS layer allows for many mechanisms to be used to render sls data from files or to use the fileserver backend to generate sls and file data from external systems.

The SLS layer can be called directly to execute individual sls formulas.

Note: SLS Formulas have historically been called ``SLS files''. This is because a single SLS was only constituted in a single file. Now the term ``SLS Formula'' better expresses how a compartmentalized SLS can be expressed in a much more dynamic way by combining pillar and other sources, and the SLS can be dynamically generated.

To call a single SLS formula named `edit.vim`, execute `state.apply` and pass `edit.vim` as an argument:

```
salt '*' state.apply edit.vim
```

HighState

Calling SLS directly logically assigns what states should be executed from the context of the calling minion. The Highstate layer is used to allow for full contextual assignment of what is executed where to be tied to groups of, or individual, minions entirely from the master. This means that the environment of a minion, and all associated execution data pertinent to said minion, can be assigned from the master without needing to execute or configure

anything on the target minion. This also means that the minion can independently retrieve information about its complete configuration from the master.

To execute the *highstate* use `state.apply`:

```
salt '*' state.apply
```

Orchestrate

The orchestrate layer expresses the highest functional layer of Salt's automated logic systems. The Overstate allows for stateful and functional orchestration of routines from the master. The orchestrate defines in data execution stages which minions should execute states, or functions, and in what order using requisite logic.

6.1.11 The Orchestrate Runner

Note: This documentation has been moved *here*.

6.1.12 Ordering States

The way in which configuration management systems are executed is a hotly debated topic in the configuration management world. Two major philosophies exist on the subject, to either execute in an imperative fashion where things are executed in the order in which they are defined, or in a declarative fashion where dependencies need to be mapped between objects.

Imperative ordering is finite and generally considered easier to write, but declarative ordering is much more powerful and flexible but generally considered more difficult to create.

Salt has been created to get the best of both worlds. States are evaluated in a finite order, which guarantees that states are always executed in the same order, and the states runtime is declarative, making Salt fully aware of dependencies via the *requisite* system.

State Auto Ordering

Salt always executes states in a finite manner, meaning that they will always execute in the same order regardless of the system that is executing them. But in Salt 0.17.0, the `state_auto_order` option was added. This option makes states get evaluated in the order in which they are defined in sls files, including the top.sls file.

The evaluation order makes it easy to know what order the states will be executed in, but it is important to note that the requisite system will override the ordering defined in the files, and the `order` option described below will also override the order in which states are defined in sls files.

If the classic ordering is preferred (lexicographic), then set `state_auto_order` to `False` in the master configuration file. Otherwise, `state_auto_order` defaults to `True`.

Requisite Statements

Note: The behavior of requisites changed in version 0.9.7 of Salt. This documentation applies to requisites in version 0.9.7 and later.

Often when setting up states any single action will require or depend on another action. Salt allows for the building of relationships between states with requisite statements. A requisite statement ensures that the named state is evaluated before the state requiring it. There are three types of requisite statements in Salt, **require**, **watch**, and **prereq**.

These requisite statements are applied to a specific state declaration:

```
httpd:
  pkg.installed: []
  file.managed:
    - name: /etc/httpd/conf/httpd.conf
    - source: salt://httpd/httpd.conf
    - require:
      - pkg: httpd
```

In this example, the **require** requisite is used to declare that the file /etc/httpd/conf/httpd.conf should only be set up if the pkg state executes successfully.

The requisite system works by finding the states that are required and executing them before the state that requires them. Then the required states can be evaluated to see if they have executed correctly.

Require statements can refer to any state defined in Salt. The basic examples are *pkg*, *service*, and *file*, but any used state can be referenced.

In addition to state declarations such as pkg, file, etc., **sls** type requisites are also recognized, and essentially allow `chaining' of states. This provides a mechanism to ensure the proper sequence for complex state formulas, especially when the discrete states are split or groups into separate sls files:

```
include:
  - network

httpd:
  pkg.installed: []
  service.running:
    - require:
      - pkg: httpd
      - sls: network
```

In this example, the httpd service running state will not be applied (i.e., the httpd service will not be started) unless both the httpd package is installed AND the network state is satisfied.

Note: Requisite matching

Requisites match on both the ID Declaration and the `name` parameter. Therefore, if using the `pkgs` or `sources` argument to install a list of packages in a pkg state, it's important to note that it is impossible to match an individual package in the list, since all packages are installed as a single state.

Multiple Requisites

The requisite statement is passed as a list, allowing for the easy addition of more requisites. Both requisite types can also be separately declared:

```
httpd:
  pkg.installed: []
  service.running:
    - enable: True
```

```
    - watch:
      - file: /etc/httpd/conf/httpd.conf
    - require:
      - pkg: httpd
      - user: httpd
      - group: httpd
  file.managed:
    - name: /etc/httpd/conf/httpd.conf
    - source: salt://httpd/httpd.conf
    - require:
      - pkg: httpd
  user.present: []
  group.present: []
```

In this example, the httpd service is only going to be started if the package, user, group, and file are executed successfully.

Requisite Documentation

For detailed information on each of the individual requisites, *please look here.*

The Order Option

Before using the *order* option, remember that the majority of state ordering should be done with a *Requisite declaration*, and that a requisite declaration will override an *order* option, so a state with order option should not require or required by other states.

The order option is used by adding an order number to a state declaration with the option *order*:

```
vim:
  pkg.installed:
    - order: 1
```

By adding the order option to *1* this ensures that the vim package will be installed in tandem with any other state declaration set to the order *1*.

Any state declared without an order option will be executed after all states with order options are executed.

But this construct can only handle ordering states from the beginning. Certain circumstances will present a situation where it is desirable to send a state to the end of the line. To do this, set the order to `last`:

```
vim:
  pkg.installed:
    - order: last
```

6.1.13 Running States in Parallel

Introduced in Salt version `2017.7.0` it is now possible to run select states in parallel. This is accomplished very easily by adding the `parallel: True` option to your state declaration:

```
nginx:
  service.running:
    - parallel: True
```

Now `nginx` will be started in a separate process from the normal state run and will therefore not block additional states.

Parallel States and Requisites

Parallel States still honor requisites. If a given state requires another state that has been run in parallel then the state runtime will wait for the required state to finish.

Given this example:

```
sleep 10:
  cmd.run:
    - parallel: True

nginx:
  service.running:
    - parallel: True
    - require:
      - cmd: sleep 10

sleep 5:
  cmd.run:
    - parallel: True
```

The `sleep 10` will be started first, then the state system will block on starting nginx until the `sleep 10` completes. Once nginx has been ensured to be running then the `sleep 5` will start.

This means that the order of evaluation of Salt States and requisites are still honored, and given that in the above case, `parallel: True` does not actually speed things up.

To run the above state much faster make sure that the `sleep 5` is evaluated before the `nginx` state

```
sleep 10:
  cmd.run:
    - parallel: True

sleep 5:
  cmd.run:
    - parallel: True

nginx:
  service.running:
    - parallel: True
    - require:
      - cmd: sleep 10
```

Now both of the sleep calls will be started in parallel and `nginx` will still wait for the state it requires, but while it waits the `sleep 5` state will also complete.

Things to be Careful of

Parallel States do not prevent you from creating parallel conflicts on your system. This means that if you start multiple package installs using Salt then the package manager will block or fail. If you attempt to manage the same file with multiple states in parallel then the result can produce an unexpected file.

Make sure that the states you choose to run in parallel do not conflict, or else, like in any parallel programming environment, the outcome may not be what you expect. Doing things like just making all states run in parallel will

almost certainly result in unexpected behavior.

With that said, running states in parallel should be safe the vast majority of the time and the most likely culprit for unexpected behavior is running multiple package installs in parallel.

6.1.14 State Providers

New in version 0.9.8.

Salt predetermines what modules should be mapped to what uses based on the properties of a system. These determinations are generally made for modules that provide things like package and service management.

Sometimes in states, it may be necessary to use an alternative module to provide the needed functionality. For instance, an very old Arch Linux system may not be running systemd, so instead of using the systemd service module, you can revert to the default service module:

```
httpd:
  service.running:
    - enable: True
    - provider: service
```

In this instance, the basic `service` module (which manages **sysvinit**-based services) will replace the `systemd` module which is used by default on Arch Linux.

This change only affects this one state though. If it is necessary to make this override for most or every service, it is better to just override the provider in the minion config file, as described *here*.

Also, keep in mind that this only works for states with an identically-named virtual module (`pkg`, `service`, etc.).

Arbitrary Module Redirects

The provider statement can also be used for more powerful means, instead of overwriting or extending the module used for the named service an arbitrary module can be used to provide certain functionality.

```
emacs:
  pkg.installed:
    - provider:
      - cmd: customcmd
```

In this example, the state is being instructed to use a custom module to invoke commands.

Arbitrary module redirects can be used to dramatically change the behavior of a given state.

6.1.15 Requisites and Other Global State Arguments

Requisites

The Salt requisite system is used to create relationships between states. The core idea being that, when one state is dependent somehow on another, that inter-dependency can be easily defined. These dependencies are expressed by declaring the relationships using state names and ID's or names. The generalized form of a requisite target is `<state name> : <ID or name>`. The specific form is defined as a *Requisite Reference*

Requisites come in two types: Direct requisites (such as `require`), and requisite_ins (such as `require_in`). The relationships are directional: a direct requisite requires something from another state. However, a requisite_in inserts a requisite into the targeted state pointing to the targeting state. The following example demonstrates a direct requisite:

```
vim:
  pkg.installed

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
    - require:
      - pkg: vim
```

In the example above, the file `/etc/vimrc` depends on the vim package.

Requisite_in statements are the opposite. Instead of saying ``I depend on something'', requisite_ins say ``Someone depends on me'':

```
vim:
  pkg.installed:
    - require_in:
      - file: /etc/vimrc

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
```

So here, with a requisite_in, the same thing is accomplished as in the first example, but the other way around. The vim package is saying ``/etc/vimrc depends on me''. This will result in a `require` being inserted into the `/etc/vimrc` state which targets the `vim` state.

In the end, a single dependency map is created and everything is executed in a finite and predictable order.

Requisite matching

Requisites need two pieces of information for matching: The state module name – e.g. `pkg` –, and the identifier – e.g. vim –, which can be either the ID (the first line in the stanza) or the `-name` parameter.

```
- require:
  - pkg: vim
```

Omitting state module in requisites

New in version 2016.3.0.

In version 2016.3.0, the state module name was made optional. If the state module is omitted, all states matching the ID will be required, regardless of which module they are using.

```
- require:
  - vim
```

State target matching

In order to understand how state targets are matched, it is helpful to know *how the state compiler is working*. Consider the following example:

```
Deploy server package:
  file.managed:
    - name: /usr/local/share/myapp.tar.xz
    - source: salt://myapp.tar.xz

Extract server package:
  archive.extracted:
    - name: /usr/local/share/myapp
    - source: /usr/local/share/myapp.tar.xz
    - archive_format: tar
    - onchanges:
      - file: Deploy server package
```

The first formula is converted to a dictionary which looks as follows (represented as YAML, some properties omitted for simplicity) as *High Data*:

```
Deploy server package:
  file:
    - managed
    - name: /usr/local/share/myapp.tar.xz
    - source: salt://myapp.tar.xz
```

The `file.managed` format used in the formula is essentially syntactic sugar: at the end, the target is `file`, which is used in the `Extract server package` state above.

Identifier matching

Requisites match on both the ID Declaration and the `name` parameter. This means that, in the ``Deploy server package'' example above, a `require` requisite would match with `Deploy server package` *or* `/usr/local/share/myapp.tar.xz`, so either of the following versions for ``Extract server package'' works:

```
# (Archive arguments omitted for simplicity)

# Match by ID declaration
Extract server package:
  archive.extracted:
    - onchanges:
      - file: Deploy server package

# Match by name parameter
Extract server package:
  archive.extracted:
    - onchanges:
      - file: /usr/local/share/myapp.tar.xz
```

Requisite overview

requisite

name of

requisite

result isstate is only executed if target execution

result is

changesstate is only executed if target has

changes

1.target 2.state (default)order

1.target 2.state (default)

description

comment or

description

---

require success default state will always execute unless target fails

---

watch success default like require, but adds additional behaviour (mod_watch)

---

prereq success has changes (run individually as dry-run) switched like onchanges, except order

---

onchanges success has changes default execute state if target execution result is success and target has changes

---

onfail failed default Only requisite where state exec. if target fails

---

In this table, the following short form of terms is used:

- **state** (= dependent state): state containing requisite
- **target** (= state target) : state referenced by requisite

Direct Requisite and Requisite_in types

There are several direct requisite statements that can be used in Salt:

- `require`
- `watch`
- `prereq`
- `use`
- `onchanges`
- `onfail`

Each direct requisite also has a corresponding requisite_in:

- `require_in`
- `watch_in`
- `prereq_in`
- `use_in`
- `onchanges_in`
- `onfail_in`

There are several corresponding requisite_any statements:

- require_any
- watch_any
- onchanges_any
- onfail_any

All of the requisites define specific relationships and always work with the dependency logic defined above.

require

The use of `require` demands that the required state executes before the dependent state. The state containing the `require` requisite is defined as the dependent state. The state specified in the `require` statement is defined as the required state. If the required state's execution succeeds, the dependent state will then execute. If the required state's execution fails, the dependent state will not execute. In the first example above, the file `/etc/vimrc` will only execute after the vim package is installed successfully.

Require an Entire SLS File

As of Salt 0.16.0, it is possible to require an entire sls file. Do this first by including the sls file and then setting a state to `require` the included sls file:

```
include:
  - foo

bar:
  pkg.installed:
    - require:
      - sls: foo
```

This will add all of the state declarations found in the given sls file. This means that every state in sls *foo* will be required. This makes it very easy to batch large groups of states easily in any requisite statement.

require_any

New in version 2018.3.0.

The use of `require_any` demands that one of the required states executes before the dependent state. The state containing the `require_any` requisite is defined as the dependent state. The states specified in the `require_any` statement are defined as the required states. If at least one of the required state's execution succeeds, the dependent state will then execute. If all of the executions by the required states fail, the dependent state will not execute.

```
A:
  cmd.run:
    - name: echo A
    - require_any:
      - cmd: B
      - cmd: C
      - cmd: D
B:
  cmd.run:
    - name: echo B
```

```
C:
  cmd.run:
    - name: /bin/false

D:
  cmd.run:
    - name: echo D
```

In this example *A* will run because at least one of the requirements specified, *B*, *C*, or *D* will succeed.

watch

`watch` statements are used to add additional behavior when there are changes in other states.

Note: If a state should only execute when another state has changes, and otherwise do nothing, the new `onchanges` requisite should be used instead of `watch`. `watch` is designed to add *additional* behavior when there are changes, but otherwise the state executes normally.

The state containing the `watch` requisite is defined as the watching state. The state specified in the `watch` statement is defined as the watched state. When the watched state executes, it will return a dictionary containing a key named ``changes''. Here are two examples of state return dictionaries, shown in json for clarity:

```
{
    "local": {
        "file_|-/tmp/foo_|-/tmp/foo_|-directory": {
            "comment": "Directory /tmp/foo updated",
            "__run_num__": 0,
            "changes": {
                "user": "bar"
            },
            "name": "/tmp/foo",
            "result": true
        }
    }
}

{
    "local": {
        "pkgrepo_|-salt-minion_|-salt-minion_|-managed": {
            "comment": "Package repo 'salt-minion' already configured",
            "__run_num__": 0,
            "changes": {},
            "name": "salt-minion",
            "result": true
        }
    }
}
```

If the ``result'' of the watched state is `True`, the watching state *will execute normally*, and if it is `False`, the watching state will never run. This part of `watch` mirrors the functionality of the `require` requisite.

If the ``result'' of the watched state is `True` *and* the ``changes'' key contains a populated dictionary (changes occurred in the watched state), then the `watch` requisite can add additional behavior. This additional behavior is defined by the `mod_watch` function within the watching state module. If the `mod_watch` function exists in the watching state module, it will be called *in addition to* the normal watching state. The return data from the

mod_watch function is what will be returned to the master in this case; the return data from the main watching function is discarded.

If the ``changes'' key contains an empty dictionary, the watch requisite acts exactly like the require requisite (the watching state will execute if ``result'' is True, and fail if ``result'' is False in the watched state).

Note: Not all state modules contain mod_watch. If mod_watch is absent from the watching state module, the watch requisite behaves exactly like a require requisite.

A good example of using watch is with a *service.running* state. When a service watches a state, then the service is reloaded/restarted when the watched state changes, in addition to Salt ensuring that the service is running.

```
ntpd:
  service.running:
    - watch:
      - file: /etc/ntp.conf
  file.managed:
    - name: /etc/ntp.conf
    - source: salt://ntp/files/ntp.conf
```

watch_any

New in version 2018.3.0.

The state containing the watch_any requisite is defined as the watching state. The states specified in the watch_any statement are defined as the watched states. When the watched states execute, they will return a dictionary containing a key named ``changes''.

If the ``result'' of any of the watched states is True, the watching state *will execute normally*, and if all of them are False, the watching state will never run. This part of watch mirrors the functionality of the require requisite.

If the ``result'' of any of the watched states is True *and* the ``changes'' key contains a populated dictionary (changes occurred in the watched state), then the watch requisite can add additional behavior. This additional behavior is defined by the mod_watch function within the watching state module. If the mod_watch function exists in the watching state module, it will be called *in addition to* the normal watching state. The return data from the mod_watch function is what will be returned to the master in this case; the return data from the main watching function is discarded.

If the ``changes'' key contains an empty dictionary, the watch requisite acts exactly like the require requisite (the watching state will execute if ``result'' is True, and fail if ``result'' is False in the watched state).

```
apache2:
  service.running:
    - watch_any:
      - file: /etc/apache2/sites-available/site1.conf
      - file: apache2-site2
  file.managed:
    - name: /etc/apache2/sites-available/site1.conf
    - source: salt://apache2/files/site1.conf
apache2-site2:
  file.managed:
    - name: /etc/apache2/sites-available/site2.conf
    - source: salt://apache2/files/site2.conf
```

In this example, the service will be reloaded/restarted if either of the file.managed states has a result of True and has changes.

prereq

New in version 0.16.0.

`prereq` allows for actions to be taken based on the expected results of a state that has not yet been executed. The state containing the `prereq` requisite is defined as the pre-requiring state. The state specified in the `prereq` statement is defined as the pre-required state.

When a `prereq` requisite is evaluated, the pre-required state reports if it expects to have any changes. It does this by running the pre-required single state as a test-run by enabling `test=True`. This test-run will return a dictionary containing a key named ``changes''. (See the `watch` section above for examples of ``changes'' dictionaries.)

If the ``changes'' key contains a populated dictionary, it means that the pre-required state expects changes to occur when the state is actually executed, as opposed to the test-run. The pre-requiring state will now actually run. If the pre-requiring state executes successfully, the pre-required state will then execute. If the pre-requiring state fails, the pre-required state will not execute.

If the ``changes'' key contains an empty dictionary, this means that changes are not expected by the pre-required state. Neither the pre-required state nor the pre-requiring state will run.

The best way to define how `prereq` operates is displayed in the following practical example: When a service should be shut down because underlying code is going to change, the service should be off-line while the update occurs. In this example, `graceful-down` is the pre-requiring state and `site-code` is the pre-required state.

```
graceful-down:
  cmd.run:
    - name: service apache graceful
    - prereq:
      - file: site-code

site-code:
  file.recurse:
    - name: /opt/site_code
    - source: salt://site/code
```

In this case the apache server will only be shutdown if the site-code state expects to deploy fresh code via the file.recurse call. The site-code deployment will only be executed if the graceful-down run completes successfully.

onfail

New in version 2014.7.0.

The `onfail` requisite allows for reactions to happen strictly as a response to the failure of another state. This can be used in a number of ways, such as executing a second attempt to set up a service or begin to execute a separate thread of states because of a failure.

The `onfail` requisite is applied in the same way as `require` as `watch`:

```
primary_mount:
  mount.mounted:
    - name: /mnt/share
    - device: 10.0.0.45:/share
    - fstype: nfs

backup_mount:
  mount.mounted:
    - name: /mnt/share
    - device: 192.168.40.34:/share
```

```
    - fstype: nfs
    - onfail:
      - mount: primary_mount
```

Note: Beginning in the 2016.11.0 release of Salt, onfail uses OR logic for multiple listed onfail requisites. Prior to the 2016.11.0 release, onfail used AND logic. See Issue #22370 for more information.

onfail_any

New in version 2018.3.0.

The onfail_any requisite allows for reactions to happen strictly as a response to the failure of at least one other state. This can be used in a number of ways, such as executing a second attempt to set up a service or begin to execute a separate thread of states because of a failure.

The onfail_any requisite is applied in the same way as require_any and watch_any:

```
primary_mount:
  mount.mounted:
    - name: /mnt/share
    - device: 10.0.0.45:/share
    - fstype: nfs

secondary_mount:
  mount.mounted:
    - name: /mnt/code
    - device: 10.0.0.45:/code
    - fstype: nfs

backup_mount:
  mount.mounted:
    - name: /mnt/share
    - device: 192.168.40.34:/share
    - fstype: nfs
    - onfail_any:
      - mount: primary_mount
      - mount: secondary_mount
```

In this example, the *backup_mount* will be mounted if either of the *primary_mount* or *secondary_mount* states results in a failure.

onchanges

New in version 2014.7.0.

The onchanges requisite makes a state only apply if the required states generate changes, and if the watched state's ``result'' is True. This can be a useful way to execute a post hook after changing aspects of a system.

If a state has multiple onchanges requisites then the state will trigger if any of the watched states changes.

Note: One easy-to-make mistake is to use onchanges_in when onchanges is supposed to be used. For example, the below configuration is not correct:

```
myservice:
  pkg.installed:
    - name: myservice
  file.managed:
    - name: /etc/myservice/myservice.conf
    - source: salt://myservice/files/myservice.conf
    - mode: 600
  cmd.run:
    - name: /usr/libexec/myservice/post-changes-hook.sh
    - onchanges_in:
      - file: /etc/myservice/myservice.conf
```

This will set up a requisite relationship in which the `cmd.run` state always executes, and the `file.managed` state only executes if the `cmd.run` state has changes (which it always will, since the `cmd.run` state includes the command results as changes).

It may semantically seem like the `cmd.run` state should only run when there are changes in the file state, but remember that requisite relationships involve one state watching another state, and a *requisite_in* does the opposite: it forces the specified state to watch the state with the `requisite_in`.

The correct usage would be:

```
myservice:
  pkg.installed:
    - name: myservice
  file.managed:
    - name: /etc/myservice/myservice.conf
    - source: salt://myservice/files/myservice.conf
    - mode: 600
  cmd.run:
    - name: /usr/libexec/myservice/post-changes-hook.sh
    - onchanges:
      - file: /etc/myservice/myservice.conf
```

---

onchanges_any

New in version 2018.3.0.

The `onchanges_any` requisite makes a state only apply one of the required states generates changes, and if one of the watched state's ``result'' is `True`. This can be a useful way to execute a post hook after changing aspects of a system.

```
myservice:
  pkg.installed:
    - name: myservice
    - name: yourservice
  file.managed:
    - name: /etc/myservice/myservice.conf
    - source: salt://myservice/files/myservice.conf
    - mode: 600
  file.managed:
    - name: /etc/yourservice/yourservice.conf
    - source: salt://yourservice/files/yourservice.conf
    - mode: 600
  cmd.run:
    - name: /usr/libexec/myservice/post-changes-hook.sh
```

```
    - onchanges_any:
      - file: /etc/myservice/myservice.conf
      - file: /etc/your_service/yourservice.conf
```

In this example, the *cmd.run* would be run only if either of the *file.managed* states generated changes and at least one of the watched state's ``result'' is `True`.

use

The `use` requisite is used to inherit the arguments passed in another id declaration. This is useful when many files need to have the same defaults.

```
/etc/foo.conf:
  file.managed:
    - source: salt://foo.conf
    - template: jinja
    - mkdirs: True
    - user: apache
    - group: apache
    - mode: 755

/etc/bar.conf:
  file.managed:
    - source: salt://bar.conf
    - use:
      - file: /etc/foo.conf
```

The `use` statement was developed primarily for the networking states but can be used on any states in Salt. This makes sense for the networking state because it can define a long list of options that need to be applied to multiple network interfaces.

The `use` statement does not inherit the requisites arguments of the targeted state. This means also a chain of `use` requisites would not inherit inherited options.

runas

New in version 2017.7.0.

The `runas` global option is used to set the user which will be used to run the command in the `cmd.run` module.

```
django:
  pip.installed:
    - name: django >= 1.6, <= 1.7
    - runas: daniel
    - require:
      - pkg: python-pip
```

In the above state, the pip command run by `cmd.run` will be run by the daniel user.

runas_password

New in version 2017.7.2.

The `runas_password` global option is used to set the password used by the runas global option. This is required by `cmd.run` on Windows when `runas` is specified. It will be set when `runas_password` is defined in the state.

```
run_script:
  cmd.run:
    - name: Powershell -NonInteractive -ExecutionPolicy Bypass -File C:\\Temp\\script.
↪ps1
    - runas: frank
    - runas_password: supersecret
```

In the above state, the Powershell script run by `cmd.run` will be run by the frank user with the password `supersecret`.

The _in versions of requisites

All of the requisites also have corresponding requisite_in versions, which do the reverse of their normal counterparts. The examples below all use `require_in` as the example, but note that all of the `_in` requisites work the same way: They result in a normal requisite in the targeted state, which targets the state which has defines the requisite_in. Thus, a `require_in` causes the target state to `require` the targeting state. Similarly, a `watch_in` causes the target state to `watch` the targeting state. This pattern continues for the rest of the requisites.

If a state declaration needs to be required by another state declaration then `require_in` can accommodate it. Therefore, these two sls files would be the same in the end:

Using `require`

```
httpd:
  pkg.installed: []
  service.running:
    - require:
      - pkg: httpd
```

Using `require_in`

```
httpd:
  pkg.installed:
    - require_in:
      - service: httpd
  service.running: []
```

The `require_in` statement is particularly useful when assigning a require in a separate sls file. For instance it may be common for httpd to require components used to set up PHP or mod_python, but the HTTP state does not need to be aware of the additional components that require it when it is set up:

http.sls

```
httpd:
  pkg.installed: []
  service.running:
    - require:
      - pkg: httpd
```

php.sls

```
include:
  - http
```

```
php:
  pkg.installed:
    - require_in:
      - service: httpd
```

mod_python.sls

```
include:
  - http

mod_python:
  pkg.installed:
    - require_in:
      - service: httpd
```

Now the httpd server will only start if both php and mod_python are first verified to be installed. Thus allowing for a requisite to be defined ``after the fact''.

Fire Event Notifications

New in version 2015.8.0.

The *fire_event* option in a state will cause the minion to send an event to the Salt Master upon completion of that individual state.

The following example will cause the minion to send an event to the Salt Master with a tag of *salt/state_result/20150505121517276431/dasalt/nano* and the result of the state will be the data field of the event. Notice that the *name* of the state gets added to the tag.

```
nano_stuff:
  pkg.installed:
    - name: nano
    - fire_event: True
```

In the following example instead of setting *fire_event* to *True*, *fire_event* is set to an arbitrary string, which will cause the event to be sent with this tag: *salt/state_result/20150505121725642845/dasalt/custom/tag/nano/finished*

```
nano_stuff:
  pkg.installed:
    - name: nano
    - fire_event: custom/tag/nano/finished
```

Altering States

The state altering system is used to make sure that states are evaluated exactly as the user expects. It can be used to double check that a state preformed exactly how it was expected to, or to make 100% sure that a state only runs under certain conditions. The use of unless or onlyif options help make states even more stateful. The `check_cmd` option helps ensure that the result of a state is evaluated correctly.

Reload

`reload_modules` is a boolean option that forces salt to reload its modules after a state finishes. `reload_pillar` and `reload_grains` can also be set. See *Reloading Modules*.

Unless

New in version 2014.7.0.

The `unless` requisite specifies that a state should only run when any of the specified commands return `False`. The `unless` requisite operates as NAND and is useful in giving more granular control over when a state should execute.

NOTE: Under the hood `unless` calls `cmd.retcode` with `python_shell=True`. This means the commands referenced by `unless` will be parsed by a shell, so beware of side-effects as this shell will be run with the same privileges as the salt-minion. Also be aware that the boolean value is determined by the shell's concept of `True` and `False`, rather than Python's concept of `True` and `False`.

```
vim:
  pkg.installed:
    - unless:
      - rpm -q vim-enhanced
      - ls /usr/bin/vim
```

In the example above, the state will only run if either the vim-enhanced package is not installed (returns `False`) or if /usr/bin/vim does not exist (returns `False`). The state will run if both commands return `False`.

However, the state will not run if both commands return `True`.

Unless checks are resolved for each name to which they are associated.

For example:

```
deploy_app:
  cmd.run:
    - names:
      - first_deploy_cmd
      - second_deploy_cmd
    - unless: ls /usr/bin/vim
```

In the above case, `some_check` will be run prior to _each_ name -- once for `first_deploy_cmd` and a second time for `second_deploy_cmd`.

Onlyif

New in version 2014.7.0.

The `onlyif` requisite specifies that if each command listed in `onlyif` returns `True`, then the state is run. If any of the specified commands return `False`, the state will not run.

NOTE: Under the hood `onlyif` calls `cmd.retcode` with `python_shell=True`. This means the commands referenced by `onlyif` will be parsed by a shell, so beware of side-effects as this shell will be run with the same privileges as the salt-minion. Also be aware that the boolean value is determined by the shell's concept of `True` and `False`, rather than Python's concept of `True` and `False`.

```
stop-volume:
  module.run:
    - name: glusterfs.stop_volume
    - m_name: work
    - onlyif:
      - gluster volume status work
    - order: 1
```

```
remove-volume:
  module.run:
    - name: glusterfs.delete
    - m_name: work
    - onlyif:
      - gluster volume info work
    - watch:
      - cmd: stop-volume
```

The above example ensures that the stop_volume and delete modules only run if the gluster commands return a 0 ret value.

Listen/Listen_in

New in version 2014.7.0.

listen and its counterpart listen_in trigger mod_wait functions for states, when those states succeed and result in changes, similar to how watch its counterpart watch_in. Unlike watch and watch_in, listen, and listen_in will not modify the order of states and can be used to ensure your states are executed in the order they are defined. All listen/listen_in actions will occur at the end of a state run, after all states have completed.

```
restart-apache2:
  service.running:
    - name: apache2
    - listen:
      - file: /etc/apache2/apache2.conf

configure-apache2:
  file.managed:
    - name: /etc/apache2/apache2.conf
    - source: salt://apache2/apache2.conf
```

This example will cause apache2 to be restarted when the apache2.conf file is changed, but the apache2 restart will happen at the end of the state run.

```
restart-apache2:
  service.running:
    - name: apache2

configure-apache2:
  file.managed:
    - name: /etc/apache2/apache2.conf
    - source: salt://apache2/apache2.conf
    - listen_in:
      - service: apache2
```

This example does the same as the above example, but puts the state argument on the file resource, rather than the service resource.

check_cmd

New in version 2014.7.0.

Check Command is used for determining that a state did or did not run as expected.

NOTE: Under the hood `check_cmd` calls `cmd.retcode` with `python_shell=True`. This means the commands referenced by unless will be parsed by a shell, so beware of side-effects as this shell will be run with the same privileges as the salt-minion.

```
comment-repo:
  file.replace:
    - name: /etc/yum.repos.d/fedora.repo
    - pattern: '^enabled=0'
    - repl: enabled=1
    - check_cmd:
      - "! grep 'enabled=0' /etc/yum.repos.d/fedora.repo"
```

This will attempt to do a replace on all `enabled=0` in the .repo file, and replace them with `enabled=1`. The `check_cmd` is just a bash command. It will do a grep for `enabled=0` in the file, and if it finds any, it will return a 0, which will be inverted by the leading `!`, causing `check_cmd` to set the state as failed. If it returns a 1, meaning it didn't find any `enabled=0`, it will be inverted by the leading `!`, returning a 0, and declaring the function succeeded.

NOTE: This requisite `check_cmd` functions differently than the `check_cmd` of the `file.managed` state.

Overriding Checks

There are two commands used for the above checks.

`mod_run_check` is used to check for `onlyif` and `unless`. If the goal is to override the global check for these to variables, include a `mod_run_check` in the salt/states/ file.

`mod_run_check_cmd` is used to check for the check_cmd options. To override this one, include a `mod_run_check_cmd` in the states file for the state.

Retrying States

New in version 2017.7.0.

The retry option in a state allows it to be executed multiple times until a desired result is obtained or the maximum number of attempts have been made.

The retry option can be configured by the `attempts`, `until`, `interval`, and `splay` parameters.

The `attempts` parameter controls the maximum number of times the state will be run. If not specified or if an invalid value is specified, `attempts` will default to `2`.

The `until` parameter defines the result that is required to stop retrying the state. If not specified or if an invalid value is specified, `until` will default to `True`

The `interval` parameter defines the amount of time, in seconds, that the system will wait between attempts. If not specified or if an invalid value is specified, `interval` will default to `30`.

The `splay` parameter allows the `interval` to be additionally spread out. If not specified or if an invalid value is specified, `splay` defaults to `0` (i.e. no splaying will occur).

The following example will run the pkg.installed state until it returns `True` or it has been run `5` times. Each attempt will be `60` seconds apart and the interval will be splayed up to an additional `10` seconds:

```
my_retried_state:
  pkg.installed:
    - name: nano
    - retry:
        attempts: 5
```

```
        until: True
        interval: 60
        splay: 10
```

The following example will run the pkg.installed state with all the defaults for `retry`. The state will run up to 2 times, each attempt being `30` seconds apart, or until it returns `True`.

```
install_nano:
  pkg.installed:
    - name: nano
    - retry: True
```

The following example will run the file.exists state every `30` seconds up to `15` times or until the file exists (i.e. the state returns `True`).

```
wait_for_file:
  file.exists:
    - name: /path/to/file
    - retry:
        attempts: 15
        interval: 30
```

Return data from a retried state

When a state is retried, the returned output is as follows:

The `result` return value is the `result` from the final run. For example, imagine a state set to `retry` up to three times or `until True`. If the state returns `False` on the first run and then `True` on the second, the `result` of the state will be `True`.

The `started` return value is the `started` from the first run.

The `duration` return value is the total duration of all attempts plus the retry intervals.

The `comment` return value will include the result and comment from all previous attempts.

For example:

```
wait_for_file:
  file.exists:
    - name: /path/to/file
    - retry:
        attempts: 10
        interval: 2
        splay: 5
```

Would return similar to the following. The state result in this case is `False` (file.exist was run 10 times with a 2 second interval, but the file specified did not exist on any run).

```
      ID: wait_for_file
Function: file.exists
  Result: False
 Comment: Attempt 1: Returned a result of "False", with the following comment:
 ↪"Specified path /path/to/file does not exist"
          Attempt 2: Returned a result of "False", with the following comment:
 ↪"Specified path /path/to/file does not exist"
          Attempt 3: Returned a result of "False", with the following comment:
 ↪"Specified path /path/to/file does not exist"
```

```
          Attempt 4: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Attempt 5: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Attempt 6: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Attempt 7: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Attempt 8: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Attempt 9: Returned a result of "False", with the following comment:
↪"Specified path /path/to/file does not exist"
          Specified path /path/to/file does not exist
 Started: 09:08:12.903000
Duration: 47000.0 ms
 Changes:
```

6.1.16 Startup States

Sometimes it may be desired that the salt minion execute a state run when it is started. This alleviates the need for the master to initiate a state run on a new minion and can make provisioning much easier.

As of Salt 0.10.3 the minion config reads options that allow for states to be executed at startup. The options are *startup_states*, *sls_list*, and *top_file*.

The *startup_states* option can be passed one of a number of arguments to define how to execute states. The available options are:

highstate Execute `state.apply`

sls Read in the `sls_list` option and execute the named sls files

top Read in the `top_file` option and execute states based on that top file on the Salt Master

Examples:

Execute `state.apply` to run the *highstate* when starting the minion:

```
startup_states: highstate
```

Execute the sls files *edit.vim* and *hyper*:

```
startup_states: sls

sls_list:
  - edit.vim
  - hyper
```

6.1.17 State Testing

Executing a Salt state run can potentially change many aspects of a system and it may be desirable to first see what a state run is going to change before applying the run.

Salt has a test interface to report on exactly what will be changed, this interface can be invoked on any of the major state run functions:

```
salt '*' state.apply test=True
salt '*' state.apply mysls test=True
salt '*' state.single test=True
```

The test run is mandated by adding the `test=True` option to the states. The return information will show states that will be applied in yellow and the result is reported as `None`.

Default Test

If the value `test` is set to `True` in the minion configuration file then states will default to being executed in test mode. If this value is set then states can still be run by calling test=False:

```
salt '*' state.apply test=False
salt '*' state.apply mysls test=False
salt '*' state.single test=False
```

6.1.18 The Top File

Introduction

Most infrastructures are made up of groups of machines, each machine in the group performing a role similar to others. Those groups of machines work in concert with each other to create an application stack.

To effectively manage those groups of machines, an administrator needs to be able to create roles for those groups. For example, a group of machines that serve front-end web traffic might have roles which indicate that those machines should all have the Apache webserver package installed and that the Apache service should always be running.

In Salt, the file which contains a mapping between groups of machines on a network and the configuration roles that should be applied to them is called a `top file`.

Top files are named `top.sls` by default and they are so-named because they always exist in the ``top'' of a directory hierarchy that contains state files. That directory hierarchy is called a `state tree`.

A Basic Example

Top files have three components:

- **Environment:** A state tree directory containing a set of state files to configure systems.
- **Target:** A grouping of machines which will have a set of states applied to them.
- **State files:** A list of state files to apply to a target. Each state file describes one or more states to be configured and enforced on the targeted machines.

The relationship between these three components is nested as follows:

- Environments contain targets
- Targets contain states

Putting these concepts together, we can describe a scenario in which all minions with an ID that begins with `web` have an `apache` state applied to them:

```
base:          # Apply SLS files from the directory root for the 'base' environment
  'web*':      # All minions with a minion_id that begins with 'web'
    - apache   # Apply the state file named 'apache.sls'
```

Environments

Environments are directory hierarchies which contain a top file and a set of state files.

Environments can be used in many ways, however there is no requirement that they be used at all. In fact, the most common way to deploy Salt is with a single environment, called `base`. It is recommended that users only create multiple environments if they have a use case which specifically calls for multiple versions of state trees.

Getting Started with Top Files

Each environment is defined inside a salt master configuration variable called, *file_roots* .

In the most common single-environment setup, only the `base` environment is defined in *file_roots* along with only one directory path for the state tree.

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

In the above example, the top file will only have a single environment to pull from.

Next is a simple single-environment top file placed in `/srv/salt/top.sls`, illustrating that for the environment called `base`, all minions will have the state files named `core.sls` and `edit.sls` applied to them.

```
base:
  '*':
    - core
    - edit
```

Assuming the `file_roots` configuration from above, Salt will look in the `/srv/salt` directory for `core.sls` and `edit.sls`.

Multiple Environments

In some cases, teams may wish to create versioned state trees which can be used to test Salt configurations in isolated sets of systems such as a staging environment before deploying states into production.

For this case, multiple environments can be used to accomplish this task.

To create multiple environments, the *file_roots* option can be expanded:

```
file_roots:
  dev:
    - /srv/salt/dev
  qa:
    - /srv/salt/qa
  prod:
    - /srv/salt/prod
```

In the above, we declare three environments: `dev`, `qa` and `prod`. Each environment has a single directory assigned to it.

Our top file references the environments:

```
dev:
  'webserver*':
    - webserver
  'db*':
    - db
qa:
  'webserver*':
    - webserver
  'db*':
    - db
prod:
  'webserver*':
    - webserver
  'db*':
    - db
```

As seen above, the top file now declares the three environments and for each, target expressions are defined to map minions to state files. For example, all minions which have an ID beginning with the string `webserver` will have the webserver state from the requested environment assigned to it.

In this manner, a proposed change to a state could first be made in a state file in `/srv/salt/dev` and then be applied to development webservers before moving the state into QA by copying the state file into `/srv/salt/qa`.

Choosing an Environment to Target

The top file is used to assign a minion to an environment unless overridden using the methods described below. The environment in the top file must match valid fileserver environment (a.k.a. `saltenv`) in order for any states to be applied to that minion. When using the default fileserver backend, environments are defined in *file_roots*.

The states that will be applied to a minion in a given environment can be viewed using the *state.show_top* function.

Minions may be pinned to a particular environment by setting the *environment* value in the minion configuration file. In doing so, a minion will only request files from the environment to which it is assigned.

The environment may also be dynamically selected at runtime by passing it to the `salt`, `salt-call` or `salt-ssh` command. This is most commonly done with functions in the `state` module by using the `saltenv` argument. For example, to run a `highstate` on all minions, using only the top file and SLS files in the `prod` environment, run: `salt '*' state.highstate saltenv=prod`.

Note: Not all functions accept `saltenv` as an argument, see the documentation for an individual function documentation to verify.

Shorthand

If you assign only one SLS to a system, as in this example, a shorthand is also available:

```
base:
  '*': global
dev:
  'webserver*': webserver
  'db*':        db
qa:
```

```
  'webserver*': webserver
  'db*':        db
prod:
  'webserver*': webserver
  'db*':        db
```

Advanced Minion Targeting

In the examples above, notice that all of the target expressions are globs. The default match type in top files (since version 2014.7.0) is actually the *compound matcher*, not the glob matcher as in the CLI.

A single glob, when passed through the compound matcher, acts the same way as matching by glob, so in most cases the two are indistinguishable. However, there is an edge case in which a minion ID contains whitespace. While it is not recommended to include spaces in a minion ID, Salt will not stop you from doing so. However, since compound expressions are parsed word-by-word, if a minion ID contains spaces it will fail to match. In this edge case, it will be necessary to explicitly use the `glob` matcher:

```
base:
  'minion 1':
    - match: glob
    - foo
```

The available match types which can be set for a target expression in the top file are:

Match Type	Description
glob	Full minion ID or glob expression to match multiple minions (e.g. `minion123` or `minion*`)
pcre	Perl-compatible regular expression (PCRE) matching a minion ID (e.g. `web[0-3].domain.com`)
grain	Match a *grain*, optionally using globbing (e.g. `kernel:Linux` or `kernel:*BSD`)
grain_pcre	Match a *grain* using PCRE (e.g. `kernel:(Free|Open)BSD`)
list	Comma-separated list of minions (e.g. `minion1,minion2,minion3`)
pillar	*Pillar* match, optionally using globbing (e.g. `role:webserver` or `role:web*`)
pillar_pcre	*Pillar* match using PCRE (e.g. `role:web(server|proxy)`
pillar_exact	*Pillar* match with no globbing or PCRE (e.g. `role:webserver`)
ipcidr	Subnet or IP address (e.g. `172.17.0.0/16` or `10.2.9.80`)
data	Match values kept in the minion's datastore (created using the *data* execution module)
range	*Range* cluster
compound	Complex expression combining multiple match types (see *here*)
nodegroup	Pre-defined compound expressions in the master config file (see *here*)

Below is a slightly more complex top file example, showing some of the above match types:

```
# All files will be taken from the file path specified in the base
# environment in the ``file_roots`` configuration value.

base:
    # All minions which begin with the strings 'nag1' or any minion with
    # a grain set called 'role' with the value of 'monitoring' will have
    # the 'server.sls' state file applied from the 'nagios/' directory.

    'nag1* or G@role:monitoring':
        - nagios.server

    # All minions get the following three state files applied
```

```
'*':
    - ldap-client
    - networking
    - salt.minion

# All minions which have an ID that begins with the phrase
# 'salt-master' will have an SLS file applied that is named
# 'master.sls' and is in the 'salt' directory, underneath
# the root specified in the ``base`` environment in the
# configuration value for ``file_roots``.

'salt-master*':
    - salt.master

# Minions that have an ID matching the following regular
# expression will have the state file called 'web.sls' in the
# nagios/mon directory applied. Additionally, minions matching
# the regular expression will also have the 'server.sls' file
# in the apache/ directory applied.

# NOTE!
#
# Take note of the 'match' directive here, which tells Salt
# to treat the target string as a regex to be matched!

'^(memcache|web).(qa|prod).loc$':
    - match: pcre
    - nagios.mon.web
    - apache.server

# Minions that have a grain set indicating that they are running
# the Ubuntu operating system will have the state file called
# 'ubuntu.sls' in the 'repos' directory applied.
#
# Again take note of the 'match' directive here which tells
# Salt to match against a grain instead of a minion ID.

'os:Ubuntu':
    - match: grain
    - repos.ubuntu

# Minions that are either RedHat or CentOS should have the 'epel.sls'
# state applied, from the 'repos/' directory.

'os:(RedHat|CentOS)':
    - match: grain_pcre
    - repos.epel

# The three minions with the IDs of 'foo', 'bar' and 'baz' should
# have 'database.sls' applied.

'foo,bar,baz':
    - match: list
    - database

# Any minion for which the pillar key 'somekey' is set and has a value
# of that key matching 'abc' will have the 'xyz.sls' state applied.
```

```
    'somekey:abc':
        - match: pillar
        - xyz
```

How Top Files Are Compiled

When a *highstate* is executed and an environment is specified (either using the `environment` config option or by passing the saltenv when executing the *highstate*), then that environment's top file is the only top file used to assign states to minions, and only states from the specified environment will be run.

The remainder of this section applies to cases in which a *highstate* is executed without an environment specified.

With no environment specified, the minion will look for a top file in each environment, and each top file will be processed to determine the SLS files to run on the minions. By default, the top files from each environment will be merged together. In configurations with many environments, such as with *GitFS* where each branch and tag is treated as a distinct environment, this may cause unexpected results as SLS files from older tags cause defunct SLS files to be included in the highstate. In cases like this, it can be helpful to set `top_file_merging_strategy` to `same` to force each environment to use its own top file.

```
top_file_merging_strategy: same
```

Another option would be to set `state_top_saltenv` to a specific environment, to ensure that any top files in other environments are disregarded:

```
state_top_saltenv: base
```

With *GitFS*, it can also be helpful to simply manage each environment's top file separately, and/or manually specify the environment when executing the highstate to avoid any complicated merging scenarios. `gitfs_env_whitelist` and `gitfs_env_blacklist` can also be used to hide unneeded branches and tags from GitFS to reduce the number of top files in play.

When using multiple environments, it is not necessary to create a top file for each environment. The easiest-to-maintain approach is to use a single top file placed in the `base` environment. This is often infeasible with *GitFS* though, since branching/tagging can easily result in extra top files. However, when only the default (`roots`) fileserver backend is used, a single top file in the `base` environment is the most common way of configuring a *highstate*.

The following minion configuration options affect how top files are compiled when no environment is specified, it is recommended to follow the below four links to learn more about how these options work:

- `state_top_saltenv`
- `top_file_merging_strategy`
- `env_order`
- `default_top`

Top File Compilation Examples

For the scenarios below, assume the following configuration:

/etc/salt/master:

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

/srv/salt/base/top.sls:

```
base:
  '*':
    - base1
dev:
  '*':
    - dev1
qa:
  '*':
    - qa1
```

/srv/salt/dev/top.sls:

```
base:
  'minion1':
    - base2
dev:
  'minion2':
    - dev2
qa:
  '*':
    - qa1
    - qa2
```

Note: For the purposes of these examples, there is no top file in the `qa` environment.

Scenario 1 - `dev` Environment Specified

In this scenario, the *highstate* was either invoked with `saltenv=dev` or the minion has `environment: dev` set in the minion config file. The result will be that only the `dev2` SLS from the dev environment will be part of the *highstate*, and it will be applied to minion2, while minion1 will have no states applied to it.

If the `base` environment were specified, the result would be that only the `base1` SLS from the `base` environment would be part of the *highstate*, and it would be applied to all minions.

If the `qa` environment were specified, the *highstate* would exit with an error.

Scenario 2 - No Environment Specified, `top_file_merging_strategy` is ``merge''

In this scenario, assuming that the `base` environment's top file was evaluated first, the `base1`, `dev1`, and `qa1` states would be applied to all minions. If, for instance, the `qa` environment is not defined in **/srv/salt/base/top.sls**, then because there is no top file for the `qa` environment, no states from the `qa` environment would be applied.

Scenario 3 - No Environment Specified, `top_file_merging_strategy` is ``same''

Changed in version 2016.11.0: In prior versions, ``same'' did not quite work as described below (see here). This has now been corrected. It was decided that changing something like top file handling in a point release had the potential to unexpectedly impact users' top files too much, and it would be better to make this correction in a feature release.

In this scenario, `base1` from the `base` environment is applied to all minions. Additionally, `dev2` from the `dev` environment is applied to minion2.

If *`default_top`* is unset (or set to `base`, which happens to be the default), then `qa1` from the `qa` environment will be applied to all minions. If *`default_top`* were set to `dev`, then both `qa1` and `qa2` from the `qa` environment would be applied to all minions.

Scenario 4 - No Environment Specified, `top_file_merging_strategy` is ``merge_all''

New in version 2016.11.0.

In this scenario, all configured states in all top files are applied. From the `base` environment, `base1` would be applied to all minions, with `base2` being applied only to `minion1`. From the `dev` environment, `dev1` would be applied to all minions, with `dev2` being applied only to `minion2`. Finally, from the `qa` environment, both the `qa1` and `qa2` states will be applied to all minions. Note that the `qa1` states would not be applied twice, even though `qa1` appears twice.

6.1.19 SLS Template Variable Reference

The template engines available to sls files and file templates come loaded with a number of context variables. These variables contain information and functions to assist in the generation of templates. See each variable below for its availability -- not all variables are available in all templating contexts.

Salt

The *salt* variable is available to abstract the salt library functions. This variable is a python dictionary containing all of the functions available to the running salt minion. It is available in all salt templates.

```
{% for file in salt['cmd.run']('ls -1 /opt/to_remove').splitlines() %}
/opt/to_remove/{{ file }}:
  file.absent
{% endfor %}
```

Opts

The *opts* variable abstracts the contents of the minion's configuration file directly to the template. The *opts* variable is a dictionary. It is available in all templates.

```
{{ opts['cachedir'] }}
```

The `config.get` function also searches for values in the *opts* dictionary.

Pillar

The *pillar* dictionary can be referenced directly, and is available in all templates:

```
{{ pillar['key'] }}
```

Using the `pillar.get` function via the *salt* variable is generally recommended since a default can be safely set in the event that the value is not available in pillar and dictionaries can be traversed directly:

```
{{ salt['pillar.get']('key', 'failover_value') }}
{{ salt['pillar.get']('stuff:more:deeper') }}
```

Grains

The *grains* dictionary makes the minion's grains directly available, and is available in all templates:

```
{{ grains['os'] }}
```

The `grains.get` function can be used to traverse deeper grains and set defaults:

```
{{ salt['grains.get']('os') }}
```

saltenv

The *saltenv* variable is available in only in sls files when gathering the sls from an environment.

```
{{ saltenv }}
```

sls

The *sls* variable contains the sls reference value, and is only available in the actual SLS file (not in any files referenced in that SLS). The sls reference value is the value used to include the sls in top files or via the include option.

```
{{ sls }}
```

slspath

The *slspath* variable contains the path to the current sls file. The value of *slspath* in files referenced in the current sls depends on the reference method. For jinja includes *slspath* is the path to the current file. For salt includes *slspath* is the path to the included file.

```
{{ slspath }}
```

6.1.20 State Modules

State Modules are the components that map to actual enforcement and management of Salt states.

States are Easy to Write!

State Modules should be easy to write and straightforward. The information passed to the SLS data structures will map directly to the states modules.

Mapping the information from the SLS data is simple, this example should illustrate:

```
/etc/salt/master: # maps to "name", unless a "name" argument is specified below
  file.managed: # maps to <filename>.<function> - e.g. "managed" in https://github.com/
↪saltstack/salt/tree/develop/salt/states/file.py
    - user: root # one of many options passed to the manage function
    - group: root
    - mode: 644
    - source: salt://salt/master
```

Therefore this SLS data can be directly linked to a module, function, and arguments passed to that function.

This does issue the burden, that function names, state names and function arguments should be very human readable inside state modules, since they directly define the user interface.

Keyword Arguments

Salt passes a number of keyword arguments to states when rendering them, including the environment, a unique identifier for the state, and more. Additionally, keep in mind that the requisites for a state are part of the keyword arguments. Therefore, if you need to iterate through the keyword arguments in a state, these must be considered and handled appropriately. One such example is in the `pkgrepo.managed` state, which needs to be able to handle arbitrary keyword arguments and pass them to module execution functions. An example of how these keyword arguments can be handled can be found here.

Best Practices

A well-written state function will follow these steps:

Note: This is an extremely simplified example. Feel free to browse the source code for Salt's state modules to see other examples.

---

1. Set up the return dictionary and perform any necessary input validation (type checking, looking for use of mutually-exclusive arguments, etc.).

```
ret = {'name': name,
       'result': False,
       'changes': {},
       'comment': ''}

if foo and bar:
    ret['comment'] = 'Only one of foo and bar is permitted'
    return ret
```

2. Check if changes need to be made. This is best done with an information-gathering function in an accompanying *execution module*. The state should be able to use the return from this function to tell whether or not the minion is already in the desired state.

```
result = __salt__['modname.check'](name)
```

3. If step 2 found that the minion is already in the desired state, then exit immediately with a `True` result and without making any changes.

```
if result:
    ret['result'] = True
    ret['comment'] = '{0} is already installed'.format(name)
    return ret
```

4. If step 2 found that changes *do* need to be made, then check to see if the state was being run in test mode (i.e. with `test=True`). If so, then exit with a `None` result, a relevant comment, and (if possible) a `changes` entry describing what changes would be made.

```
if __opts__['test']:
    ret['result'] = None
    ret['comment'] = '{0} would be installed'.format(name)
    ret['changes'] = result
    return ret
```

5. Make the desired changes. This should again be done using a function from an accompanying execution module. If the result of that function is enough to tell you whether or not an error occurred, then you can exit with a `False` result and a relevant comment to explain what happened.

```
result = __salt__['modname.install'](name)
```

6. Perform the same check from step 2 again to confirm whether or not the minion is in the desired state. Just as in step 2, this function should be able to tell you by its return data whether or not changes need to be made.

```
ret['changes'] = __salt__['modname.check'](name)
```

   As you can see here, we are setting the `changes` key in the return dictionary to the result of the `modname.check` function (just as we did in step 4). The assumption here is that the information-gathering function will return a dictionary explaining what changes need to be made. This may or may not fit your use case.

7. Set the return data and return!

```
if ret['changes']:
    ret['comment'] = '{0} failed to install'.format(name)
else:
    ret['result'] = True
    ret['comment'] = '{0} was installed'.format(name)

return ret
```

Using Custom State Modules

Before the state module can be used, it must be distributed to minions. This can be done by placing them into `salt://_states/`. They can then be distributed manually to minions by running *saltutil.sync_states* or *saltutil.sync_all*. Alternatively, when running a *highstate* custom types will automatically be synced.

Any custom states which have been synced to a minion, that are named the same as one of Salt's default set of states, will take the place of the default state with the same name. Note that a state module's name defaults to one based on its filename (i.e. `foo.py` becomes state module `foo`), but that its name can be overridden by using a *__virtual__ function*.

Cross Calling Execution Modules from States

As with Execution Modules, State Modules can also make use of the `__salt__` and `__grains__` data. See *cross calling execution modules*.

It is important to note that the real work of state management should not be done in the state module unless it is needed. A good example is the pkg state module. This module does not do any package management work, it just calls the pkg execution module. This makes the pkg state module completely generic, which is why there is only one pkg state module and many backend pkg execution modules.

On the other hand some modules will require that the logic be placed in the state module, a good example of this is the file module. But in the vast majority of cases this is not the best approach, and writing specific execution modules to do the backend work will be the optimal solution.

Cross Calling State Modules

All of the Salt state modules are available to each other and state modules can call functions available in other state modules.

The variable `__states__` is packed into the modules after they are loaded into the Salt minion.

The `__states__` variable is a Python dictionary containing all of the state modules. Dictionary keys are strings representing the names of the modules and the values are the functions themselves.

Salt state modules can be cross-called by accessing the value in the `__states__` dict:

```
ret = __states__['file.managed'](name='/tmp/myfile', source='salt://myfile')
```

This code will call the *managed* function in the *file* state module and pass the arguments `name` and `source` to it.

Return Data

A State Module must return a dict containing the following keys/values:

- **name:** The same value passed to the state as ``name''.
- **changes:** A dict describing the changes made. Each thing changed should be a key, with its value being another dict with keys called ``old'' and ``new'' containing the old/new values. For example, the pkg state's **changes** dict has one key for each package changed, with the ``old'' and ``new'' keys in its sub-dict containing the old and new versions of the package. For example, the final changes dictionary for this scenario would look something like this:

```
ret['changes'].update({'my_pkg_name': {'old': '',
                                       'new': 'my_pkg_name-1.0'}})
```

- **result:** A tristate value. `True` if the action was successful, `False` if it was not, or `None` if the state was run in test mode, `test=True`, and changes would have been made if the state was not run in test mode.

	live mode	test mode
no changes	True	True
successful changes	True	None
failed changes	False	False or None

Note: Test mode does not predict if the changes will be successful or not, and hence the result for pending changes is usually `None`.

However, if a state is going to fail and this can be determined in test mode without applying the change, `False` can be returned.

- **comment:** A list of strings or a single string summarizing the result. Note that support for lists of strings is available as of Salt 2018.3.0. Lists of strings will be joined with newlines to form the final comment; this is useful to allow multiple comments from subparts of a state. Prefer to keep line lengths short (use multiple lines as needed), and end with punctuation (e.g. a period) to delimit multiple comments.

The return data can also, include the **pchanges** key, this stands for *predictive changes*. The **pchanges** key informs the State system what changes are predicted to occur.

Note: States should not return data which cannot be serialized such as frozensets.

Test State

All states should check for and support `test` being passed in the options. This will return data about what changes would occur if the state were actually run. An example of such a check could look like this:

```
# Return comment of changes if test.
if __opts__['test']:
    ret['result'] = None
    ret['comment'] = 'State Foo will execute with param {0}'.format(bar)
    return ret
```

Make sure to test and return before performing any real actions on the minion.

Note: Be sure to refer to the `result` table listed above and displaying any possible changes when writing support for `test`. Looking for changes in a state is essential to `test=true` functionality. If a state is predicted to have no changes when `test=true` (or `test: true` in a config file) is used, then the result of the final state **should not** be `None`.

Watcher Function

If the state being written should support the watch requisite then a watcher function needs to be declared. The watcher function is called whenever the watch requisite is invoked and should be generic to the behavior of the state itself.

The watcher function should accept all of the options that the normal state functions accept (as they will be passed into the watcher function).

A watcher function typically is used to execute state specific reactive behavior, for instance, the watcher for the service module restarts the named service and makes it useful for the watcher to make the service react to changes in the environment.

The watcher function also needs to return the same data that a normal state function returns.

Mod_init Interface

Some states need to execute something only once to ensure that an environment has been set up, or certain conditions global to the state behavior can be predefined. This is the realm of the mod_init interface.

A state module can have a function called **mod_init** which executes when the first state of this type is called. This interface was created primarily to improve the pkg state. When packages are installed the package metadata needs to be refreshed, but refreshing the package metadata every time a package is installed is wasteful. The mod_init function for the pkg state sets a flag down so that the first, and only the first, package installation attempt will refresh the package database (the package database can of course be manually called to refresh via the `refresh` option in the pkg state).

The mod_init function must accept the **Low State Data** for the given executing state as an argument. The low state data is a dict and can be seen by executing the state.show_lowstate function. Then the mod_init function must return a bool. If the return value is True, then the mod_init function will not be executed again, meaning that the needed behavior has been set up. Otherwise, if the mod_init function returns False, then the function will be called the next time.

A good example of the mod_init function is found in the pkg state module:

```
def mod_init(low):
    '''
    Refresh the package database here so that it only needs to happen once
    '''
    if low['fun'] == 'installed' or low['fun'] == 'latest':
        rtag = __gen_rtag()
        if not os.path.exists(rtag):
            open(rtag, 'w+').write('')
        return True
    else:
        return False
```

The mod_init function in the pkg state accepts the low state data as `low` and then checks to see if the function being called is going to install packages, if the function is not going to install packages then there is no need to refresh the package database. Therefore if the package database is prepared to refresh, then return True and the mod_init will not be called the next time a pkg state is evaluated, otherwise return False and the mod_init will be called next time a pkg state is evaluated.

Log Output

You can call the logger from custom modules to write messages to the minion logs. The following code snippet demonstrates writing log messages:

```
import logging

log = logging.getLogger(__name__)

log.info('Here is Some Information')
log.warning('You Should Not Do That')
log.error('It Is Busted')
```

Strings and Unicode

A state module author should always assume that strings fed to the module have already decoded from strings into Unicode. In Python 2, these will be of type `Unicode' and in Python 3 they will be of type `str`. Calling from a state to other Salt sub-systems, such as execution modules should pass Unicode (or bytes if passing binary data). In the rare event that a state needs to write directly to disk, Unicode should be encoded to a string immediately before writing to disk. An author may use `__salt_system_encoding__` to learn what the encoding type of the system is. For example, *`my_string'.encode(__salt_system_encoding__').*

Full State Module Example

The following is a simplistic example of a full state module and function. Remember to call out to execution modules to perform all the real work. The state module should only perform ``before'' and ``after'' checks.

1. Make a custom state module by putting the code into a file at the following path: **/srv/salt/_states/my_custom_state.py**.
2. Distribute the custom state module to the minions:

```
salt '*' saltutil.sync_states
```

3. Write a new state to use the custom state by making a new state file, for instance **/srv/salt/my_custom_state.sls**.
4. Add the following SLS configuration to the file created in Step 3:

```
human_friendly_state_id:        # An arbitrary state ID declaration.
  my_custom_state:              # The custom state module name.
    - enforce_custom_thing      # The function in the custom state module.
    - name: a_value             # Maps to the ``name`` parameter in the custom
↪function.
    - foo: Foo                  # Specify the required ``foo`` parameter.
    - bar: False                # Override the default value for the ``bar``
↪parameter.
```

Example state module

```
import salt.exceptions

def enforce_custom_thing(name, foo, bar=True):
    '''
    Enforce the state of a custom thing

    This state module does a custom thing. It calls out to the execution module
    ``my_custom_module`` in order to check the current system and perform any
    needed changes.

    name
        The thing to do something to
    foo
        A required argument
    bar : True
        An argument with a default value
    '''
    ret = {
        'name': name,
        'changes': {},
        'result': False,
        'comment': '',
        'pchanges': {},
        }

    # Start with basic error-checking. Do all the passed parameters make sense
    # and agree with each-other?
    if bar == True and foo.startswith('Foo'):
        raise salt.exceptions.SaltInvocationError(
```

```
            'Argument "foo" cannot start with "Foo" if argument "bar" is True.')

    # Check the current state of the system. Does anything need to change?
    current_state = __salt__['my_custom_module.current_state'](name)

    if current_state == foo:
        ret['result'] = True
        ret['comment'] = 'System already in the correct state'
        return ret

    # The state of the system does need to be changed. Check if we're running
    # in ``test=true`` mode.
    if __opts__['test'] == True:
        ret['comment'] = 'The state of "{0}" will be changed.'.format(name)
        ret['pchanges'] = {
            'old': current_state,
            'new': 'Description, diff, whatever of the new state',
        }

        # Return ``None`` when running with ``test=true``.
        ret['result'] = None

        return ret

    # Finally, make the actual change and return the result.
    new_state = __salt__['my_custom_module.change_state'](name, foo)

    ret['comment'] = 'The state of "{0}" was changed!'.format(name)

    ret['changes'] = {
        'old': current_state,
        'new': new_state,
    }

    ret['result'] = True

    return ret
```

6.1.21 State Management

State management, also frequently called Software Configuration Management (SCM), is a program that puts and keeps a system into a predetermined state. It installs software packages, starts or restarts services or puts configuration files in place and watches them for changes.

Having a state management system in place allows one to easily and reliably configure and manage a few servers or a few thousand servers. It allows configurations to be kept under version control.

Salt States is an extension of the Salt Modules that we discussed in the previous *remote execution* tutorial. Instead of calling one-off executions the state of a system can be easily defined and then enforced.

6.1.22 Understanding the Salt State System Components

The Salt state system is comprised of a number of components. As a user, an understanding of the SLS and renderer systems are needed. But as a developer, an understanding of Salt states and how to write the states is needed as well.

Note: States are compiled and executed only on minions that have been targeted. To execute functions directly on masters, see *runners*.

Salt SLS System

The primary system used by the Salt state system is the SLS system. SLS stands for **S**a**L**t **S**tate.

The Salt States are files which contain the information about how to configure Salt minions. The states are laid out in a directory tree and can be written in many different formats.

The contents of the files and the way they are laid out is intended to be as simple as possible while allowing for maximum flexibility. The files are laid out in states and contains information about how the minion needs to be configured.

SLS File Layout

SLS files are laid out in the Salt file server.

A simple layout can look like this:

```
top.sls
ssh.sls
sshd_config
users/init.sls
users/admin.sls
salt/master.sls
web/init.sls
```

The `top.sls` file is a key component. The `top.sls` files is used to determine which SLS files should be applied to which minions.

The rest of the files with the `.sls` extension in the above example are state files.

Files without a `.sls` extensions are seen by the Salt master as files that can be downloaded to a Salt minion.

States are translated into dot notation. For example, the `ssh.sls` file is seen as the ssh state and the `users/admin.sls` file is seen as the users.admin state.

Files named `init.sls` are translated to be the state name of the parent directory, so the `web/init.sls` file translates to the `web` state.

In Salt, everything is a file; there is no ``magic translation'' of files and file types. This means that a state file can be distributed to minions just like a plain text or binary file.

SLS Files

The Salt state files are simple sets of data. Since SLS files are just data they can be represented in a number of different ways.

The default format is YAML generated from a Jinja template. This allows for the states files to have all the language constructs of Python and the simplicity of YAML.

State files can then be complicated Jinja templates that translate down to YAML, or just plain and simple YAML files.

The State files are simply common data structures such as dictionaries and lists, constructed using a templating language such as YAML.

Here is an example of a Salt State:

```
vim:
  pkg.installed: []

salt:
  pkg.latest:
    - name: salt
  service.running:
    - names:
      - salt-master
      - salt-minion
    - require:
      - pkg: salt
    - watch:
      - file: /etc/salt/minion

/etc/salt/minion:
  file.managed:
    - source: salt://salt/minion
    - user: root
    - group: root
    - mode: 644
    - require:
      - pkg: salt
```

This short stanza will ensure that vim is installed, Salt is installed and up to date, the salt-master and salt-minion daemons are running and the Salt minion configuration file is in place. It will also ensure everything is deployed in the right order and that the Salt services are restarted when the watched file updated.

The Top File

The top file controls the mapping between minions and the states which should be applied to them.

The top file specifies which minions should have which SLS files applied and which environments they should draw those SLS files from.

The top file works by specifying environments on the top-level.

Each environment contains *target expressions* to match minions. Finally, each target expression contains a list of Salt states to apply to matching minions:

```
base:
  '*':
    - salt
    - users
    - users.admin
  'saltmaster.*':
    - match: pcre
    - salt.master
```

This above example uses the base environment which is built into the default Salt setup.

The base environment has target expressions. The first one matches all minions, and the SLS files below it apply to all minions.

The second expression is a regular expression that will match all minions with an ID matching `saltmaster.*` and specifies that for those minions, the salt.master state should be applied.

Important: Since version 2014.7.0, the default matcher (when one is not explicitly defined as in the second expression in the above example) is the *compound* matcher. Since this matcher parses individual words in the expression, minion IDs containing spaces will not match properly using this matcher. Therefore, if your target expression is designed to match a minion ID containing spaces, it will be necessary to specify a different match type (such as `glob`). For example:

```
base:
  'test minion':
    - match: glob
    - foo
    - bar
    - baz
```

A full table of match types available in the top file can be found *here*.

Reloading Modules

Some Salt states require that specific packages be installed in order for the module to load. As an example the *pip* state module requires the pip package for proper name and version parsing.

In most of the common cases, Salt is clever enough to transparently reload the modules. For example, if you install a package, Salt reloads modules because some other module or state might require just that package which was installed.

On some edge-cases salt might need to be told to reload the modules. Consider the following state file which we'll call `pep8.sls`:

```
python-pip:
  cmd.run:
    - name: |
        easy_install --script-dir=/usr/bin -U pip
    - cwd: /

pep8:
  pip.installed:
    - require:
      - cmd: python-pip
```

The above example installs pip using `easy_install` from setuptools and installs pep8 using *pip*, which, as told earlier, requires pip to be installed system-wide. Let's execute this state:

```
salt-call state.apply pep8
```

The execution output would be something like:

```
----------
    State: - pip
    Name:      pep8
    Function:  installed
        Result:    False
        Comment:   State pip.installed found in sls pep8 is unavailable
```

```
        Changes:

Summary
------------
Succeeded: 1
Failed:    1
------------
Total:     2
```

If we executed the state again the output would be:

```
----------
    State: - pip
    Name:      pep8
    Function:  installed
        Result:    True
        Comment:   Package was successfully installed
        Changes:   pep8==1.4.6: Installed

Summary
------------
Succeeded: 2
Failed:    0
------------
Total:     2
```

Since we installed pip using *cmd*, Salt has no way to know that a system-wide package was installed.

On the second execution, since the required pip package was installed, the state executed correctly.

Note: Salt does not reload modules on every state run because doing so would greatly slow down state execution.

So how do we solve this *edge-case*? `reload_modules`!

`reload_modules` is a boolean option recognized by salt on **all** available states which forces salt to reload its modules once a given state finishes.

The modified state file would now be:

```
python-pip:
  cmd.run:
    - name: |
        easy_install --script-dir=/usr/bin -U pip
    - cwd: /
    - reload_modules: true

pep8:
  pip.installed:
    - require:
      - cmd: python-pip
```

Let's run it, once:

```
salt-call state.apply pep8
```

The output is:

```
----------
    State: - pip
    Name:      pep8
    Function:  installed
        Result:    True
        Comment:   Package was successfully installed
        Changes:   pep8==1.4.6: Installed

Summary
------------
Succeeded: 2
Failed:    0
------------
Total:     2
```

Utility Modules - Code Reuse in Custom Modules

New in version 2015.5.0.

Changed in version 2016.11.0: These can now be synced to the Master for use in custom Runners, and in custom execution modules called within Pillar SLS files.

When extending Salt by writing custom (*state modules*), *execution modules*, etc., sometimes there is a need for a function to be available to more than just one kind of custom module. For these cases, Salt supports what are called ``utility modules''. These modules are like normal execution modules, but instead of being invoked in Salt code using `__salt__`, the `__utils__` prefix is used instead.

For example, assuming the following simple utility module, saved to `salt://_utils/foo.py`

```
# -*- coding: utf-8 -*-
'''
My utils module
---------------

This module contains common functions for use in my other custom types.
'''

def bar():
    return 'baz'
```

Once synced to a minion, this function would be available to other custom Salt types like so:

```
# -*- coding: utf-8 -*-
'''
My awesome execution module
---------------------------
'''

def observe_the_awesomeness():
    '''
    Prints information from my utility module

    CLI Example:

    .. code-block:: bash

        salt '*' mymodule.observe_the_awesomeness
    '''
    return __utils__['foo.bar']()
```

Utility modules, like any other kind of Salt extension, support using a *__virtual__ function* to conditionally load them, or load them under a different namespace. For instance, if the utility module above were named `salt://_utils/mymodule.py` it could be made to be loaded as the `foo` utility module with a `__virtual__` function.

```
# -*- coding: utf-8 -*-
'''
My utils module
---------------

This module contains common functions for use in my other custom types.
'''

def __virtual__():
    '''
    Load as a different name
    '''
    return 'foo'

def bar():
    return 'baz'
```

New in version 2018.3.0: Instantiating objects from classes declared in util modules works with Master side modules, such as Runners, Outputters, etc.

Also you could even write your utility modules in object oriented fashion:

```
# -*- coding: utf-8 -*-
'''
My OOP-style utils module
-------------------------

This module contains common functions for use in my other custom types.
'''

class Foo(object):

    def __init__(self):
        pass

    def bar(self):
        return 'baz'
```

And import them into other custom modules:

```
# -*- coding: utf-8 -*-
'''
My awesome execution module
---------------------------
'''

import mymodule

def observe_the_awesomeness():
    '''
    Prints information from my utility module

    CLI Example:
```

```
    .. code-block:: bash

        salt '*' mymodule.observe_the_awesomeness
    '''
    foo = mymodule.Foo()
    return foo.bar()
```

These are, of course, contrived examples, but they should serve to show some of the possibilities opened up by writing utility modules. Keep in mind though that states still have access to all of the execution modules, so it is not necessary to write a utility module to make a function available to both a state and an execution module. One good use case for utility modules is one where it is necessary to invoke the same function from a custom *outputter*/returner, as well as an execution module.

Utility modules placed in `salt://_utils/` will be synced to the minions when a *highstate* is run, as well as when any of the following Salt functions are called:

- *saltutil.sync_utils*
- *saltutil.sync_all*

As of the Fluorine release, as well as 2017.7.7 and 2018.3.2 in their respective release cycles, the `sync` argument to `state.apply`/*state.sls* can be used to sync custom types when running individual SLS files.

To sync to the Master, use either of the following:

- *saltutil.sync_utils*
- *saltutil.sync_all*

CHAPTER 8

Events & Reactor

8.1 Event System

The Salt Event System is used to fire off events enabling third party applications or external processes to react to behavior within Salt. The event system uses a publish-subscribe pattern, otherwise know as pub/sub.

8.1.1 Event Bus

The event system is comprised of a two primary components, which make up the concept of an Event Bus:

- The event sockets, which publish events
- The event library, which can listen to events and send events into the salt system

Events are published onto the event bus and event bus subscribers listen for the published events.

The event bus is used for both inter-process communication as well as network transport in Salt. Inter-process communication is provided through UNIX domain sockets (UDX).

The Salt Master and each Salt Minion has their own event bus.

8.1.2 Event types

Salt Master Events

These events are fired on the Salt Master event bus. This list is **not** comprehensive.

Authentication events

`salt/auth`

Fired when a minion performs an authentication check with the master.

Variables

- **`id`** -- The minion ID.
- **`act`** -- The current status of the minion key: `accept`, `pend`, `reject`.
- **`pub`** -- The minion public key.

Note: Minions fire auth events on fairly regular basis for a number of reasons. Writing reactors to respond to events through the auth cycle can lead to infinite reactor event loops (minion tries to auth, reactor responds by doing something that generates another auth event, minion sends auth event, etc.). Consider reacting to `salt/key` or `salt/minion/<MID>/start` or firing a custom event tag instead.

Start events

`salt/minion/<MID>/start`
Fired every time a minion connects to the Salt master.
Variables **`id`** -- The minion ID.

Key events

`salt/key`
Fired when accepting and rejecting minions keys on the Salt master. These happen as a result of actions undertaken by the *salt-key* command.
Variables

- **`id`** -- The minion ID.
- **`act`** -- The new status of the minion key: `accept`, `delete`,

Warning: If a master is in `auto_accept mode`, `salt/key` events will not be fired when the keys are accepted. In addition, pre-seeding keys (like happens through *Salt-Cloud*) will not cause firing of these events.

Job events

`salt/job/<JID>/new`
Fired as a new job is sent out to minions.
Variables

- **`jid`** -- The job ID.
- **`tgt`** -- The target of the job: `*`, a minion ID, `G@os_family:RedHat`, etc.
- **`tgt_type`** -- The type of targeting used: `glob`, `grain`, `compound`, etc.
- **`fun`** -- The function to run on minions: `test.ping`, `network.interfaces`, etc.
- **`arg`** -- A list of arguments to pass to the function that will be called.
- **`minions`** -- A list of minion IDs that Salt expects will return data for this job.
- **`user`** -- The name of the user that ran the command as defined in Salt's Publisher ACL or external auth.

`salt/job/<JID>/ret/<MID>`
Fired each time a minion returns data for a job.
Variables

- **`id`** -- The minion ID.
- **`jid`** -- The job ID.

- **retcode** -- The return code for the job.
- **fun** -- The function the minion ran. E.g., `test.ping`.
- **return** -- The data returned from the execution module.

salt/job/<JID>/prog/<MID>/<RUN NUM>
Fired each time a each function in a state run completes execution. Must be enabled using the *state_events* option.
Variables

- **data** -- The data returned from the state module function.
- **id** -- The minion ID.
- **jid** -- The job ID.

Runner Events

salt/run/<JID>/new
Fired as a runner begins execution
Variables

- **jid** -- The job ID.
- **fun** -- The name of the runner function, with `runner.` prepended to it (e.g. `runner.jobs.lookup_jid`)
- **fun_args** -- The arguments passed to the runner function (e.g. `['20160829225914848058']`)
- **user** -- The user who executed the runner (e.g. `root`)

salt/run/<JID>/ret
Fired when a runner function returns
Variables

- **jid** -- The job ID.
- **fun** -- The name of the runner function, with `runner.` prepended to it (e.g. `runner.jobs.lookup_jid`)
- **fun_args** -- The arguments passed to the runner function (e.g. `['20160829225914848058']`)
- **return** -- The data returned by the runner function

salt/run/<JID>/args
New in version 2016.11.0.

Fired by the *state.orchestrate* runner
Variables

- **name** -- The ID declaration for the orchestration job (i.e. the line above `salt.state`, `salt.function`, `salt.runner`, etc.)
- **type** -- The type of orchestration job being run (e.g. `state`)
- **tgt** -- The target expression (e.g. `*`). Included for `state` and `function` types only.
- **args** -- The args passed to the orchestration job. **Note:** for `state` and `function` types, also includes a `tgt_type` value which shows what kind of match (`glob`,

pcre, etc.) was used. This value was named expr_form in the 2016.11 release cycle but has been renamed to tgt_type in 2017.7.0 for consistency with other events.

Presence Events

salt/presence/present
Events fired on a regular interval about currently connected, newly connected, or recently disconnected minions. Requires the *presence_events* setting to be enabled.

Variables **present** -- A list of minions that are currently connected to the Salt master.

salt/presence/change
Fired when the Presence system detects new minions connect or disconnect.

Variables

- **new** -- A list of minions that have connected since the last presence event.
- **lost** -- A list of minions that have disconnected since the last presence event.

Cloud Events

Unlike other Master events, salt-cloud events are not fired on behalf of a Salt Minion. Instead, salt-cloud events are fired on behalf of a VM. This is because the minion-to-be may not yet exist to fire events to or also may have been destroyed.

This behavior is reflected by the name variable in the event data for salt-cloud events as compared to the id variable for Salt Minion-triggered events.

salt/cloud/<VM NAME>/creating
Fired when salt-cloud starts the VM creation process.

Variables

- **name** -- the name of the VM being created.
- **event** -- description of the event.
- **provider** -- the cloud provider of the VM being created.
- **profile** -- the cloud profile for the VM being created.

salt/cloud/<VM NAME>/deploying
Fired when the VM is available and salt-cloud begins deploying Salt to the new VM.

Variables

- **name** -- the name of the VM being created.
- **event** -- description of the event.
- **kwargs** -- options available as the deploy script is invoked: conf_file, deploy_command, display_ssh_output, host, keep_tmp, key_filename, make_minion, minion_conf, name, parallel, preseed_minion_keys, script, script_args, script_env, sock_dir, start_action, sudo, tmp_dir, tty, username

salt/cloud/<VM NAME>/requesting
Fired when salt-cloud sends the request to create a new VM.

Variables

- **event** -- description of the event.
- **location** -- the location of the VM being requested.

- **kwargs** -- options available as the VM is being requested: `Action`, `ImageId`, `InstanceType`, `KeyName`, `MaxCount`, `MinCount`, `SecurityGroup.1`

salt/cloud/<VM NAME>/querying
Fired when salt-cloud queries data for a new instance.
Variables

- **event** -- description of the event.
- **instance_id** -- the ID of the new VM.

salt/cloud/<VM NAME>/tagging
Fired when salt-cloud tags a new instance.
Variables

- **event** -- description of the event.
- **tags** -- tags being set on the new instance.

salt/cloud/<VM NAME>/waiting_for_ssh
Fired while the salt-cloud deploy process is waiting for ssh to become available on the new instance.
Variables

- **event** -- description of the event.
- **ip_address** -- IP address of the new instance.

salt/cloud/<VM NAME>/deploy_script
Fired once the deploy script is finished.
Variables **event** -- description of the event.

salt/cloud/<VM NAME>/created
Fired once the new instance has been fully created.
Variables

- **name** -- the name of the VM being created.
- **event** -- description of the event.
- **instance_id** -- the ID of the new instance.
- **provider** -- the cloud provider of the VM being created.
- **profile** -- the cloud profile for the VM being created.

salt/cloud/<VM NAME>/destroying
Fired when salt-cloud requests the destruction of an instance.
Variables

- **name** -- the name of the VM being created.
- **event** -- description of the event.
- **instance_id** -- the ID of the new instance.

salt/cloud/<VM NAME>/destroyed
Fired when an instance has been destroyed.
Variables

- **name** -- the name of the VM being created.
- **event** -- description of the event.
- **instance_id** -- the ID of the new instance.

8.1.3 Listening for Events

Salt's event system is used heavily within Salt and it is also written to integrate heavily with existing tooling and scripts. There is a variety of ways to consume it.

From the CLI

The quickest way to watch the event bus is by calling the *state.event runner*:

```
salt-run state.event pretty=True
```

That runner is designed to interact with the event bus from external tools and shell scripts. See the documentation for more examples.

Remotely via the REST API

Salt's event bus can be consumed *salt.netapi.rest_cherrypy.app.Events* as an HTTP stream from external tools or services.

```
curl -SsNk https://salt-api.example.com:8000/events?token=05A3
```

From Python

Python scripts can access the event bus only as the same system user that Salt is running as.

The event system is accessed via the event library and can only be accessed by the same system user that Salt is running as. To listen to events a SaltEvent object needs to be created and then the get_event function needs to be run. The SaltEvent object needs to know the location that the Salt Unix sockets are kept. In the configuration this is the `sock_dir` option. The `sock_dir` option defaults to ``/var/run/salt/master'' on most systems.

The following code will check for a single event:

```
import salt.config
import salt.utils.event

opts = salt.config.client_config('/etc/salt/master')

event = salt.utils.event.get_event(
        'master',
        sock_dir=opts['sock_dir'],
        transport=opts['transport'],
        opts=opts)

data = event.get_event()
```

Events will also use a ``tag''. Tags allow for events to be filtered by prefix. By default all events will be returned. If only authentication events are desired, then pass the tag ``salt/auth''.

The `get_event` method has a default poll time assigned of 5 seconds. To change this time set the ``wait'' option.

The following example will only listen for auth events and will wait for 10 seconds instead of the default 5.

```
data = event.get_event(wait=10, tag='salt/auth')
```

To retrieve the tag as well as the event data, pass `full=True`:

```
evdata = event.get_event(wait=10, tag='salt/job', full=True)

tag, data = evdata['tag'], evdata['data']
```

Instead of looking for a single event, the `iter_events` method can be used to make a generator which will continually yield salt events.

The iter_events method also accepts a tag but not a wait time:

```
for data in event.iter_events(tag='salt/auth'):
    print(data)
```

And finally event tags can be globbed, such as they can be in the Reactor, using the fnmatch library.

```
import fnmatch

import salt.config
import salt.utils.event

opts = salt.config.client_config('/etc/salt/master')

sevent = salt.utils.event.get_event(
        'master',
        sock_dir=opts['sock_dir'],
        transport=opts['transport'],
        opts=opts)

while True:
    ret = sevent.get_event(full=True)
    if ret is None:
        continue

    if fnmatch.fnmatch(ret['tag'], 'salt/job/*/ret/*'):
        do_something_with_job_return(ret['data'])
```

8.1.4 Firing Events

It is possible to fire events on either the minion's local bus or to fire events intended for the master.

To fire a local event from the minion on the command line call the *event.fire* execution function:

```
salt-call event.fire '{"data": "message to be sent in the event"}' 'tag'
```

To fire an event to be sent up to the master from the minion call the *event.send* execution function. Remember YAML can be used at the CLI in function arguments:

```
salt-call event.send 'myco/mytag/success' '{success: True, message: "It works!"}'
```

If a process is listening on the minion, it may be useful for a user on the master to fire an event to it:

```
# Job on minion
import salt.utils.event

event = salt.utils.event.MinionEvent(**__opts__)

for evdata in event.iter_events(tag='customtag/'):
    return evdata # do your processing here...
```

```
salt minionname event.fire '{"data": "message for the minion"}' 'customtag/african/
↪unladen'
```

8.1.5 Firing Events from Python

From Salt execution modules

Events can be very useful when writing execution modules, in order to inform various processes on the master when a certain task has taken place. This is easily done using the normal cross-calling syntax:

```
# /srv/salt/_modules/my_custom_module.py

def do_something():
    '''
    Do something and fire an event to the master when finished

    CLI Example::

        salt '*' my_custom_module:do_something
    '''
    # do something!
    __salt__['event.send']('myco/my_custom_module/finished', {
        'finished': True,
        'message': "The something is finished!",
    })
```

From Custom Python Scripts

Firing events from custom Python code is quite simple and mirrors how it is done at the CLI:

```
import salt.client

caller = salt.client.Caller()

caller.sminion.functions['event.send'](
    'myco/myevent/success',
    {
        'success': True,
        'message': "It works!",
    }
)
```

8.2 Beacons

Beacons let you use the Salt event system to monitor non-Salt processes. The beacon system allows the minion to hook into a variety of system processes and continually monitor these processes. When monitored activity occurs in a system process, an event is sent on the Salt event bus that can be used to trigger a *reactor*.

Salt beacons can currently monitor and send Salt events for many system activities, including:

- file system changes
- system load

- service status
- shell activity, such as user login
- network and disk usage

See *beacon modules* for a current list.

Note: Salt beacons are an event generation mechanism. Beacons leverage the Salt *reactor* system to make changes when beacon events occur.

8.2.1 Configuring Beacons

Salt beacons do not require any changes to the system components that are being monitored, everything is configured using Salt.

Beacons are typically enabled by placing a `beacons:` top level block in `/etc/salt/minion` or any file in `/etc/salt/minion.d/` such as `/etc/salt/minion.d/beacons.conf` or add it to pillars for that minion:

```
beacons:
  inotify:
    - files:
        /etc/important_file: {}
        /opt: {}
```

The beacon system, like many others in Salt, can also be configured via the minion pillar, grains, or local config file.

Note: The *inotify* beacon only works on OSes that have *inotify* kernel support. Currently this excludes FreeBSD, macOS, and Windows.

All beacon configuration is done using list based configuration.

Beacon Monitoring Interval

Beacons monitor on a 1-second interval by default. To set a different interval, provide an `interval` argument to a beacon. The following beacons run on 5- and 10-second intervals:

```
beacons:
  inotify:
    - files:
        /etc/important_file: {}
        /opt: {}
    - interval: 5
    - disable_during_state_run: True
  load:
    - averages:
        1m:
          - 0.0
          - 2.0
        5m:
          - 0.0
          - 1.5
        15m:
```

```
        - 0.1
        - 1.0
    - interval: 10
```

Avoiding Event Loops

It is important to carefully consider the possibility of creating a loop between a reactor and a beacon. For example, one might set up a beacon which monitors whether a file is read which in turn fires a reactor to run a state which in turn reads the file and re-fires the beacon.

To avoid these types of scenarios, the `disable_during_state_run` argument may be set. If a state run is in progress, the beacon will not be run on its regular interval until the minion detects that the state run has completed, at which point the normal beacon interval will resume.

```
beacons:
  inotify:
    - files:
        /etc/important_file: {}
    - disable_during_state_run: True
```

Note: For beacon writers: If you need extra stuff to happen, like closing file handles for the `disable_during_state_run` to actually work, you can add a *close()* function to the beacon to run those extra things. See the *inotify* beacon.

8.2.2 Beacon Example

This example demonstrates configuring the *inotify* beacon to monitor a file for changes, and then restores the file to its original contents if a change was made.

Note: The inotify beacon requires Pyinotify on the minion, install it using `salt myminion pkg.install python-inotify`.

Create Watched File

Create the file named `/etc/important_file` and add some simple content:

```
important_config: True
```

Add Beacon Configs to Minion

On the Salt minion, add the following configuration to `/etc/salt/minion.d/beacons.conf`:

```
beacons:
  inotify:
    - files:
        /etc/important_file:
          mask:
```

```
            - modify
    - disable_during_state_run: True
```

Save the configuration file and restart the minion service. The beacon is now set up to notify salt upon modifications made to the file.

Note: The `disable_during_state_run: True` parameter *prevents* the inotify beacon from generating reactor events due to salt itself modifying the file.

View Events on the Master

On your Salt master, start the event runner using the following command:

```
salt-run state.event pretty=true
```

This runner displays events as they are received by the master on the Salt event bus. To test the beacon you set up in the previous section, make and save a modification to `/etc/important_file`. You'll see an event similar to the following on the event bus:

```
{
 "_stamp": "2015-09-09T15:59:37.972753",
 "data": {
     "change": "IN_IGNORED",
     "id": "larry",
     "path": "/etc/important_file"
 },
 "tag": "salt/beacon/larry/inotify//etc/important_file"
}
```

This indicates that the event is being captured and sent correctly. Now you can create a reactor to take action when this event occurs.

Create a Reactor

This reactor reverts the file named `/etc/important_file` to the contents provided by salt each time it is modified.

Reactor SLS

On your Salt master, create a file named `/srv/reactor/revert.sls`.

Note: If the `/srv/reactor` directory doesn't exist, create it.

```
mkdir -p /srv/reactor
```

Add the following to `/srv/reactor/revert.sls`:

```
revert-file:
  local.state.apply:
    - tgt: {{ data['data']['id'] }}
```

```
    - arg:
      - maintain_important_file
```

Note: In addition to *setting* `disable_during_state_run: True` for an inotify beacon whose reaction is to modify the watched file, it is important to ensure the state applied is also *idempotent*.

Note: The expression `{{ data['data']['id'] }}` *is correct* as it matches the event structure *shown above*.

State SLS

Create the state sls file referenced by the reactor sls file. This state file will be located at `/srv/salt/maintain_important_file.sls`.

```
important_file:
  file.managed:
    - name: /etc/important_file
    - contents: |
        important_config: True
```

Master Config

Configure the master to map the inotify beacon event to the `revert` reaction in `/etc/salt/master.d/reactor.conf`:

```
reactor:
  - salt/beacon/*/inotify//etc/important_file:
    - /srv/reactor/revert.sls
```

Note: You can have only one top level `reactor` section, so if one already exists, add this code to the existing section. See *here* to learn more about reactor SLS syntax.

Start the Salt Master in Debug Mode

To help with troubleshooting, start the Salt master in debug mode:

```
service salt-master stop
salt-master -l debug
```

When debug logging is enabled, event and reactor data are displayed so you can discover syntax and other issues.

Trigger the Reactor

On your minion, make and save another change to `/etc/important_file`. On the Salt master, you'll see debug messages that indicate the event was received and the `state.apply` job was sent. When you inspect the file on the minion, you'll see that the file contents have been restored to `important_config: True`.

All beacons are configured using a similar process of enabling the beacon, writing a reactor SLS (and state SLS if needed), and mapping a beacon event to the reactor SLS.

8.2.3 Writing Beacon Plugins

Beacon plugins use the standard Salt loader system, meaning that many of the constructs from other plugin systems holds true, such as the `__virtual__` function.

The important function in the Beacon Plugin is the `beacon` function. When the beacon is configured to run, this function will be executed repeatedly by the minion. The `beacon` function therefore cannot block and should be as lightweight as possible. The `beacon` also must return a list of dicts, each dict in the list will be translated into an event on the master.

Beacons may also choose to implement a `__validate__` function which takes the beacon configuration as an argument and ensures that it is valid prior to continuing. This function is called automatically by the Salt loader when a beacon is loaded.

Please see the *inotify* beacon as an example.

The *beacon* Function

The beacons system will look for a function named *beacon* in the module. If this function is not present then the beacon will not be fired. This function is called on a regular basis and defaults to being called on every iteration of the minion, which can be tens to hundreds of times a second. This means that the *beacon* function cannot block and should not be CPU or IO intensive.

The beacon function will be passed in the configuration for the executed beacon. This makes it easy to establish a flexible configuration for each called beacon. This is also the preferred way to ingest the beacon's configuration as it allows for the configuration to be dynamically updated while the minion is running by configuring the beacon in the minion's pillar.

The Beacon Return

The information returned from the beacon is expected to follow a predefined structure. The returned value needs to be a list of dictionaries (standard python dictionaries are preferred, no ordered dicts are needed).

The dictionaries represent individual events to be fired on the minion and master event buses. Each dict is a single event. The dict can contain any arbitrary keys but the `tag' key will be extracted and added to the tag of the fired event.

The return data structure would look something like this:

```
[{'changes': ['/foo/bar'], 'tag': 'foo'},
 {'changes': ['/foo/baz'], 'tag': 'bar'}]
```

Calling Execution Modules

Execution modules are still the preferred location for all work and system interaction to happen in Salt. For this reason the *__salt__* variable is available inside the beacon.

Please be careful when calling functions in *__salt__*, while this is the preferred means of executing complicated routines in Salt not all of the execution modules have been written with beacons in mind. Watch out for execution modules that may be CPU intense or IO bound. Please feel free to add new execution modules and functions to back specific beacons.

Distributing Custom Beacons

Custom beacons can be distributed to minions using `saltutil`, see *Dynamic Module Distribution*.

8.3 Reactor System

Salt's Reactor system gives Salt the ability to trigger actions in response to an event. It is a simple interface to watching Salt's event bus for event tags that match a given pattern and then running one or more commands in response.

This system binds sls files to event tags on the master. These sls files then define reactions. This means that the reactor system has two parts. First, the reactor option needs to be set in the master configuration file. The reactor option allows for event tags to be associated with sls reaction files. Second, these reaction files use highdata (like the state system) to define reactions to be executed.

8.3.1 Event System

A basic understanding of the event system is required to understand reactors. The event system is a local ZeroMQ PUB interface which fires salt events. This event bus is an open system used for sending information notifying Salt and other systems about operations.

The event system fires events with a very specific criteria. Every event has a **tag**. Event tags allow for fast top-level filtering of events. In addition to the tag, each event has a data structure. This data structure is a dictionary, which contains information about the event.

8.3.2 Mapping Events to Reactor SLS Files

Reactor SLS files and event tags are associated in the master config file. By default this is /etc/salt/master, or /etc/salt/master.d/reactor.conf.

New in version 2014.7.0: Added Reactor support for `salt://` file paths.

In the master config section `reactor:' is a list of event tags to be matched and each event tag has a list of reactor SLS files to be run.

```
reactor:                            # Master config section "reactor"

  - 'salt/minion/*/start':          # Match tag "salt/minion/*/start"
    - /srv/reactor/start.sls        # Things to do when a minion starts
    - /srv/reactor/monitor.sls      # Other things to do

  - 'salt/cloud/*/destroyed':       # Globs can be used to match tags
    - /srv/reactor/destroy/*.sls    # Globs can be used to match file names

  - 'myco/custom/event/tag':        # React to custom event tags
    - salt://reactor/mycustom.sls   # Reactor files can come from the salt fileserver
```

Note: In the above example, `salt://reactor/mycustom.sls` refers to the `base` environment. To pull this file from a different environment, use the *querystring syntax* (e.g. `salt://reactor/mycustom.sls?saltenv=reactor`).

Reactor SLS files are similar to State and Pillar SLS files. They are by default YAML + Jinja templates and are passed familiar context variables. Click *here* for more detailed information on the variables available in Jinja templating.

Here is the SLS for a simple reaction:

```
{% if data['id'] == 'mysql1' %}
highstate_run:
  local.state.apply:
    - tgt: mysql1
{% endif %}
```

This simple reactor file uses Jinja to further refine the reaction to be made. If the `id` in the event data is `mysql1` (in other words, if the name of the minion is `mysql1`) then the following reaction is defined. The same data structure and compiler used for the state system is used for the reactor system. The only difference is that the data is matched up to the salt command API and the runner system. In this example, a command is published to the `mysql1` minion with a function of `state.apply`, which performs a *highstate*. Similarly, a runner can be called:

```
{% if data['data']['custom_var'] == 'runit' %}
call_runit_orch:
  runner.state.orchestrate:
    - args:
      - mods: orchestrate.runit
{% endif %}
```

This example will execute the state.orchestrate runner and intiate an execution of the `runit` orchestrator located at `/srv/salt/orchestrate/runit.sls`.

8.3.3 Types of Reactions

Name	Description
local	Runs a *remote-execution function* on targeted minions
runner	Executes a *runner function*
wheel	Executes a *wheel function* on the master
caller	Runs a *remote-execution function* on a masterless minion

Note: The `local` and `caller` reaction types will likely be renamed in a future release. These reaction types were named after Salt's internal client interfaces, and are not intuitively named. Both `local` and `caller` will continue to work in Reactor SLS files, however.

8.3.4 Where to Put Reactor SLS Files

Reactor SLS files can come both from files local to the master, and from any of backends enabled via the *`fileserver_backend`* config option. Files placed in the Salt fileserver can be referenced using a `salt://` URL, just like they can in State SLS files.

It is recommended to place reactor and orchestrator SLS files in their own uniquely-named subdirectories such as `orch/`, `orchestrate/`, `react/`, `reactor/`, etc., to keep them organized.

8.3.5 Writing Reactor SLS

The different reaction types were developed separately and have historically had different methods for passing arguments. For the 2017.7.2 release a new, unified configuration schema has been introduced, which applies to all reaction types.

The old config schema will continue to be supported, and there is no plan to deprecate it at this time.

Local Reactions

A `local` reaction runs a *remote-execution function* on the targeted minions.

The old config schema required the positional and keyword arguments to be manually separated by the user under `arg` and `kwarg` parameters. However, this is not very user-friendly, as it forces the user to distinguish which type of argument is which, and make sure that positional arguments are ordered properly. Therefore, the new config schema is recommended if the master is running a supported release.

The below two examples are equivalent:

<table>
<tr><th>Supported in 2017.7.2 and later</th><th>Supported in all releases</th></tr>
<tr><td><pre>install_zsh:
  local.state.single:
    - tgt: 'kernel:Linux'
    - tgt_type: grain
    - args:
      - fun: pkg.installed
      - name: zsh
      - fromrepo: updates</pre></td><td><pre>install_zsh:
  local.state.single:
    - tgt: 'kernel:Linux'
    - tgt_type: grain
    - arg:
      - pkg.installed
      - zsh
    - kwarg:
        fromrepo: updates</pre></td></tr>
</table>

This reaction would be equivalent to running the following Salt command:

```
salt -G 'kernel:Linux' state.single pkg.installed name=zsh fromrepo=updates
```

Note: Any other parameters in the *LocalClient().cmd_async()* method can be passed at the same indentation level as `tgt`.

Note: `tgt_type` is only required when the target expression defined in `tgt` uses a *target type* other than a minion ID glob.

The `tgt_type` argument was named `expr_form` in releases prior to 2017.7.0.

Runner Reactions

Runner reactions execute *runner functions* locally on the master.

The old config schema called for passing arguments to the reaction directly under the name of the runner function. However, this can cause unpredictable interactions with the Reactor system's internal arguments. It is also possible to pass positional and keyword arguments under `arg` and `kwarg` like above in *local reactions*, but as noted above this is not very user-friendly. Therefore, the new config schema is recommended if the master is running a supported release.

The below two examples are equivalent:

<table>
<tr><th>Supported in 2017.7.2 and later</th><th>Supported in all releases</th></tr>
<tr><td><pre>deploy_app:
  runner.state.orchestrate:
    - args:
      - mods: orchestrate.deploy_app
      - pillar:
          event_tag: {{ tag }}
          event_data: {{ data['data']|json }}</pre></td><td><pre>deploy_app:
  runner.state.orchestrate:
    - mods: orchestrate.deploy_app
    - kwarg:
        pillar:
          event_tag: {{ tag }}
          event_data: {{ data['data']|json }}</pre></td></tr>
</table>

Assuming that the event tag is `foo`, and the data passed to the event is `{'bar': 'baz'}`, then this reaction is equivalent to running the following Salt command:

```
salt-run state.orchestrate mods=orchestrate.deploy_app pillar='{"event_tag": "foo",
↪"event_data": {"bar": "baz"}}'
```

Wheel Reactions

Wheel reactions run *wheel functions* locally on the master.

Like *runner reactions*, the old config schema called for wheel reactions to have arguments passed directly under the name of the *wheel function* (or in `arg` or `kwarg` parameters).

The below two examples are equivalent:

<table>
<tr><th>Supported in 2017.7.2 and later</th><th>Supported in all releases</th></tr>
<tr><td><pre>remove_key:
  wheel.key.delete:
    - args:
      - match: {{ data['id'] }}</pre></td><td><pre>remove_key:
  wheel.key.delete:
    - match: {{ data['id'] }}</pre></td></tr>
</table>

Caller Reactions

Caller reactions run *remote-execution functions* on a minion daemon's Reactor system. To run a Reactor on the minion, it is necessary to configure the *Reactor Engine* in the minion config file, and then setup your watched events in a `reactor` section in the minion config file as well.

Note: Masterless Minions use this Reactor

This is the only way to run the Reactor if you use masterless minions.

Both the old and new config schemas involve passing arguments under an `args` parameter. However, the old config schema only supports positional arguments. Therefore, the new config schema is recommended if the masterless minion is running a supported release.

The below two examples are equivalent:

Supported in 2017.7.2 and later	Supported in all releases

`touch_file:`
`  caller.file.touch:`
`    - args:`
`      - name: /tmp/foo`
	`touch_file:`
	`  caller.file.touch:`
	`    - args:`
	`      - /tmp/foo`

This reaction is equivalent to running the following Salt command:

```
salt-call file.touch name=/tmp/foo
```

8.3.6 Best Practices for Writing Reactor SLS Files

The Reactor works as follows:

1. The Salt Reactor watches Salt's event bus for new events.
2. Each event's tag is matched against the list of event tags configured under the *reactor* section in the Salt Master config.
3. The SLS files for any matches are rendered into a data structure that represents one or more function calls.
4. That data structure is given to a pool of worker threads for execution.

Matching and rendering Reactor SLS files is done sequentially in a single process. For that reason, reactor SLS files should contain few individual reactions (one, if at all possible). Also, keep in mind that reactions are fired asynchronously (with the exception of *caller*) and do *not* support *requisites*.

Complex Jinja templating that calls out to slow *remote-execution* or *runner* functions slows down the rendering and causes other reactions to pile up behind the current one. The worker pool is designed to handle complex and long-running processes like *orchestration* jobs.

Therefore, when complex tasks are in order, *orchestration* is a natural fit. Orchestration SLS files can be more complex, and use requisites. Performing a complex task using orchestration lets the Reactor system fire off the orchestration job and proceed with processing other reactions.

8.3.7 Jinja Context

Reactor SLS files only have access to a minimal Jinja context. `grains` and `pillar` are *not* available. The `salt` object is available for calling *remote-execution* or *runner* functions, but it should be used sparingly and only for quick tasks for the reasons mentioned above.

In addition to the `salt` object, the following variables are available in the Jinja context:

- `tag` - the tag from the event that triggered execution of the Reactor SLS file
- `data` - the event's data dictionary

The `data` dict will contain an `id` key containing the minion ID, if the event was fired from a minion, and a `data` key containing the data passed to the event.

8.3.8 Advanced State System Capabilities

Reactor SLS files, by design, do not support *requisites*, ordering, `onlyif`/`unless` conditionals and most other powerful constructs from Salt's State system.

Complex Master-side operations are best performed by Salt's Orchestrate system so using the Reactor to kick off an Orchestrate run is a very common pairing.

For example:

```
# /etc/salt/master.d/reactor.conf
# A custom event containing: {"foo": "Foo!", "bar: "bar*", "baz": "Baz!"}
reactor:
  - my/custom/event:
    - /srv/reactor/some_event.sls
```

```
# /srv/reactor/some_event.sls
invoke_orchestrate_file:
  runner.state.orchestrate:
    - args:
        - mods: orchestrate.do_complex_thing
        - pillar:
            event_tag: {{ tag }}
            event_data: {{ data|json }}
```

```
# /srv/salt/orchestrate/do_complex_thing.sls
{% set tag = salt.pillar.get('event_tag') %}
{% set data = salt.pillar.get('event_data') %}

# Pass data from the event to a custom runner function.
# The function expects a 'foo' argument.
do_first_thing:
  salt.runner:
    - name: custom_runner.custom_function
    - foo: {{ data.foo }}

# Wait for the runner to finish then send an execution to minions.
# Forward some data from the event down to the minion's state run.
do_second_thing:
  salt.state:
    - tgt: {{ data.bar }}
    - sls:
      - do_thing_on_minion
    - kwarg:
        pillar:
          baz: {{ data.baz }}
    - require:
      - salt: do_first_thing
```

8.3.9 Beacons and Reactors

An event initiated by a beacon, when it arrives at the master will be wrapped inside a second event, such that the data object containing the beacon information will be `data['data']`, rather than `data`.

For example, to access the `id` field of the beacon event in a reactor file, you will need to reference `{{ data['data']['id'] }}` rather than `{{ data['id'] }}` as for events initiated directly on the event bus.

Similarly, the data dictionary attached to the event would be located in `{{ data['data']['data'] }}` instead of `{{ data['data'] }}`.

See the *beacon documentation* for examples.

8.3.10 Manually Firing an Event

From the Master

Use the *event.send* runner:

```
salt-run event.send foo '{orchestrate: refresh}'
```

From the Minion

To fire an event to the master from a minion, call *event.send*:

```
salt-call event.send foo '{orchestrate: refresh}'
```

To fire an event to the minion's local event bus, call *event.fire*:

```
salt-call event.fire '{orchestrate: refresh}' foo
```

Referencing Data Passed in Events

Assuming any of the above examples, any reactor SLS files triggered by watching the event tag `foo` will execute with `{{ data['data']['orchestrate'] }}` equal to `'refresh'`.

8.3.11 Getting Information About Events

The best way to see exactly what events have been fired and what data is available in each event is to use the *state.event runner*.

See also:

Common Salt Events

Example usage:

```
salt-run state.event pretty=True
```

Example output:

```
salt/job/20150213001905721678/new       {
    "_stamp": "2015-02-13T00:19:05.724583",
    "arg": [],
    "fun": "test.ping",
    "jid": "20150213001905721678",
    "minions": [
        "jerry"
    ],
    "tgt": "*",
    "tgt_type": "glob",
    "user": "root"
}
salt/job/20150213001910749506/ret/jerry {
    "_stamp": "2015-02-13T00:19:11.136730",
    "cmd": "_return",
    "fun": "saltutil.find_job",
```

```
    "fun_args": [
        "20150213001905721678"
    ],
    "id": "jerry",
    "jid": "20150213001910749506",
    "retcode": 0,
    "return": {},
    "success": true
}
```


8.3.12 Debugging the Reactor

The best window into the Reactor is to run the master in the foreground with debug logging enabled. The output will include when the master sees the event, what the master does in response to that event, and it will also include the rendered SLS file (or any errors generated while rendering the SLS file).

1. Stop the master.
2. Start the master manually:

```
salt-master -l debug
```

3. Look for log entries in the form:

```
[DEBUG   ] Gathering reactors for tag foo/bar
[DEBUG   ] Compiling reactions for tag foo/bar
[DEBUG   ] Rendered data from file: /path/to/the/reactor_file.sls:
<... Rendered output appears here. ...>
```

The rendered output is the result of the Jinja parsing and is a good way to view the result of referencing Jinja variables. If the result is empty then Jinja produced an empty result and the Reactor will ignore it.

Passing Event Data to Minions or Orchestration as Pillar

An interesting trick to pass data from the Reactor SLS file to `state.apply` is to pass it as inline Pillar data since both functions take a keyword argument named `pillar`.

The following example uses Salt's Reactor to listen for the event that is fired when the key for a new minion is accepted on the master using `salt-key`.

`/etc/salt/master.d/reactor.conf`:

```
reactor:
  - 'salt/key':
    - /srv/salt/haproxy/react_new_minion.sls
```

The Reactor then fires a :`state.apply` command targeted to the HAProxy servers and passes the ID of the new minion from the event to the state file via inline Pillar.

`/srv/salt/haproxy/react_new_minion.sls`:

```
{% if data['act'] == 'accept' and data['id'].startswith('web') %}
add_new_minion_to_pool:
  local.state.apply:
    - tgt: 'haproxy*'
    - args:
```

```
      - mods: haproxy.refresh_pool
      - pillar:
          new_minion: {{ data['id'] }}
{% endif %}
```

The above command is equivalent to the following command at the CLI:

```
salt 'haproxy*' state.apply haproxy.refresh_pool pillar='{new_minion: minionid}'
```

This works with Orchestrate files as well:

```
call_some_orchestrate_file:
  runner.state.orchestrate:
    - args:
      - mods: orchestrate.some_orchestrate_file
      - pillar:
          stuff: things
```

Which is equivalent to the following command at the CLI:

```
salt-run state.orchestrate orchestrate.some_orchestrate_file pillar='{stuff: things}'
```

Finally, that data is available in the state file using the normal Pillar lookup syntax. The following example is grabbing web server names and IP addresses from *Salt Mine*. If this state is invoked from the Reactor then the custom Pillar value from above will be available and the new minion will be added to the pool but with the `disabled` flag so that HAProxy won't yet direct traffic to it.

`/srv/salt/haproxy/refresh_pool.sls`:

```
{% set new_minion = salt['pillar.get']('new_minion') %}

listen web *:80
    balance source
    {% for server,ip in salt['mine.get']('web*', 'network.interfaces', ['eth0']).
↪items() %}
    {% if server == new_minion %}
    server {{ server }} {{ ip }}:80 disabled
    {% else %}
    server {{ server }} {{ ip }}:80 check
    {% endif %}
    {% endfor %}
```

8.3.13 A Complete Example

In this example, we're going to assume that we have a group of servers that will come online at random and need to have keys automatically accepted. We'll also add that we don't want all servers being automatically accepted. For this example, we'll assume that all hosts that have an id that starts with `ink' will be automatically accepted and have `state.apply` executed. On top of this, we're going to add that a host coming up that was replaced (meaning a new key) will also be accepted.

Our master configuration will be rather simple. All minions that attempte to authenticate will match the **tag** of **salt/auth**. When it comes to the minion key being accepted, we get a more refined **tag** that includes the minion id, which we can use for matching.

`/etc/salt/master.d/reactor.conf`:

```
reactor:
  - 'salt/auth':
    - /srv/reactor/auth-pending.sls
  - 'salt/minion/ink*/start':
    - /srv/reactor/auth-complete.sls
```

In this SLS file, we say that if the key was rejected we will delete the key on the master and then also tell the master to ssh in to the minion and tell it to restart the minion, since a minion process will die if the key is rejected.

We also say that if the key is pending and the id starts with ink we will accept the key. A minion that is waiting on a pending key will retry authentication every ten seconds by default.

`/srv/reactor/auth-pending.sls`:

```
{# Ink server failed to authenticate -- remove accepted key #}
{% if not data['result'] and data['id'].startswith('ink') %}
minion_remove:
  wheel.key.delete:
    - args:
      - match: {{ data['id'] }}
minion_rejoin:
  local.cmd.run:
    - tgt: salt-master.domain.tld
    - args:
      - cmd: ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no "{{ data[
↪'id'] }}" 'sleep 10 && /etc/init.d/salt-minion restart'
{% endif %}

{# Ink server is sending new key -- accept this key #}
{% if 'act' in data and data['act'] == 'pend' and data['id'].startswith('ink') %}
minion_add:
  wheel.key.accept:
    - args:
      - match: {{ data['id'] }}
{% endif %}
```

No if statements are needed here because we already limited this action to just Ink servers in the master configuration.

`/srv/reactor/auth-complete.sls`:

```
{# When an Ink server connects, run state.apply. #}
highstate_run:
  local.state.apply:
    - tgt: {{ data['id'] }}
    - ret: smtp
```

The above will also return the *highstate* result data using the *smtp_return* returner (use virtualname like when using from the command line with *--return*). The returner needs to be configured on the minion for this to work. See *salt.returners.smtp_return* documentation for that.

8.3.14 Syncing Custom Types on Minion Start

Salt will sync all custom types (by running a *saltutil.sync_all*) on every *highstate*. However, there is a chicken-and-egg issue where, on the initial *highstate*, a minion will not yet have these custom types synced when the top file is first compiled. This can be worked around with a simple reactor which watches for `minion_start` events, which each minion fires when it first starts up and connects to the master.

On the master, create **/srv/reactor/sync_grains.sls** with the following contents:

```
sync_grains:
  local.saltutil.sync_grains:
    - tgt: {{ data['id'] }}
```

And in the master config file, add the following reactor configuration:

```
reactor:
  - 'salt/minion/*/start':
    - /srv/reactor/sync_grains.sls
```

This will cause the master to instruct each minion to sync its custom grains when it starts, making these grains available when the initial *highstate* is executed.

Other types can be synced by replacing `local.saltutil.sync_grains` with `local.saltutil.sync_modules`, `local.saltutil.sync_all`, or whatever else suits the intended use case.

Also, if it is not desirable that *every* minion syncs on startup, the `*` can be replaced with a different glob to narrow down the set of minions which will match that reactor (e.g. `salt/minion/appsrv*/start`, which would only match minion IDs beginning with `appsrv`).

CHAPTER 9

Orchestration

9.1 Orchestrate Runner

Executing states or highstate on a minion is perfect when you want to ensure that minion configured and running the way you want. Sometimes however you want to configure a set of minions all at once.

For example, if you want to set up a load balancer in front of a cluster of web servers you can ensure the load balancer is set up first, and then the same matching configuration is applied consistently across the whole cluster.

Orchestration is the way to do this.

9.1.1 The Orchestrate Runner

New in version 0.17.0.

Note: Orchestrate Deprecates OverState

The Orchestrate Runner (originally called the state.sls runner) offers all the functionality of the OverState, but with some advantages:

- All *Requisites and Other Global State Arguments* available in states can be used.
- The states/functions will also work on salt-ssh minions.

The Orchestrate Runner replaced the OverState system in Salt 2015.8.0.

The orchestrate runner generalizes the Salt state system to a Salt master context. Whereas the `state.sls`, `state.highstate`, et al. functions are concurrently and independently executed on each Salt minion, the `state.orchestrate` runner is executed on the master, giving it a master-level view and control over requisites, such as state ordering and conditionals. This allows for inter minion requisites, like ordering the application of states on different minions that must not happen simultaneously, or for halting the state run on all minions if a minion fails one of its states.

The `state.sls`, `state.highstate`, et al. functions allow you to statefully manage each minion and the `state.orchestrate` runner allows you to statefully manage your entire infrastructure.

Writing SLS Files

Orchestrate SLS files are stored in the same location as State SLS files. This means that both `file_roots` and `gitfs_remotes` impact what SLS files are available to the reactor and orchestrator.

It is recommended to keep reactor and orchestrator SLS files in their own uniquely named subdirectories such as `_orch/`, `orch/`, `_orchestrate/`, `react/`, `_reactor/`, etc. This will avoid duplicate naming and will help prevent confusion.

Executing the Orchestrate Runner

The Orchestrate Runner command format is the same as for the `state.sls` function, except that since it is a runner, it is executed with `salt-run` rather than `salt`. Assuming you have a state.sls file called `/srv/salt/orch/webserver.sls` the following command, run on the master, will apply the states defined in that file.

```
salt-run state.orchestrate orch.webserver
```

Note: `state.orch` is a synonym for `state.orchestrate`

Changed in version 2014.1.1: The runner function was renamed to `state.orchestrate` to avoid confusion with the *state.sls* execution function. In versions 0.17.0 through 2014.1.0, `state.sls` must be used.

Masterless Orchestration

New in version 2016.11.0.

To support salt orchestration on masterless minions, the Orchestrate Runner is available as an execution module. The syntax for masterless orchestration is exactly the same, but it uses the `salt-call` command and the minion configuration must contain the `file_mode: local` option. Alternatively, use `salt-call --local` on the command line.

```
salt-call --local state.orchestrate orch.webserver
```

Note: Masterless orchestration supports only the `salt.state` command in an sls file; it does not (currently) support the `salt.function` command.

Examples

Function

To execute a function, use *salt.function*:

```
# /srv/salt/orch/cleanfoo.sls
cmd.run:
  salt.function:
    - tgt: '*'
    - arg:
      - rm -rf /tmp/foo
```

```
salt-run state.orchestrate orch.cleanfoo
```

If you omit the ``name'' argument, the ID of the state will be the default name, or in the case of `salt.function`, the execution module function to run. You can specify the ``name'' argument to avoid conflicting IDs:

```
copy_some_file:
  salt.function:
    - name: file.copy
    - tgt: '*'
    - arg:
      - /path/to/file
      - /tmp/copy_of_file
    - kwarg:
        remove_existing: true
```

Fail Functions

When running a remote execution function in orchestration, certain return values for those functions may indicate failure, while the function itself doesn't set a return code. For those circumstances, using a ``fail function'' allows for a more flexible means of assessing success or failure.

A fail function can be written as part of a *custom execution module*. The function should accept one argument, and return a boolean result. For example:

```
def check_func_result(retval):
    if some_condition:
        return True
    else:
        return False
```

The function can then be referenced in orchestration SLS like so:

```
do_stuff:
  salt.function:
    - name: modname.funcname
    - tgt: '*'
    - fail_function: mymod.check_func_result
```

Important: Fail functions run *on the master*, so they must be synced using `salt-run saltutil.sync_modules`.

State

To execute a state, use `salt.state`.

```
# /srv/salt/orch/webserver.sls
install_nginx:
  salt.state:
    - tgt: 'web*'
    - sls:
      - nginx
```

```
salt-run state.orchestrate orch.webserver
```

Highstate

To run a highstate, set `highstate: True` in your state config:

```
# /srv/salt/orch/web_setup.sls
webserver_setup:
  salt.state:
    - tgt: 'web*'
    - highstate: True
```

```
salt-run state.orchestrate orch.web_setup
```

Runner

To execute another runner, use *salt.runner*. For example to use the `cloud.profile` runner in your orchestration state additional options to replace values in the configured profile, use this:

```
# /srv/salt/orch/deploy.sls
create_instance:
  salt.runner:
    - name: cloud.profile
    - prof: cloud-centos
    - provider: cloud
    - instances:
      - server1
    - opts:
        minion:
          master: master1
```

To get a more dynamic state, use jinja variables together with `inline pillar data`. Using the same example but passing on pillar data, the state would be like this.

```
# /srv/salt/orch/deploy.sls
{% set servers = salt['pillar.get']('servers', 'test') %}
{% set master = salt['pillar.get']('master', 'salt') %}
create_instance:
  salt.runner:
    - name: cloud.profile
    - prof: cloud-centos
    - provider: cloud
    - instances:
      - {{ servers }}
    - opts:
        minion:
          master: {{ master }}
```

To execute with pillar data.

```
salt-run state.orch orch.deploy pillar='{"servers": "newsystem1",
"master": "mymaster"}'
```

Return Codes in Runner/Wheel Jobs

New in version 2018.3.0.

State (`salt.state`) jobs are able to report failure via the *state return dictionary*. Remote execution (`salt.function`) jobs are able to report failure by setting a `retcode` key in the `__context__` dictionary. However, runner (`salt.runner`) and wheel (`salt.wheel`) jobs would only report a `False` result when the runner/wheel function raised an exception. As of the 2018.3.0 release, it is now possible to set a retcode in runner and wheel functions just as you can do in remote execution functions. Here is some example pseudocode:

```
def myrunner():
    ...
    do stuff
    ...
    if some_error_condition:
        __context__['retcode'] = 1
    return result
```

This allows a custom runner/wheel function to report its failure so that requisites can accurately tell that a job has failed.

More Complex Orchestration

Many states/functions can be configured in a single file, which when combined with the full suite of *Requisites and Other Global State Arguments*, can be used to easily configure complex orchestration tasks. Additionally, the states/functions will be executed in the order in which they are defined, unless prevented from doing so by any *Requisites and Other Global State Arguments*, as is the default in SLS files since 0.17.0.

```
bootstrap_servers:
  salt.function:
    - name: cmd.run
    - tgt: 10.0.0.0/24
    - tgt_type: ipcidr
    - arg:
      - bootstrap

storage_setup:
  salt.state:
    - tgt: 'role:storage'
    - tgt_type: grain
    - sls: ceph
    - require:
      - salt: webserver_setup

webserver_setup:
  salt.state:
    - tgt: 'web*'
    - highstate: True
```

Given the above setup, the orchestration will be carried out as follows:

1. The shell command `bootstrap` will be executed on all minions in the 10.0.0.0/24 subnet.
2. A Highstate will be run on all minions whose ID starts with ``web'', since the `storage_setup` state requires it.
3. Finally, the `ceph` SLS target will be executed on all minions which have a grain called `role` with a value of `storage`.

Note: Remember, salt-run is *always* executed on the master.

9.1.2 Parsing Results Programmatically

Orchestration jobs return output in a specific data structure. That data structure is represented differently depending on the outputter used. With the default outputter for orchestration, you get a nice human-readable output. Assume the following orchestration SLS:

```
good_state:
  salt.state:
    - tgt: myminion
    - sls:
    - succeed_with_changes

bad_state:
  salt.state:
    - tgt: myminion
    - sls:
    - fail_with_changes

mymod.myfunc:
  salt.function:
    - tgt: myminion

mymod.myfunc_false_result:
  salt.function:
    - tgt: myminion
```

Running this using the default outputter would produce output which looks like this:

```
fa5944a73aa8_master:
----------
          ID: good_state
    Function: salt.state
      Result: True
     Comment: States ran successfully. Updating myminion.
     Started: 21:08:02.681604
    Duration: 265.565 ms
     Changes:
              myminion:
              ----------
                        ID: test succeed with changes
                  Function: test.succeed_with_changes
                    Result: True
                   Comment: Success!
                   Started: 21:08:02.835893
                  Duration: 0.375 ms
                   Changes:
                            ----------
                            testing:
                                ----------
                                new:
                                    Something pretended to change
                                old:
                                    Unchanged

              Summary for myminion
              ------------
              Succeeded: 1 (changed=1)
              Failed:    0
```

```
              ------------
              Total states run:     1
              Total run time:   0.375 ms
----------
          ID: bad_state
    Function: salt.state
      Result: False
     Comment: Run failed on minions: myminion
     Started: 21:08:02.947702
    Duration: 177.01 ms
     Changes:
              myminion:
              ----------
                        ID: test fail with changes
                  Function: test.fail_with_changes
                    Result: False
                   Comment: Failure!
                   Started: 21:08:03.116634
                  Duration: 0.502 ms
                   Changes:
                            ----------
                            testing:
                                ----------
                                new:
                                    Something pretended to change
                                old:
                                    Unchanged

              Summary for myminion
              ------------
              Succeeded: 0 (changed=1)
              Failed:    1
              ------------
              Total states run:     1
              Total run time:   0.502 ms
----------
          ID: mymod.myfunc
    Function: salt.function
      Result: True
     Comment: Function ran successfully. Function mymod.myfunc ran on myminion.
     Started: 21:08:03.125011
    Duration: 159.488 ms
     Changes:
              myminion:
                  True
----------
          ID: mymod.myfunc_false_result
    Function: salt.function
      Result: False
     Comment: Running function mymod.myfunc_false_result failed on minions: myminion.
 ↪Function mymod.myfunc_false_result ran on myminion.
     Started: 21:08:03.285148
    Duration: 176.787 ms
     Changes:
              myminion:
                  False

Summary for fa5944a73aa8_master
```

```
-------------
Succeeded: 2 (changed=4)
Failed:    2
-------------
Total states run:     4
Total run time: 778.850 ms
```

However, using the `json` outputter, you can get the output in an easily loadable and parsable format:

```
salt-run state.orchestrate test --out=json
```



```
{
    "outputter": "highstate",
    "data": {
        "fa5944a73aa8_master": {
            "salt_|-good_state_|-good_state_|-state": {
                "comment": "States ran successfully. Updating myminion.",
                "name": "good_state",
                "start_time": "21:35:16.868345",
                "result": true,
                "duration": 267.299,
                "__run_num__": 0,
                "__jid__": "20171130213516897392",
                "__sls__": "test",
                "changes": {
                    "ret": {
                        "myminion": {
                            "test_|-test succeed with changes_|-test succeed with
↪changes_|-succeed_with_changes": {
                                "comment": "Success!",
                                "name": "test succeed with changes",
                                "start_time": "21:35:17.022592",
                                "result": true,
                                "duration": 0.362,
                                "__run_num__": 0,
                                "__sls__": "succeed_with_changes",
                                "changes": {
                                    "testing": {
                                        "new": "Something pretended to change",
                                        "old": "Unchanged"
                                    }
                                },
                                "__id__": "test succeed with changes"
                            }
                        }
                    },
                    "out": "highstate"
                },
                "__id__": "good_state"
            },
            "salt_|-bad_state_|-bad_state_|-state": {
                "comment": "Run failed on minions: test",
                "name": "bad_state",
                "start_time": "21:35:17.136511",
                "result": false,
                "duration": 197.635,
                "__run_num__": 1,
                "__jid__": "20171130213517202203",
```

```
                "__sls__": "test",
                "changes": {
                    "ret": {
                        "myminion": {
                            "test_|-test fail with changes_|-test fail with changes_|-
↪fail_with_changes": {
                                "comment": "Failure!",
                                "name": "test fail with changes",
                                "start_time": "21:35:17.326268",
                                "result": false,
                                "duration": 0.509,
                                "__run_num__": 0,
                                "__sls__": "fail_with_changes",
                                "changes": {
                                    "testing": {
                                        "new": "Something pretended to change",
                                        "old": "Unchanged"
                                    }
                                },
                                "__id__": "test fail with changes"
                            }
                        }
                    },
                    "out": "highstate"
                },
                "__id__": "bad_state"
            },
            "salt_|-mymod.myfunc_|-mymod.myfunc_|-function": {
                "comment": "Function ran successfully. Function mymod.myfunc ran on
↪myminion.",
                "name": "mymod.myfunc",
                "start_time": "21:35:17.334373",
                "result": true,
                "duration": 151.716,
                "__run_num__": 2,
                "__jid__": "20171130213517361706",
                "__sls__": "test",
                "changes": {
                    "ret": {
                        "myminion": true
                    },
                    "out": "highstate"
                },
                "__id__": "mymod.myfunc"
            },
            "salt_|-mymod.myfunc_false_result-mymod.myfunc_false_result-function": {
                "comment": "Running function mymod.myfunc_false_result failed on
↪minions: myminion. Function mymod.myfunc_false_result ran on myminion.",
                "name": "mymod.myfunc_false_result",
                "start_time": "21:35:17.486625",
                "result": false,
                "duration": 174.241,
                "__run_num__": 3,
                "__jid__": "20171130213517536270",
                "__sls__": "test",
                "changes": {
                    "ret": {
                        "myminion": false
```

```
                    },
                    "out": "highstate"
                },
                "__id__": "mymod.myfunc_false_result"
            }
        }
    },
    "retcode": 1
}
```


The 2018.3.0 release includes a couple fixes to make parsing this data easier and more accurate. The first is the ability to set a *return code* in a custom runner or wheel function, as noted above. The second is a change to how failures are included in the return data. Prior to the 2018.3.0 release, minions that failed a `salt.state` orchestration job would show up in the `comment` field of the return data, in a human-readable string that was not easily parsed. They are now included in the `changes` dictionary alongside the minions that succeeded. In addition, `salt.function` jobs which failed because the *fail function* returned `False` used to handle their failures in the same way `salt.state` jobs did, and this has likewise been corrected.

Running States on the Master without a Minion

The orchestrate runner can be used to execute states on the master without using a minion. For example, assume that `salt://foo.sls` contains the following SLS:

```
/etc/foo.conf:
  file.managed:
    - source: salt://files/foo.conf
    - mode: 0600
```

In this case, running `salt-run state.orchestrate foo` would be the equivalent of running a `state.sls foo`, but it would execute on the master only, and would not require a minion daemon to be running on the master.

This is not technically orchestration, but it can be useful in certain use cases.

Limitations

Only one SLS target can be run at a time using this method, while using *state.sls* allows for multiple SLS files to be passed in a comma-separated list.

CHAPTER 10

Solaris

This section contains details on Solaris specific quirks and workarounds.

Note: Solaris refers to both Solaris 10 comaptible platforms like Solaris 10, illumos, SmartOS, OmniOS, OpenIndiana,... and Oracle Solaris 11 platforms.

10.1 Solaris-specific Behaviour

Salt is capable of managing Solaris systems, however due to various differences between the operating systems, there are some things you need to keep in mind.

This document will contain any quirks that apply across Salt or limitations in some modules.

10.1.1 FQDN/UQDN

On Solaris platforms the FQDN will not always be properly detected. If an IPv6 address is configured pythons `` `socket.getfqdn()` `` fails to return a FQDN and returns the nodename instead. For a full breakdown see the following issue on github: #37027

10.1.2 Grains

Not all grains are available or some have empty or 0 as value. Mostly grains that are depenend on hardware discovery like: - num_gpus - gpus

Also some resolver related grains like: - domain - dns:options - dns:sortlist

CHAPTER 11

Salt SSH

11.1 Getting Started

Salt SSH is very easy to use, simply set up a basic *roster* file of the systems to connect to and run `salt-ssh` commands in a similar way as standard `salt` commands.

- Salt ssh is considered production ready in version 2014.7.0
- Python is required on the remote system (unless using the `-r` option to send raw ssh commands)
- On many systems, the `salt-ssh` executable will be in its own package, usually named `salt-ssh`
- The Salt SSH system does not supersede the standard Salt communication systems, it simply offers an SSH-based alternative that does not require ZeroMQ and a remote agent. Be aware that since all communication with Salt SSH is executed via SSH it is substantially slower than standard Salt with ZeroMQ.
- At the moment fileserver operations must be wrapped to ensure that the relevant files are delivered with the `salt-ssh` commands. The state module is an exception, which compiles the state run on the master, and in the process finds all the references to `salt://` paths and copies those files down in the same tarball as the state run. However, needed fileserver wrappers are still under development.

11.2 Salt SSH Roster

The roster system in Salt allows for remote minions to be easily defined.

Note: See the *SSH roster docs* for more details.

Simply create the roster file, the default location is */etc/salt/roster*:

```
web1: 192.168.42.1
```

This is a very basic roster file where a Salt ID is being assigned to an IP address. A more elaborate roster can be created:

```
web1:
  host: 192.168.42.1 # The IP addr or DNS hostname
  user: fred         # Remote executions will be executed as user fred
  passwd: foobarbaz  # The password to use for login, if omitted, keys are used
  sudo: True         # Whether to sudo to root, not enabled by default
```

```
web2:
  host: 192.168.42.2
```

Note: sudo works only if NOPASSWD is set for user in /etc/sudoers: `fred ALL=(ALL) NOPASSWD: ALL`

11.3 Deploy ssh key for salt-ssh

By default, salt-ssh will generate key pairs for ssh, the default path will be `/etc/salt/pki/master/ssh/salt-ssh.rsa`. The key generation happens when you run `salt-ssh` for the first time.

You can use ssh-copy-id, (the OpenSSH key deployment tool) to deploy keys to your servers.

```
ssh-copy-id -i /etc/salt/pki/master/ssh/salt-ssh.rsa.pub user@server.demo.com
```

One could also create a simple shell script, named salt-ssh-copy-id.sh as follows:

```
#!/bin/bash
if [ -z $1 ]; then
   echo $0 user@host.com
   exit 0
fi
ssh-copy-id -i /etc/salt/pki/master/ssh/salt-ssh.rsa.pub $1
```

Note: Be certain to chmod +x salt-ssh-copy-id.sh.

```
./salt-ssh-copy-id.sh user@server1.host.com
./salt-ssh-copy-id.sh user@server2.host.com
```

Once keys are successfully deployed, salt-ssh can be used to control them.

Alternatively ssh agent forwarding can be used by setting the priv to agent-forwarding.

11.4 Calling Salt SSH

Note: `salt-ssh` on RHEL/CentOS 5

The `salt-ssh` command requires at least python 2.6, which is not installed by default on RHEL/CentOS 5. An easy workaround in this situation is to use the `-r` option to run a raw shell command that installs python26:

```
salt-ssh centos-5-minion -r 'yum -y install epel-release ; yum -y install python26'
```

Note: `salt-ssh` on systems with Python 3.x

Salt, before the 2017.7.0 release, does not support Python 3.x which is the default on for example the popular 16.04 LTS release of Ubuntu. An easy workaround for this scenario is to use the `-r` option similar to the example above:

```
salt-ssh ubuntu-1604-minion -r 'apt update ; apt install -y python-minimal'
```

The `salt-ssh` command can be easily executed in the same way as a salt command:

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

Commands with `salt-ssh` follow the same syntax as the `salt` command.

The standard salt functions are available! The output is the same as `salt` and many of the same flags are available. Please see http://docs.saltstack.com/ref/cli/salt-ssh.html for all of the available options.

11.4.1 Raw Shell Calls

By default `salt-ssh` runs Salt execution modules on the remote system, but `salt-ssh` can also execute raw shell commands:

```
salt-ssh '*' -r 'ifconfig'
```

11.5 States Via Salt SSH

The Salt State system can also be used with `salt-ssh`. The state system abstracts the same interface to the user in `salt-ssh` as it does when using standard `salt`. The intent is that Salt Formulas defined for standard `salt` will work seamlessly with `salt-ssh` and vice-versa.

The standard Salt States walkthroughs function by simply replacing `salt` commands with `salt-ssh`.

11.6 Targeting with Salt SSH

Due to the fact that the targeting approach differs in salt-ssh, only glob and regex targets are supported as of this writing, the remaining target systems still need to be implemented.

Note: By default, Grains are settable through `salt-ssh`. By default, these grains will *not* be persisted across reboots.

See the ``thin_dir'' setting in *Roster documentation* for more details.

11.7 Configuring Salt SSH

Salt SSH takes its configuration from a master configuration file. Normally, this file is in `/etc/salt/master`. If one wishes to use a customized configuration file, the `-c` option to Salt SSH facilitates passing in a directory to look inside for a configuration file named `master`.

11.7.1 Minion Config

New in version 2015.5.1.

Minion config options can be defined globally using the master configuration option `ssh_minion_opts`. It can also be defined on a per-minion basis with the `minion_opts` entry in the roster.

11.8 Running Salt SSH as non-root user

By default, Salt read all the configuration from /etc/salt/. If you are running Salt SSH with a regular user you have to modify some paths or you will get ``Permission denied'' messages. You have to modify two parameters: `pki_dir` and `cachedir`. Those should point to a full path writable for the user.

It's recommended not to modify /etc/salt for this purpose. Create a private copy of /etc/salt for the user and run the command with `-c /new/config/path`.

11.9 Define CLI Options with Saltfile

If you are commonly passing in CLI options to `salt-ssh`, you can create a `Saltfile` to automatically use these options. This is common if you're managing several different salt projects on the same server.

So you can `cd` into a directory that has a `Saltfile` with the following YAML contents:

```
salt-ssh:
  config_dir: path/to/config/dir
  ssh_max_procs: 30
  ssh_wipe: True
```

Instead of having to call `salt-ssh --config-dir=path/to/config/dir --max-procs=30 --wipe * test.ping` you can call `salt-ssh * test.ping`.

Boolean-style options should be specified in their YAML representation.

Note: The option keys specified must match the destination attributes for the options specified in the parser `salt.utils.parsers.SaltSSHOptionParser`. For example, in the case of the `--wipe` command line option, its `dest` is configured to be `ssh_wipe` and thus this is what should be configured in the `Saltfile`. Using the names of flags for this option, being `wipe: True` or `w: True`, will not work.

Note: For the *Saltfile* to be automatically detected it needs to be named *Saltfile* with a capital *S* and be readable by the user running salt-ssh.

At last you can create `~/.salt/Saltfile` and `salt-ssh` will automatically load it by default.

11.10 Debugging salt-ssh

One common approach for debugging `salt-ssh` is to simply use the tarball that salt ships to the remote machine and call `salt-call` directly.

To determine the location of `salt-call`, simply run `salt-ssh` with the `-ltrace` flag and look for a line containing the string, SALT_ARGV. This contains the `salt-call` command that `salt-ssh` attempted to execute.

It is recommended that one modify this command a bit by removing the `-l quiet`, `--metadata` and `--output json` to get a better idea of what's going on the target system.

11.10.1 Salt Rosters

Salt rosters are pluggable systems added in Salt 0.17.0 to facilitate the `salt-ssh` system. The roster system was created because `salt-ssh` needs a means to identify which systems need to be targeted for execution.

See also:

roster modules

Note: The Roster System is not needed or used in standard Salt because the master does not need to be initially aware of target systems, since the Salt Minion checks itself into the master.

Since the roster system is pluggable, it can be easily augmented to attach to any existing systems to gather information about what servers are presently available and should be attached to by `salt-ssh`. By default the roster file is located at /etc/salt/roster.

How Rosters Work

The roster system compiles a data structure internally referred to as `targets`. The `targets` is a list of target systems and attributes about how to connect to said systems. The only requirement for a roster module in Salt is to return the `targets` data structure.

Targets Data

The information which can be stored in a roster `target` is the following:

```
<Salt ID>:       # The id to reference the target system with
    host:        # The IP address or DNS name of the remote host
    user:        # The user to log in as
    passwd:      # The password to log in with

    # Optional parameters
    port:        # The target system's ssh port number
    sudo:        # Boolean to run command via sudo
    sudo_user:   # Str: Set this to execute Salt as a sudo user other than root.
                 # This user must be in the same system group as the remote user
                 # that is used to login and is specified above. Alternatively,
                 # the user must be a super-user.
    tty:         # Boolean: Set this option to True if sudo is also set to
                 # True and requiretty is also set on the target system
    priv:        # File path to ssh private key, defaults to salt-ssh.rsa
                 # The priv can also be set to agent-forwarding to not specify
                 # a key, but use ssh agent forwarding
    timeout:     # Number of seconds to wait for response when establishing
                 # an SSH connection
    minion_opts: # Dictionary of minion opts
    thin_dir:    # The target system's storage directory for Salt
                 # components. Defaults to /tmp/salt-<hash>.
```

```
    cmd_umask:    # umask to enforce for the salt-call command. Should be in
                  # octal (so for 0o077 in YAML you would do 0077, or 63)
```

Target Defaults

The *roster_defaults* dictionary in the master config is used to set the default login variables for minions in the roster so that the same arguments do not need to be passed with commandline arguments.

```
roster_defaults:
  user: daniel
  sudo: True
  priv: /root/.ssh/id_rsa
  tty: True
```

thin_dir

Salt needs to upload a standalone environment to the target system, and this defaults to /tmp/salt-<hash>. This directory will be cleaned up per normal systems operation.

If you need a persistent Salt environment, for instance to set persistent grains, this value will need to be changed.

CHAPTER 12

Thorium Complex Reactor

Note: Thorium is a provisional feature of Salt and is subject to change and removal if the feature proves to not be a viable solution.

Note: Thorium was added to Salt as an experimental feature in the 2016.3.0 release, as of 2016.3.0 this feature is considered experimental, no guarantees are made for support of any kind yet.

The original Salt Reactor is based on the idea of listening for a specific event and then reacting to it. This model comes with many logical limitations, for instance it is very difficult (and hacky) to fire a reaction based on aggregate data or based on multiple events.

The Thorium reactor is intended to alleviate this problem in a very elegant way. Instead of using extensive jinja routines or complex python sls files the aggregation of data and the determination of what should run becomes isolated to the sls data logic, makes the definitions much cleaner.

12.1 Starting the Thorium Engine

To enable the thorium engine add the following configuration to the engines section of your Salt Master or Minion configuration file and restart the daemon:

```
engines:
  - thorium: {}
```

12.2 Thorium Modules

Because of its specialized nature, Thorium uses its own set of modules. However, many of these modules are designed to wrap the more commonly-used Salt subsystems. These modules are:

- local: Execution modules
- runner: Runner modules
- wheel: Wheel modules

There are other modules that ship with Thorium as well. Some of these will be highlighted later in this document.

12.3 Writing Thorium Formulas

Like some other Salt subsystems, Thorium uses its own directory structure. The default location for this structure is /srv/thorium/, but it can be changed using the thorium_roots setting in the master configuration file.

This would explicitly set the roots to the default:

```
thorium_roots:
  base:
    - /srv/thorium
```

Example thorium_roots configuration:

```
thorium_roots:
  base:
    - /etc/salt/thorium
```

12.3.1 The Thorium top.sls File

Thorium uses its own top.sls file, which follows the same convention as is found in /srv/salt/:

```
<srv>:
  <target>:
    - <formula 1>
    - <formula 2>
    - <etc...>
```

For instance, a top.sls using a standard base environment and a single Thorium formula called key_clean, would look like:

```
base:
  '*':
    - key_clean
```

Take note that the target in a Thorium top.sls is not used; it only exists to follow the same convention as other top.sls files. Leave this set to '*' in your own Thorium top.sls.

12.3.2 Thorium Formula Files

Thorium SLS files are processed by the same state compiler that processes Salt state files. This means that features like requisites, templates, and so on are available.

Let's take a look at an example, and then discuss each component of it. This formula uses Thorium to detect when a minion has disappeared and then deletes the key from the master when the minion has been gone for 60 seconds:

```
statreg:
  status.reg

keydel:
  key.timeout:
    - delete: 60
    - require:
      - status: statreg
```

There are two stanzas in this formula, whose IDs are `statreg` and `keydel`. The first stanza, `statreg`, tells Thorium to keep track of minion status beacons in its *register*. We'll talk more about the register in a moment.

The second stanza, `keydel`, is the one that does the real work. It uses the `key` module to apply an expiration (using the `timeout` function) to a minion. Because `delete` is set to `60`, this is a 60 second expiration. If a minion does not check in at least once every 60 seconds, its key will be deleted from the master. This particular function also allows you to use `reject` instead of `delete`, allowing for a minion to be rejected instead of deleted if it does not check in within the specified time period.

There is also a `require` requisite in this stanza. It states that the `key.timeout` function will not be called unless the `status.reg` function in the `statreg` codeblock has been successfully called first.

12.3.3 Thorium Links to Beacons

The above example was added in the 2016.11.0 release of Salt and makes use of the `status` beacon also added in the 2016.11.0 release. For the above Thorium state to function properly you will also need to enable the `status` beacon in the `minion` configuration file:

```
beacons:
  status:
    - interval: 10
```

This will cause the minion to use the status beacon to check in with the master every 10 seconds.

12.4 The Thorium Register

In order to keep track of information, Thorium uses an in-memory register (or rather, collection of registers) on the master. These registers are only populated when told to by a formula, and they normally will be erased when the master is restarted. It is possible to persist the registers to disk, but we'll get to that in a moment.

The example above uses `status.reg` to populate a register for you, which is automatically used by the `key.timeout` function. However, you can set your own register values as well, using the `reg` module.

Because Thorium watches the event bus, the `reg` module is designed to look for user-specified tags, and then extract data from the payload of events that match those tags. For instance, the following stanza will look for an event with a tag of `my/custom/event`:

```
foo:
  reg.list:
    - add: bar
    - match: my/custom/event
```

When such an event is found, the data found in the payload dictionary key of `bar` will be stored in a register called `foo`. This register will store that data in a `list`. You may also use `reg.set` to add data to a `set()` instead.

If you would like to see a copy of the register as it is stored in memory, you can use the `file.save` function:

```
myreg:
  file.save
```

In this case, each time the register is updated, a copy will be saved in JSON format at `/var/cache/salt/master/thorium/saves/myreg`. If you would like to see when particular events are added to a list-type register, you may add a `stamp` option to `reg.list` (but not `reg.set`). With the above two stanzas put together, this would look like:

```
foo:
  reg.list:
    - add: bar
    - match: my/custom/event
    - stamp: True

myreg:
  file.save
```

If you would like to only keep a certain number of the most recent register entries, you may also add a `prune` option to `reg.list` (but not `reg.set`):

```
foo:
  reg.list:
    - add: bar
    - match: my/custom/event
    - stamp: True
    - prune: 50
```

This example will only keep the 50 most recent entries in the `foo` register.

12.4.1 Using Register Data

Putting data in a register is useless if you don't do anything with it. The `check` module is designed to examine register data and determine whether it matches the given parameters. For instance, the `check.contains` function will return `True` if the given `value` is contained in the specified register:

```
foo:
  reg.list:
    - add: bar
    - match: my/custom/event
    - stamp: True
    - prune: 50
  check.contains:
    - value: somedata
```

Used with a `require` requisite, we can call one of the wrapper modules and perform an operation. For example:

```
shell_test:
  local.cmd:
    - tgt: dufresne
    - func: cmd.run
    - arg:
      - echo 'thorium success' > /tmp/thorium.txt
    - require:
      - check: foo
```

This stanza will only run if the `check.contains` function under the `foo` ID returns true (meaning the match was found).

There are a number of other functions in the `check` module which use different means of comparing values:

- `gt`: Check whether the register entry is greater than the given value
- `gte`: Check whether the register entry is greater than or equal to the given value
- `lt`: Check whether the register entry is less than the given value

- lte: Check whether the register entry is less than or equal to the given value
- eq: Check whether the register entry is equal to the given value
- ne: Check whether the register entry is not equal to the given value

There is also a function called check.event which does not examine the register. Instead, it looks directly at an event as it is coming in on the event bus, and returns True if that event's tag matches. For example:

```
salt/foo/*/bar:
  check.event

run_remote_ex:
  local.cmd:
    - tgt: '*'
    - func: test.ping
    - require:
      - check: salt/foo/*/bar
```

This formula will look for an event whose tag is salt/foo/<anything>/bar and if it comes in, issue a test.ping to all minions.

12.4.2 Register Persistence

It is possible to persist the register data to disk when a master is stopped gracefully, and reload it from disk when the master starts up again. This functionality is provided by the returner subsystem, and is enabled whenever any returner containing a load_reg and a save_reg function is used.

CHAPTER 13

Salt Cloud

13.1 Configuration

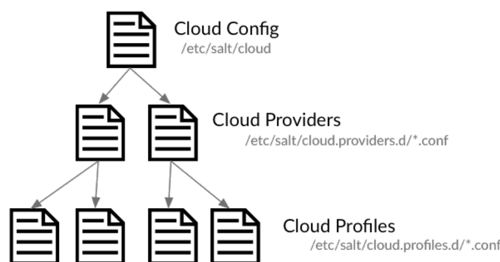
Salt Cloud provides a powerful interface to interact with cloud hosts. This interface is tightly integrated with Salt, and new virtual machines are automatically connected to your Salt master after creation.

Since Salt Cloud is designed to be an automated system, most configuration is done using the following YAML configuration files:

- `/etc/salt/cloud`: The main configuration file, contains global settings that apply to all cloud hosts. See *Salt Cloud Configuration*.
- `/etc/salt/cloud.providers.d/*.conf`: Contains settings that configure a specific cloud host, such as credentials, region settings, and so on. Since configuration varies significantly between each cloud host, a separate file should be created for each cloud host. In Salt Cloud, a provider is synonymous with a cloud host (Amazon EC2, Google Compute Engine, Rackspace, and so on). See *Provider Specifics*.
- `/etc/salt/cloud.profiles.d/*.conf`: Contains settings that define a specific VM type. A profile defines the systems specs and image, and any other settings that are specific to this VM type. Each specific VM type is called a profile, and multiple profiles can be defined in a profile file. Each profile references a parent provider that defines the cloud host in which the VM is created (the provider settings are in the provider configuration explained above). Based on your needs, you might define different profiles for web servers, database servers, and so on. See *VM Profiles*.

13.2 Configuration Inheritance

Configuration settings are inherited in order from the cloud config => providers => profile.



For example, if you wanted to use the same image for all virtual machines for a specific provider, the image name could be placed in the provider file. This value is inherited by all profiles that use that provider, but is overridden if a image name is defined in the profile.

Most configuration settings can be defined in any file, the main difference being how that setting is inherited.

13.3 QuickStart

The *Salt Cloud Quickstart* walks you through defining a provider, a VM profile, and shows you how to create virtual machines using Salt Cloud.

Note that if you installed Salt via Salt Bootstrap, it may not have automatically installed salt-cloud for you. Use your distribution's package manager to install the `salt-cloud` package from the same repo that you used to install Salt. These repos will automatically be setup by Salt Bootstrap.

Alternatively, the `-L` option can be passed to the Salt Bootstrap script when installing Salt. The `-L` option will install `salt-cloud` and the required `libcloud` package.

13.4 Using Salt Cloud

13.4.1 salt-cloud

Provision virtual machines in the cloud with Salt

Synopsis

```
salt-cloud -m /etc/salt/cloud.map

salt-cloud -m /etc/salt/cloud.map NAME

salt-cloud -m /etc/salt/cloud.map NAME1 NAME2

salt-cloud -p PROFILE NAME

salt-cloud -p PROFILE NAME1 NAME2 NAME3 NAME4 NAME5 NAME6
```

Description

Salt Cloud is the system used to provision virtual machines on various public clouds via a cleanly controlled profile and mapping system.

Options

--version
Print the version of Salt 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`.

Execution Options

-L LOCATION, **--location**=LOCATION
Specify which region to connect to.

-a ACTION, **--action**=ACTION
Perform an action that may be specific to this cloud provider. This argument requires one or more instance names to be specified.

-f <FUNC-NAME> <PROVIDER>, **--function**=<FUNC-NAME> <PROVIDER>
Perform an function that may be specific to this cloud provider, that does not apply to an instance. This argument requires a provider to be specified (i.e.: nova).

-p PROFILE, **--profile**=PROFILE
Select a single profile to build the named cloud VMs from. The profile must be defined in the specified profiles file.

-m MAP, **--map**=MAP
Specify a map file to use. If used without any other options, this option will ensure that all of the mapped VMs are created. If the named VM already exists then it will be skipped.

-H, **--hard**
When specifying a map file, the default behavior is to ensure that all of the VMs specified in the map file are created. If the --hard option is set, then any VMs that exist on configured cloud providers that are not specified in the map file will be destroyed. Be advised that this can be a destructive operation and should be used with care.

-d, **--destroy**
Pass in the name(s) of VMs to destroy, salt-cloud will search the configured cloud providers for the specified names and destroy the VMs. Be advised that this is a destructive operation and should be used with care. Can be used in conjunction with the -m option to specify a map of VMs to be deleted.

-P, **--parallel**
Normally when building many cloud VMs they are executed serially. The -P option will run each cloud vm build in a separate process allowing for large groups of VMs to be build at once.

Be advised that some cloud provider's systems don't seem to be well suited for this influx of vm creation. When creating large groups of VMs watch the cloud provider carefully.

-u, **--update-bootstrap**
Update salt-bootstrap to the latest stable bootstrap release.

-y, **--assume-yes**
Default yes in answer to all confirmation questions.

-k, **--keep-tmp**
Do not remove files from /tmp/ after deploy.sh finishes.

--show-deploy-args
Include the options used to deploy the minion in the data returned.

--script-args=SCRIPT_ARGS
Script arguments to be fed to the bootstrap script when deploying the VM.

Query Options

-Q, **--query**
Execute a query and return some information about the nodes running on configured cloud providers

-F, --full-query
Execute a query and print out all available information about all cloud VMs. Can be used in conjunction with -m to display only information about the specified map.

-S, --select-query
Execute a query and print out selected information about all cloud VMs. Can be used in conjunction with -m to display only information about the specified map.

--list-providers
Display a list of configured providers.

--list-profiles
New in version 2014.7.0.

Display a list of configured profiles. Pass in a cloud provider to view the provider's associated profiles, such as `digitalocean`, or pass in `all` to list all the configured profiles.

Cloud Providers Listings

--list-locations=LIST_LOCATIONS
Display a list of locations available in configured cloud providers. Pass the cloud provider that available locations are desired on, aka ``linode'', or pass ``all'' to list locations for all configured cloud providers

--list-images=LIST_IMAGES
Display a list of images available in configured cloud providers. Pass the cloud provider that available images are desired on, aka ``linode'', or pass ``all'' to list images for all configured cloud providers

--list-sizes=LIST_SIZES
Display a list of sizes available in configured cloud providers. Pass the cloud provider that available sizes are desired on, aka ``AWS'', or pass ``all'' to list sizes for all configured cloud providers

Cloud Credentials

--set-password=<USERNAME> <PROVIDER>
Configure password for a cloud provider and save it to the keyring. PROVIDER can be specified with or without a driver, for example: ``--set-password bob rackspace'' or more specific ``--set-password bob rackspace:openstack'' DEPRECATED!

Output Options

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

`grains`, `highstate`, `json`, `key`, `overstatestage`, `pprint`, `raw`, `txt`, `yaml`

Some outputters are formatted only for data returned from specific functions; for instance, the `grains` outputter will not work for non-grains data.

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 static 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.

Examples

To create 4 VMs named web1, web2, db1, and db2 from specified profiles:

```
salt-cloud -p fedora_rackspace web1 web2 db1 db2
```

To read in a map file and create all VMs specified therein:

```
salt-cloud -m /path/to/cloud.map
```

To read in a map file and create all VMs specified therein in parallel:

```
salt-cloud -m /path/to/cloud.map -P
```

To delete any VMs specified in the map file:

```
salt-cloud -m /path/to/cloud.map -d
```

To delete any VMs NOT specified in the map file:

```
salt-cloud -m /path/to/cloud.map -H
```

To display the status of all VMs specified in the map file:

```
salt-cloud -m /path/to/cloud.map -Q
```

See also

salt-cloud(7) salt(7) salt-master(1) salt-minion(1)

13.4.2 Salt Cloud basic usage

Salt Cloud needs, at least, one configured *Provider* and *Profile* to be functional.

Creating a VM

To create a VM with salt cloud, use command:

```
salt-cloud -p <profile> name_of_vm
```

Assuming there is a profile configured as following:

```
fedora_rackspace:
    provider: my-rackspace-config
    image: Fedora 17
    size: 256 server
    script: bootstrap-salt
```

Then, the command to create new VM named `fedora_http_01` is:

```
salt-cloud -p fedora_rackspace fedora_http_01
```

Destroying a VM

To destroy a created-by-salt-cloud VM, use command:

```
salt-cloud -d name_of_vm
```

For example, to delete the VM created on above example, use:

```
salt-cloud -d fedora_http_01
```

13.4.3 VM Profiles

Salt cloud designates virtual machines inside the profile configuration file. The profile configuration file defaults to `/etc/salt/cloud.profiles` and is a yaml configuration. The syntax for declaring profiles is simple:

```
fedora_rackspace:
    provider: my-rackspace-config
    image: Fedora 17
    size: 256 server
    script: bootstrap-salt
```

It should be noted that the `script` option defaults to `bootstrap-salt`, and does not normally need to be specified. Further examples in this document will not show the `script` option.

A few key pieces of information need to be declared and can change based on the cloud provider. A number of additional parameters can also be inserted:

```
centos_rackspace:
  provider: my-rackspace-config
  image: CentOS 6.2
  size: 1024 server
  minion:
    master: salt.example.com
    append_domain: webs.example.com
    grains:
      role: webserver
```

The image must be selected from available images. Similarly, sizes must be selected from the list of sizes. To get a list of available images and sizes use the following command:

```
salt-cloud --list-images openstack
salt-cloud --list-sizes openstack
```

Some parameters can be specified in the main Salt cloud configuration file and then are applied to all cloud profiles. For instance if only a single cloud provider is being used then the provider option can be declared in the Salt cloud configuration file.

Multiple Configuration Files

In addition to `/etc/salt/cloud.profiles`, profiles can also be specified in any file matching `cloud.profiles.d/*conf` which is a sub-directory relative to the profiles configuration file(with the above configuration file as an example, `/etc/salt/cloud.profiles.d/*.conf`). This allows for more extensible configuration, and plays nicely with various configuration management tools as well as version control systems.

Larger Example

```
rhel_ec2:
  provider: my-ec2-config
  image: ami-e565ba8c
  size: t1.micro
  minion:
    cheese: edam

ubuntu_ec2:
  provider: my-ec2-config
  image: ami-7e2da54e
  size: t1.micro
  minion:
    cheese: edam

ubuntu_rackspace:
  provider: my-rackspace-config
  image: Ubuntu 12.04 LTS
  size: 256 server
  minion:
    cheese: edam

fedora_rackspace:
  provider: my-rackspace-config
  image: Fedora 17
  size: 256 server
  minion:
```

```
    cheese: edam

cent_linode:
  provider: my-linode-config
  image: CentOS 6.2 64bit
  size: Linode 512

cent_gogrid:
  provider: my-gogrid-config
  image: 12834
  size: 512MB

cent_joyent:
  provider: my-joyent-config
  image: centos-7
  size: g4-highram-16G
```

13.4.4 Cloud Map File

A number of options exist when creating virtual machines. They can be managed directly from profiles and the command line execution, or a more complex map file can be created. The map file allows for a number of virtual machines to be created and associated with specific profiles. The map file is designed to be run once to create these more complex scenarios using salt-cloud.

Map files have a simple format, specify a profile and then a list of virtual machines to make from said profile:

```
fedora_small:
  - web1
  - web2
  - web3
fedora_high:
  - redis1
  - redis2
  - redis3
cent_high:
  - riak1
  - riak2
  - riak3
```

This map file can then be called to roll out all of these virtual machines. Map files are called from the salt-cloud command with the -m option:

```
$ salt-cloud -m /path/to/mapfile
```

Remember, that as with direct profile provisioning the -P option can be passed to create the virtual machines in parallel:

```
$ salt-cloud -m /path/to/mapfile -P
```

Note: Due to limitations in the GoGrid API, instances cannot be provisioned in parallel with the GoGrid driver. Map files will work with GoGrid, but the `-P` argument should not be used on maps referencing GoGrid instances.

A map file can also be enforced to represent the total state of a cloud deployment by using the `--hard` option. When using the hard option any vms that exist but are not specified in the map file will be destroyed:

```
$ salt-cloud -m /path/to/mapfile -P -H
```

Be careful with this argument, it is very dangerous! In fact, it is so dangerous that in order to use it, you must explicitly enable it in the main configuration file.

```
enable_hard_maps: True
```

A map file can include grains and minion configuration options:

```
fedora_small:
  - web1:
      minion:
        log_level: debug
      grains:
        cheese: tasty
        omelet: du fromage
  - web2:
      minion:
        log_level: warn
      grains:
        cheese: more tasty
        omelet: with peppers
```

Any top level data element from your profile may be overridden in the map file:

```
fedora_small:
  - web1:
    size: t2.micro
  - web2:
    size: t2.nano
```

As of Salt 2017.7.0, nested elements are merged, and can can be specified individually without having to repeat the complete definition for each top level data element. In this example a separate MAC is assigned to each VMware instance while inheriting device parameters for for disk and network configuration:

```
nyc-vm:
  - db1:
      devices:
        network:
          Network Adapter 1:
            mac: '44:44:44:44:44:41'
  - db2:
      devices:
        network:
          Network Adapter 1:
            mac: '44:44:44:44:44:42'
```

A map file may also be used with the various query options:

```
$ salt-cloud -m /path/to/mapfile -Q
{'ec2': {'web1': {'id': 'i-e6aqfegb',
                     'image': None,
                     'private_ips': [],
                     'public_ips': [],
                     'size': None,
                     'state': 0}},
         'web2': {'Absent'}}
```

...or with the delete option:

```
$ salt-cloud -m /path/to/mapfile -d
The following virtual machines are set to be destroyed:
  web1
  web2

Proceed? [N/y]
```

> **Warning:** Specifying Nodes with Maps on the Command Line Specifying the name of a node or nodes with the maps options on the command line is *not* supported. This is especially important to remember when using `--destroy` with maps; `salt-cloud` will ignore any arguments passed in which are not directly relevant to the map file. *When using ``--destroy`` with a map, every node in the map file will be deleted!* Maps don't provide any useful information for destroying individual nodes, and should not be used to destroy a subset of a map.

Setting up New Salt Masters

Bootstrapping a new master in the map is as simple as:

```
fedora_small:
  - web1:
      make_master: True
  - web2
  - web3
```

Notice that **ALL** bootstrapped minions from the map will answer to the newly created salt-master.

To make any of the bootstrapped minions answer to the bootstrapping salt-master as opposed to the newly created salt-master, as an example:

```
fedora_small:
  - web1:
      make_master: True
      minion:
        master: <the local master ip address>
        local_master: True
  - web2
  - web3
```

The above says the minion running on the newly created salt-master responds to the local master, ie, the master used to bootstrap these VMs.

Another example:

```
fedora_small:
  - web1:
      make_master: True
  - web2
  - web3:
      minion:
        master: <the local master ip address>
        local_master: True
```

The above example makes the `web3` minion answer to the local master, not the newly created master.

Using Direct Map Data

When using modules that access the CloudClient directly (notably, the cloud execution and runner modules), it is possible to pass in the contents of a map file, rather than a path to the location of the map file.

Normally when using these modules, the path to the map file is passed in using:

```
salt-run cloud.map_run /path/to/cloud.map
```

To pass in the actual map data, use the map_data argument:

```
salt-run cloud.map_run map_data='{"centos7": [{"saltmaster": {"minion": \
    {"transport": "tcp"}, "make_master": true, "master": {"transport": \
    "tcp"}}}, {"minion001": {"minion": {"transport": "tcp"}}}]}'
```

13.4.5 Cloud Actions

Once a VM has been created, there are a number of actions that can be performed on it. The ``reboot'' action can be used across all providers, but all other actions are specific to the cloud provider. In order to perform an action, you may specify it from the command line, including the name(s) of the VM to perform the action on:

```
$ salt-cloud -a reboot vm_name
$ salt-cloud -a reboot vm1 vm2 vm2
```

Or you may specify a map which includes all VMs to perform the action on:

```
$ salt-cloud -a reboot -m /path/to/mapfile
```

The following is an example list of actions currently supported by salt-cloud:

```
all providers:
    - reboot
ec2:
    - start
    - stop
joyent:
    - stop
linode:
    - start
    - stop
```

Another useful reference for viewing more salt-cloud actions is the *Salt Cloud Feature Matrix*.

13.4.6 Cloud Functions

Cloud functions work much the same way as cloud actions, except that they don't perform an operation on a specific instance, and so do not need a machine name to be specified. However, since they perform an operation on a specific cloud provider, that provider must be specified.

```
$ salt-cloud -f show_image ec2 image=ami-fd20ad94
```

There are three universal salt-cloud functions that are extremely useful for gathering information about instances on a provider basis:

- list_nodes: Returns some general information about the instances for the given provider.

- `list_nodes_full`: Returns all information about the instances for the given provider.
- `list_nodes_select`: Returns select information about the instances for the given provider.

```
$ salt-cloud -f list_nodes linode
$ salt-cloud -f list_nodes_full linode
$ salt-cloud -f list_nodes_select linode
```

Another useful reference for viewing `salt-cloud` functions is the *Salt Cloud Feature Matrix*.

13.5 Core Configuration

13.5.1 Install Salt Cloud

Salt Cloud is now part of Salt proper. It was merged in as of *Salt version 2014.1.0*.

On Ubuntu, install Salt Cloud by using following command:

```
sudo add-apt-repository ppa:saltstack/salt
sudo apt-get update
sudo apt-get install salt-cloud
```

If using Salt Cloud on macOS, `curl-ca-bundle` must be installed. Presently, this package is not available via `brew`, but it is available using MacPorts:

```
sudo port install curl-ca-bundle
```

Salt Cloud depends on `apache-libcloud`. Libcloud can be installed via pip with `pip install apache-libcloud`.

Installing Salt Cloud for development

Installing Salt for development enables Salt Cloud development as well, just make sure `apache-libcloud` is installed as per above paragraph.

See these instructions: *Installing Salt for development*.

13.5.2 Core Configuration

A number of core configuration options and some options that are global to the VM profiles can be set in the cloud configuration file. By default this file is located at `/etc/salt/cloud`.

Thread Pool Size

When salt cloud is operating in parallel mode via the `-P` argument, you can control the thread pool size by specifying the `pool_size` parameter with a positive integer value.

By default, the thread pool size will be set to the number of VMs that salt cloud is operating on.

```
pool_size: 10
```

Minion Configuration

The default minion configuration is set up in this file. Minions created by salt-cloud derive their configuration from this file. Almost all parameters found in *Configuring the Salt Minion* can be used here.

```
minion:
  master: saltmaster.example.com
```

In particular, this is the location to specify the location of the salt master and its listening port, if the port is not set to the default.

Similar to most other settings, Minion configuration settings are inherited across configuration files. For example, the master setting might be contained in the main `cloud` configuration file as demonstrated above, but additional settings can be placed in the provider or profile:

```
ec2-web:
  size: t1.micro
  minion:
    environment: test
    startup_states: sls
    sls_list:
      - web
```

When salt cloud creates a new minon, it can automatically add grain information to the minion configuration file identifying the sources originally used to define it.

The generated grain information will appear similar to:

```
grains:
  salt-cloud:
    driver: ec2
    provider: my_ec2:ec2
    profile: ec2-web
```

The generation of the salt-cloud grain can be suppressed by the option `enable_cloud_grains: 'False'` in the cloud configuration file.

Cloud Configuration Syntax

The data specific to interacting with public clouds is set up *here*.

Cloud provider configuration settings can live in several places. The first is in `/etc/salt/cloud`:

```
# /etc/salt/cloud
providers:
  my-aws-migrated-config:
    id: HJGRYCILJLKJYG
    key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
    keyname: test
    securitygroup: quick-start
    private_key: /root/test.pem
    driver: ec2
```

Cloud provider configuration data can also be housed in `/etc/salt/cloud.providers` or any file matching `/etc/salt/cloud.providers.d/*.conf`. All files in any of these locations will be parsed for cloud provider data.

Using the example configuration above:

```
# /etc/salt/cloud.providers
# or could be /etc/salt/cloud.providers.d/*.conf
my-aws-config:
  id: HJGRYCILJLKJYG
  key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
  keyname: test
  securitygroup: quick-start
  private_key: /root/test.pem
  driver: ec2
```

Note: Salt Cloud provider configurations within `/etc/cloud.provider.d/` should not specify the `providers` starting key.

It is also possible to have multiple cloud configuration blocks within the same alias block. For example:

```
production-config:
  - id: HJGRYCILJLKJYG
    key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
    keyname: test
    securitygroup: quick-start
    private_key: /root/test.pem
    driver: ec2

  - user: example_user
    apikey: 123984bjjas87034
    driver: rackspace
```

However, using this configuration method requires a change with profile configuration blocks. The provider alias needs to have the provider key value appended as in the following example:

```
rhel_aws_dev:
  provider: production-config:ec2
  image: ami-e565ba8c
  size: t1.micro

rhel_aws_prod:
  provider: production-config:ec2
  image: ami-e565ba8c
  size: High-CPU Extra Large Instance

database_prod:
  provider: production-config:rackspace
  image: Ubuntu 12.04 LTS
  size: 256 server
```

Notice that because of the multiple entries, one has to be explicit about the provider alias and name, from the above example, `production-config: ec2`.

This data interactions with the `salt-cloud` binary regarding its `--list-location`, `--list-images`, and `--list-sizes` which needs a cloud provider as an argument. The argument used should be the configured cloud provider alias. If the provider alias has multiple entries, `<provider-alias>: <provider-name>` should be used.

To allow for a more extensible configuration, `--providers-config`, which defaults to `/etc/salt/cloud.providers`, was added to the cli parser. It allows for the providers' configuration to be added on a per-file basis.

Pillar Configuration

It is possible to configure cloud providers using pillars. This is only used when inside the cloud module. You can setup a variable called `cloud` that contains your profile and provider to pass that information to the cloud servers instead of having to copy the full configuration to every minion. In your pillar file, you would use something like this:

```
cloud:
  ssh_key_name: saltstack
  ssh_key_file: /root/.ssh/id_rsa
  update_cachedir: True
  diff_cache_events: True

  providers:
    my-openstack:
      driver: openstack
      region_name: ORD
      cloud: mycloud

  profiles:
    ubuntu-openstack:
      provider: my-openstack
      size: ds512M
      image: CentOS 7
      script_args: git develop
```

Cloud Configurations

Scaleway

To use Salt Cloud with Scaleway, you need to get an `access key` and an `API token`. `API tokens` are unique identifiers associated with your Scaleway account. To retrieve your `access key` and `API token`, log-in to the Scaleway control panel, open the pull-down menu on your account name and click on ``My Credentials'' link.

If you do not have `API token` you can create one by clicking the ``Create New Token'' button on the right corner.

```
my-scaleway-config:
  access_key: 15cf404d-4560-41b1-9a0c-21c3d5c4ff1f
  token: a7347ec8-5de1-4024-a5e3-24b77d1ba91d
  driver: scaleway
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-scaleway-config`.

Rackspace

Rackspace cloud requires two configuration options; a `user` and an `apikey`:

```
my-rackspace-config:
  user: example_user
  apikey: 123984bjjas87034
  driver: rackspace
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-rackspace-config`.

Amazon AWS

A number of configuration options are required for Amazon AWS including `id`, `key`, `keyname`, `securitygroup`, and `private_key`:

```
my-aws-quick-start:
  id: HJGRYCILJLKJYG
  key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
  keyname: test
  securitygroup: quick-start
  private_key: /root/test.pem
  driver: ec2

my-aws-default:
  id: HJGRYCILJLKJYG
  key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
  keyname: test
  securitygroup: default
  private_key: /root/test.pem
  driver: ec2
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be either `provider: my-aws-quick-start` or `provider: my-aws-default`.

Linode

Linode requires a single API key, but the default root password also needs to be set:

```
my-linode-config:
  apikey: asldkgfakl;sdfjsjaslfjaklsdjf;askldjfaaklsjdfhasldsadfghdkf
  password: F00barbaz
  ssh_pubkey: ssh-ed25519
↪AAAAC3NzaC1lZDI1NTE5AAAAIKHEOLLbeXgaqRQT9NBAopVz366SdYc0KKX33vAnq+2R user@host
  ssh_key_file: ~/.ssh/id_ed25519
  driver: linode
```

The password needs to be 8 characters and contain lowercase, uppercase, and numbers.

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-linode-config`

Joyent Cloud

The Joyent cloud requires three configuration parameters: The username and password that are used to log into the Joyent system, as well as the location of the private SSH key associated with the Joyent account. The SSH key

is needed to send the provisioning commands up to the freshly created virtual machine.

```
my-joyent-config:
  user: fred
  password: saltybacon
  private_key: /root/joyent.pem
  driver: joyent
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-joyent-config`

GoGrid

To use Salt Cloud with GoGrid, log into the GoGrid web interface and create an API key. Do this by clicking on ``My Account'' and then going to the API Keys tab.

The `apikey` and the `sharedsecret` configuration parameters need to be set in the configuration file to enable interfacing with GoGrid:

```
my-gogrid-config:
  apikey: asdff7896asdh789
  sharedsecret: saltybacon
  driver: gogrid
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-gogrid-config`.

OpenStack

Using Salt for OpenStack uses the *shade <https://docs.openstack.org/shade/latest/>* driver managed by the openstack-infra team.

This driver can be configured using the `/etc/openstack/clouds.yml` file with *os-client-config <https://docs.openstack.org/os-client-config/latest/>*

```
myopenstack:
  driver: openstack
  region_name: RegionOne
  cloud: mycloud
```

Or by just configuring the same auth block directly in the cloud provider config.

```
myopenstack:
  driver: openstack
  region_name: RegionOne
  auth:
    username: 'demo'
    password: secret
    project_name: 'demo'
    auth_url: 'http://openstack/identity'
```

Both of these methods support using the *vendor <https://docs.openstack.org/os-client-config/latest/user/vendor-support.html>* options.

For more information, look at *Openstack Cloud Driver Docs*

DigitalOcean

Using Salt for DigitalOcean requires a `client_key` and an `api_key`. These can be found in the DigitalOcean web interface, in the ``My Settings'' section, under the API Access tab.

```
my-digitalocean-config:
  driver: digitalocean
  personal_access_token: xxx
  location: New York 1
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-digital-ocean-config`.

Parallels

Using Salt with Parallels requires a `user`, `password` and URL. These can be obtained from your cloud provider.

```
my-parallels-config:
  user: myuser
  password: xyzzy
  url: https://api.cloud.xmission.com:4465/paci/v1.0/
  driver: parallels
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-parallels-config`.

Proxmox

Using Salt with Proxmox requires a `user`, `password`, and URL. These can be obtained from your cloud host. Both PAM and PVE users can be used.

```
my-proxmox-config:
  driver: proxmox
  user: saltcloud@pve
  password: xyzzy
  url: your.proxmox.host
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: my-proxmox-config`.

LXC

The lxc driver uses saltify to install salt and attach the lxc container as a new lxc minion. As soon as we can, we manage baremetal operation over SSH. You can also destroy those containers via this driver.

```
devhost10-lxc:
  target: devhost10
  driver: lxc
```

And in the map file:

```
devhost10-lxc:
  provider: devhost10-lxc
  from_container: ubuntu
  backing: lvm
  sudo: True
  size: 3g
  ip: 10.0.3.9
  minion:
    master: 10.5.0.1
    master_port: 4506
  lxc_conf:
    - lxc.utsname: superlxc
```

Note: In the cloud profile that uses this provider configuration, the syntax for the `provider` required field would be `provider: devhost10-lxc`.

Saltify

The Saltify driver is a new, experimental driver designed to install Salt on a remote machine, virtual or bare metal, using SSH. This driver is useful for provisioning machines which are already installed, but not Salted. For more information about using this driver and for configuration examples, please see the *Gettting Started with Saltify* documentation.

Vagrant

The Vagrant driver is a new, experimental driver for controlling a VagrantBox virtual machine, and installing Salt on it. The target host machine must be a working salt minion, which is controlled via the salt master using salt-api. For more information, see *Getting Started With Vagrant*.

Extending Profiles and Cloud Providers Configuration

As of 0.8.7, the option to extend both the profiles and cloud providers configuration and avoid duplication was added. The extends feature works on the current profiles configuration, but, regarding the cloud providers configuration, **only** works in the new syntax and respective configuration files, i.e. `/etc/salt/salt/cloud.providers` or `/etc/salt/cloud.providers.d/*.conf`.

Note: Extending cloud profiles and providers is not recursive. For example, a profile that is extended by a second profile is possible, but the second profile cannot be extended by a third profile.

Also, if a profile (or provider) is extending another profile and each contains a list of values, the lists from the extending profile will override the list from the original profile. The lists are not merged together.

Extending Profiles

Some example usage on how to use `extends` with profiles. Consider `/etc/salt/salt/cloud.profiles` containing:

```
development-instances:
  provider: my-ec2-config
  size: t1.micro
  ssh_username: ec2_user
  securitygroup:
    - default
  deploy: False

Amazon-Linux-AMI-2012.09-64bit:
  image: ami-54cf5c3d
  extends: development-instances

Fedora-17:
  image: ami-08d97e61
  extends: development-instances

CentOS-5:
  provider: my-aws-config
  image: ami-09b61d60
  extends: development-instances
```

The above configuration, once parsed would generate the following profiles data:

```
[{'deploy': False,
  'image': 'ami-08d97e61',
  'profile': 'Fedora-17',
  'provider': 'my-ec2-config',
  'securitygroup': ['default'],
  'size': 't1.micro',
  'ssh_username': 'ec2_user'},
 {'deploy': False,
  'image': 'ami-09b61d60',
  'profile': 'CentOS-5',
  'provider': 'my-aws-config',
  'securitygroup': ['default'],
  'size': 't1.micro',
  'ssh_username': 'ec2_user'},
 {'deploy': False,
  'image': 'ami-54cf5c3d',
  'profile': 'Amazon-Linux-AMI-2012.09-64bit',
  'provider': 'my-ec2-config',
  'securitygroup': ['default'],
  'size': 't1.micro',
  'ssh_username': 'ec2_user'},
 {'deploy': False,
  'profile': 'development-instances',
  'provider': 'my-ec2-config',
  'securitygroup': ['default'],
```

```
  'size': 't1.micro',
  'ssh_username': 'ec2_user'}]
```

Pretty cool right?

Extending Providers

Some example usage on how to use `extends` within the cloud providers configuration. Consider `/etc/salt/salt/cloud.providers` containing:

```
my-develop-envs:
  - id: HJGRYCILJLKJYG
    key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
    keyname: test
    securitygroup: quick-start
    private_key: /root/test.pem
    location: ap-southeast-1
    availability_zone: ap-southeast-1b
    driver: ec2

  - user: myuser@mycorp.com
    password: mypass
    ssh_key_name: mykey
    ssh_key_file: '/etc/salt/ibm/mykey.pem'
    location: Raleigh
    driver: ibmsce


my-productions-envs:
  - extends: my-develop-envs:ibmsce
    user: my-production-user@mycorp.com
    location: us-east-1
    availability_zone: us-east-1
```

The above configuration, once parsed would generate the following providers data:

```
'providers': {
    'my-develop-envs': [
        {'availability_zone': 'ap-southeast-1b',
         'id': 'HJGRYCILJLKJYG',
         'key': 'kdjgfsgm;woormgl/aserigjksjdhasdfgn',
         'keyname': 'test',
         'location': 'ap-southeast-1',
         'private_key': '/root/test.pem',
         'driver': 'aws',
         'securitygroup': 'quick-start'
        },
        {'location': 'Raleigh',
         'password': 'mypass',
         'driver': 'ibmsce',
         'ssh_key_file': '/etc/salt/ibm/mykey.pem',
         'ssh_key_name': 'mykey',
         'user': 'myuser@mycorp.com'
        }
    ],
    'my-productions-envs': [
```

```
        {'availability_zone': 'us-east-1',
         'location': 'us-east-1',
         'password': 'mypass',
         'driver': 'ibmsce',
         'ssh_key_file': '/etc/salt/ibm/mykey.pem',
         'ssh_key_name': 'mykey',
         'user': 'my-production-user@mycorp.com'
        }
    ]
}
```

13.6 Windows Configuration

13.6.1 Spinning up Windows Minions

It is possible to use Salt Cloud to spin up Windows instances, and then install Salt on them. This functionality is available on all cloud providers that are supported by Salt Cloud. However, it may not necessarily be available on all Windows images.

Requirements

Salt Cloud makes use of *impacket* and *winexe* to set up the Windows Salt Minion installer.

impacket is usually available as either the *impacket* or the *python-impacket* package, depending on the distribution. More information on *impacket* can be found at the project home:

- impacket project home

winexe is less commonly available in distribution-specific repositories. However, it is currently being built for various distributions in 3rd party channels:

- RPMs at pbone.net
- openSUSE Build Service

Optionally WinRM can be used instead of *winexe* if the python module *pywinrm* is available and WinRM is supported on the target Windows version. Information on pywinrm can be found at the project home:

- pywinrm project home

Additionally, a copy of the Salt Minion Windows installer must be present on the system on which Salt Cloud is running. This installer may be downloaded from saltstack.com:

- SaltStack Download Area

Self Signed Certificates with WinRM

Salt-Cloud can use versions of `pywinrm<=0.1.1` or `pywinrm>=0.2.1`.

For versions greater than *0.2.1*, `winrm_verify_ssl` needs to be set to *False* if the certificate is self signed and not verifiable.

Firewall Settings

Because Salt Cloud makes use of *smbclient* and *winexe*, port 445 must be open on the target image. This port is not generally open by default on a standard Windows distribution, and care must be taken to use an image in which this port is open, or the Windows firewall is disabled.

If supported by the cloud provider, a PowerShell script may be used to open up this port automatically, using the cloud provider's *userdata*. The following script would open up port 445, and apply the changes:

```
<powershell>
New-NetFirewallRule -Name "SMB445" -DisplayName "SMB445" -Protocol TCP -LocalPort 445
Set-Item (dir wsman:\localhost\Listener\*\Port -Recurse).pspath 445 -Force
Restart-Service winrm
</powershell>
```

For EC2, this script may be saved as a file, and specified in the provider or profile configuration as *userdata_file*. For instance:

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/windows-firewall.ps1
```

Note: From versions 2016.11.0 and 2016.11.3, this file was passed through the master's `renderer` to template it. However, this caused issues with non-YAML data, so templating is no longer performed by default. To template the userdata_file, add a `userdata_template` option to the cloud profile:

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/windows-firewall.ps1
  userdata_template: jinja
```

If no `userdata_template` is set in the cloud profile, then the master configuration will be checked for a `userdata_template` value. If this is not set, then no templating will be performed on the userdata_file.

To disable templating in a cloud profile when a `userdata_template` has been set in the master configuration file, simply set `userdata_template` to `False` in the cloud profile:

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/windows-firewall.ps1
  userdata_template: False
```

If you are using WinRM on EC2 the HTTPS port for the WinRM service must also be enabled in your userdata. By default EC2 Windows images only have insecure HTTP enabled. To enable HTTPS and basic authentication required by pywinrm consider the following userdata example:

```
<powershell>
New-NetFirewallRule -Name "SMB445" -DisplayName "SMB445" -Protocol TCP -LocalPort 445
New-NetFirewallRule -Name "WINRM5986" -DisplayName "WINRM5986" -Protocol TCP -
↪LocalPort 5986

winrm quickconfig -q
winrm set winrm/config/winrs '@{MaxMemoryPerShellMB="300"}'
winrm set winrm/config '@{MaxTimeoutms="1800000"}'
winrm set winrm/config/service/auth '@{Basic="true"}'
```

```
$SourceStoreScope = 'LocalMachine'
$SourceStorename = 'Remote Desktop'

$SourceStore = New-Object -TypeName System.Security.Cryptography.X509Certificates.
↪X509Store -ArgumentList $SourceStorename, $SourceStoreScope
$SourceStore.Open([System.Security.Cryptography.X509Certificates.OpenFlags]::ReadOnly)

$cert = $SourceStore.Certificates | Where-Object -FilterScript {
    $_.subject -like '*'
}

$DestStoreScope = 'LocalMachine'
$DestStoreName = 'My'

$DestStore = New-Object -TypeName System.Security.Cryptography.X509Certificates.
↪X509Store -ArgumentList $DestStoreName, $DestStoreScope
$DestStore.Open([System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite)
$DestStore.Add($cert)

$SourceStore.Close()
$DestStore.Close()

winrm create winrm/config/listener?Address=*+Transport=HTTPS `@`
↪{CertificateThumbprint=`"($cert.Thumbprint)`"`}

Restart-Service winrm
</powershell>
```

No certificate store is available by default on EC2 images and creating one does not seem possible without an MMC (cannot be automated). To use the default EC2 Windows images the above copies the RDP store.

Configuration

Configuration is set as usual, with some extra configuration settings. The location of the Windows installer on the machine that Salt Cloud is running on must be specified. This may be done in any of the regular configuration files (main, providers, profiles, maps). For example:

Setting the installer in `/etc/salt/cloud.providers`:

```
my-softlayer:
  driver: softlayer
  user: MYUSER1138
  apikey: 'e3b68aa711e6deadc62d5b76355674beef7cc3116062ddbacafe5f7e465bfdc9'
  minion:
    master: saltmaster.example.com
  win_installer: /root/Salt-Minion-2014.7.0-AMD64-Setup.exe
  win_username: Administrator
  win_password: letmein
  smb_port: 445
```

The default Windows user is *Administrator*, and the default Windows password is blank.

If WinRM is to be used `use_winrm` needs to be set to *True*. `winrm_port` can be used to specify a custom port (must be HTTPS listener). And `winrm_verify_ssl` can be set to *False* to use a self signed certificate.

Auto-Generated Passwords on EC2

On EC2, when the *win_password* is set to *auto*, Salt Cloud will query EC2 for an auto-generated password. This password is expected to take at least 4 minutes to generate, adding additional time to the deploy process.

When the EC2 API is queried for the auto-generated password, it will be returned in a message encrypted with the specified *keyname*. This requires that the appropriate *private_key* file is also specified. Such a profile configuration might look like:

```
windows-server-2012:
  provider: my-ec2-config
  image: ami-c49c0dac
  size: m1.small
  securitygroup: windows
  keyname: mykey
  private_key: /root/mykey.pem
  userdata_file: /etc/salt/windows-firewall.ps1
  win_installer: /root/Salt-Minion-2014.7.0-AMD64-Setup.exe
  win_username: Administrator
  win_password: auto
```

13.7 Cloud Provider Specifics

13.7.1 Getting Started With Aliyun ECS

The Aliyun ECS (Elastic Computer Service) is one of the most popular public cloud hosts in China. This cloud host can be used to manage aliyun instance using salt-cloud.

http://www.aliyun.com/

Dependencies

This driver requires the Python `requests` library to be installed.

Configuration

Using Salt for Aliyun ECS requires aliyun access key id and key secret. These can be found in the aliyun web interface, in the ``User Center'' section, under ``My Service'' tab.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-aliyun-config:
  # aliyun Access Key ID
  id: wDGEwGregedg3435gDgxd
  # aliyun Access Key Secret
  key: GDd45t43RDBTrkkkg43934t34qT43t4dgegerGEgg
  location: cn-qingdao
  driver: aliyun
```

Note: Changed in version 2015.8.0.

The provider parameter in cloud provider definitions was renamed to driver. This change was made to avoid confusion with the provider parameter that is used in cloud profile definitions. Cloud provider definitions now use driver to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use provider to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at /etc/salt/cloud.profiles or in the /etc/salt/cloud.profiles.d/ directory:

```
aliyun_centos:
    provider: my-aliyun-config
    size: ecs.t1.small
    location: cn-qingdao
    securitygroup: G1989096784427999
    image: centos6u3_64_20G_aliaegis_20130816.vhd
```

Sizes can be obtained using the --list-sizes option for the salt-cloud command:

```
# salt-cloud --list-sizes my-aliyun-config
my-aliyun-config:
    ----------
    aliyun:
        ----------
        ecs.c1.large:
            ----------
            CpuCoreCount:
                8
            InstanceTypeId:
                ecs.c1.large
            MemorySize:
                16.0

...SNIP...
```

Images can be obtained using the --list-images option for the salt-cloud command:

```
# salt-cloud --list-images my-aliyun-config
my-aliyun-config:
    ----------
    aliyun:
        ----------
        centos5u8_64_20G_aliaegis_20131231.vhd:
            ----------
            Architecture:
                x86_64
            Description:

            ImageId:
                centos5u8_64_20G_aliaegis_20131231.vhd
            ImageName:
                CentOS 5.8 64�
            ImageOwnerAlias:
```

```
            system
        ImageVersion:
            1.0
        OSName:
            CentOS  5.8 64⊠
        Platform:
            CENTOS5
        Size:
            20
        Visibility:
            public
...SNIP...
```

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
my-aliyun-config:
    ----------
    aliyun:
        ----------
        cn-beijing:
            ----------
            LocalName:
                ⊠⊠
            RegionId:
                cn-beijing
        cn-hangzhou:
            ----------
            LocalName:
                ⊠⊠
            RegionId:
                cn-hangzhou
        cn-hongkong:
            ----------
            LocalName:
                ⊠⊠
            RegionId:
                cn-hongkong
        cn-qingdao:
            ----------
            LocalName:
                ⊠⊠
            RegionId:
                cn-qingdao
```

Security Group can be obtained using the `-f list_securitygroup` option for the `salt-cloud` command:

```
# salt-cloud --location=cn-qingdao -f list_securitygroup my-aliyun-config
my-aliyun-config:
    ----------
    aliyun:
        ----------
        G1989096784427999:
            ----------
            Description:
                G1989096784427999
            SecurityGroupId:
                G1989096784427999
```

Note: Aliyun ECS REST API documentation is available from Aliyun ECS API.

13.7.2 Getting Started With Azure

New in version 2014.1.0.

Azure is a cloud service by Microsoft providing virtual machines, SQL services, media services, and more. This document describes how to use Salt Cloud to create a virtual machine on Azure, with Salt installed.

More information about Azure is located at http://www.windowsazure.com/.

Dependencies

- Microsoft Azure SDK for Python >= 1.0.2
- The python-requests library, for Python < 2.7.9.
- A Microsoft Azure account
- OpenSSL (to generate the certificates)
- Salt

Note: The Azure driver is currently being updated to work with the new version of the Python Azure SDK, 1.0.0. However until that process is complete, this driver will not work with Azure 1.0.0. Please be sure you're running on a minimum version of 0.10.2 and less than version 1.0.0.

See Issue #27980 for more information.

Configuration

Set up the provider config at `/etc/salt/cloud.providers.d/azure.conf`:

```
# Note: This example is for /etc/salt/cloud.providers.d/azure.conf

my-azure-config:
  driver: azure
  subscription_id: 3287abc8-f98a-c678-3bde-326766fd3617
  certificate_path: /etc/salt/azure.pem

  # Set up the location of the salt master
  #
  minion:
    master: saltmaster.example.com

  # Optional
  management_host: management.core.windows.net
```

The certificate used must be generated by the user. OpenSSL can be used to create the management certificates. Two certificates are needed: a .cer file, which is uploaded to Azure, and a .pem file, which is stored locally.

To create the .pem file, execute the following command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout /etc/salt/azure.pem -out /
↪etc/salt/azure.pem
```

To create the .cer file, execute the following command:

```
openssl x509 -inform pem -in /etc/salt/azure.pem -outform der -out /etc/salt/azure.cer
```

After creating these files, the .cer file will need to be uploaded to Azure via the ``Upload a Management Certificate'' action of the ``Management Certificates'' tab within the ``Settings'' section of the management portal.

Optionally, a `management_host` may be configured, if necessary for the region.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles`:

```
azure-ubuntu:
  provider: my-azure-config
  image: 'b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-12_04_3-LTS-amd64-server-20131003-
↪en-us-30GB'
  size: Small
  location: 'East US'
  ssh_username: azureuser
  ssh_password: verybadpass
  slot: production
  media_link: 'http://portalvhdabcdefghijklmn.blob.core.windows.net/vhds'
  virtual_network_name: azure-virtual-network
  subnet_name: azure-subnet
```

These options are described in more detail below. Once configured, the profile can be realized with a salt command:

```
salt-cloud -p azure-ubuntu newinstance
```

This will create an salt minion instance named `newinstance` in Azure. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
salt newinstance test.ping
```

Profile Options

The following options are currently available for Azure.

provider

The name of the provider as configured in */etc/salt/cloud.providers.d/azure.conf*.

image

The name of the image to use to create a VM. Available images can be viewed using the following command:

```
salt-cloud --list-images my-azure-config
```

size

The name of the size to use to create a VM. Available sizes can be viewed using the following command:

```
salt-cloud --list-sizes my-azure-config
```

location

The name of the location to create a VM in. Available locations can be viewed using the following command:

```
salt-cloud --list-locations my-azure-config
```

affinity_group

The name of the affinity group to create a VM in. Either a `location` or an `affinity_group` may be specified, but not both. See Affinity Groups below.

ssh_username

The user to use to log into the newly-created VM to install Salt.

ssh_password

The password to use to log into the newly-created VM to install Salt.

slot

The environment to which the hosted service is deployed. Valid values are *staging* or *production*. When set to *production*, the resulting URL of the new VM will be *<vm_name>.cloudapp.net*. When set to *staging*, the resulting URL will contain a generated hash instead.

media_link

This is the URL of the container that will store the disk that this VM uses. Currently, this container must already exist. If a VM has previously been created in the associated account, a container should already exist. In the web interface, go into the Storage area and click one of the available storage selections. Click the Containers link, and then copy the URL from the container that will be used. It generally looks like:

```
http://portalvhdabcdefghijklmn.blob.core.windows.net/vhds
```

service_name

The name of the service in which to create the VM. If this is not specified, then a service will be created with the same name as the VM.

virtual_network_name

Optional. The name of the virtual network for the VM to join. If this is not specified, then no virtual network will be joined.

subnet_name

Optional. The name of the subnet in the virtual network for the VM to join. Requires that a `virtual_network_name` is specified.

Show Instance

This action is a thin wrapper around `--full-query`, which displays details on a single instance only. In an environment with several machines, this will save a user from having to sort through all instance data, just to examine a single instance.

```
salt-cloud -a show_instance myinstance
```

Destroying VMs

There are certain options which can be specified in the global cloud configuration file (usually `/etc/salt/cloud`) which affect Salt Cloud's behavior when a VM is destroyed.

cleanup_disks

New in version 2015.8.0.

Default is `False`. When set to `True`, Salt Cloud will wait for the VM to be destroyed, then attempt to destroy the main disk that is associated with the VM.

cleanup_vhds

New in version 2015.8.0.

Default is `False`. Requires `cleanup_disks` to be set to `True`. When also set to `True`, Salt Cloud will ask Azure to delete the VHD associated with the disk that is also destroyed.

cleanup_services

New in version 2015.8.0.

Default is `False`. Requires `cleanup_disks` to be set to `True`. When also set to `True`, Salt Cloud will wait for the disk to be destroyed, then attempt to remove the service that is associated with the VM. Because the disk belongs to the service, the disk must be destroyed before the service can be.

Managing Hosted Services

New in version 2015.8.0.

An account can have one or more hosted services. A hosted service is required in order to create a VM. However, as mentioned above, if a hosted service is not specified when a VM is created, then one will automatically be created with the name of the name. The following functions are also available.

create_service

Create a hosted service. The following options are available.

name

Required. The name of the hosted service to create.

label

Required. A label to apply to the hosted service.

description

Optional. A longer description of the hosted service.

location

Required, if `affinity_group` is not set. The location in which to create the hosted service. Either the `location` or the `affinity_group` must be set, but not both.

affinity_group

Required, if `location` is not set. The affinity group in which to create the hosted service. Either the `location` or the `affinity_group` must be set, but not both.

extended_properties

Optional. Dictionary containing name/value pairs of hosted service properties. You can have a maximum of 50 extended property name/value pairs. The maximum length of the Name element is 64 characters, only alphanumeric characters and underscores are valid in the Name, and the name must start with a letter. The value has a maximum length of 255 characters.

CLI Example

The following example illustrates creating a hosted service.

```
salt-cloud -f create_service my-azure name=my-service label=my-service location='West
↪US'
```

show_service

Return details about a specific hosted service. Can also be called with `get_service`.

```
salt-cloud -f show_storage my-azure name=my-service
```

list_services

List all hosted services associates with the subscription.

```
salt-cloud -f list_services my-azure-config
```

delete_service

Delete a specific hosted service.

```
salt-cloud -f delete_service my-azure name=my-service
```

Managing Storage Accounts

New in version 2015.8.0.

Salt Cloud can manage storage accounts associated with the account. The following functions are available. Deprecated marked as deprecated are marked as such as per the SDK documentation, but are still included for completeness with the SDK.

create_storage

Create a storage account. The following options are supported.

name

Required. The name of the storage account to create.

label

Required. A label to apply to the storage account.

description

Optional. A longer description of the storage account.

location

Required, if `affinity_group` is not set. The location in which to create the storage account. Either the `location` or the `affinity_group` must be set, but not both.

affinity_group

Required, if `location` is not set. The affinity group in which to create the storage account. Either the `location` or the `affinity_group` must be set, but not both.

extended_properties

Optional. Dictionary containing name/value pairs of storage account properties. You can have a maximum of 50 extended property name/value pairs. The maximum length of the Name element is 64 characters, only alphanumeric characters and underscores are valid in the Name, and the name must start with a letter. The value has a maximum length of 255 characters.

geo_replication_enabled

Deprecated. Replaced by the account_type parameter.

account_type

Specifies whether the account supports locally-redundant storage, geo-redundant storage, zone-redundant storage, or read access geo-redundant storage. Possible values are:

- Standard_LRS
- Standard_ZRS
- Standard_GRS
- Standard_RAGRS

CLI Example

The following example illustrates creating a storage account.

```
salt-cloud -f create_storage my-azure name=my-storage label=my-storage location='West
↪US'
```

list_storage

List all storage accounts associates with the subscription.

```
salt-cloud -f list_storage my-azure-config
```

show_storage

Return details about a specific storage account. Can also be called with `get_storage`.

```
salt-cloud -f show_storage my-azure name=my-storage
```

update_storage

Update details concerning a storage account. Any of the options available in `create_storage` can be used, but the name cannot be changed.

```
salt-cloud -f update_storage my-azure name=my-storage label=my-storage
```

delete_storage

Delete a specific storage account.

```
salt-cloud -f delete_storage my-azure name=my-storage
```

show_storage_keys

Returns the primary and secondary access keys for the specified storage account.

```
salt-cloud -f show_storage_keys my-azure name=my-storage
```

regenerate_storage_keys

Regenerate storage account keys. Requires a key_type (``primary'' or ``secondary'') to be specified.

```
salt-cloud -f regenerate_storage_keys my-azure name=my-storage key_type=primary
```

Managing Disks

New in version 2015.8.0.

When a VM is created, a disk will also be created for it. The following functions are available for managing disks. Deprecated marked as deprecated are marked as such as per the SDK documentation, but are still included for completeness with the SDK.

show_disk

Return details about a specific disk. Can also be called with `get_disk`.

```
salt-cloud -f show_disk my-azure name=my-disk
```

list_disks

List all disks associates with the account.

```
salt-cloud -f list_disks my-azure
```

update_disk

Update details for a disk. The following options are available.

name

Required. The name of the disk to update.

has_operating_system

Deprecated.

label

Required. The label for the disk.

media_link

Deprecated. The location of the disk in the account, including the storage container that it is in. This should not need to be changed.

new_name

Deprecated. If renaming the disk, the new name.

os

Deprecated.

CLI Example

The following example illustrates updating a disk.

```
salt-cloud -f update_disk my-azure name=my-disk label=my-disk
```

delete_disk

Delete a specific disk.

```
salt-cloud -f delete_disk my-azure name=my-disk
```

Managing Service Certificates

New in version 2015.8.0.

Stored at the cloud service level, these certificates are used by your deployed services. For more information on service certificates, see the following link:

- Manage Certificates

The following functions are available.

list_service_certificates

List service certificates associated with the account.

```
salt-cloud -f list_service_certificates my-azure
```

show_service_certificate

Show the data for a specific service certificate associated with the account. The `name`, `thumbprint`, and `thumbalgorithm` can be obtained from `list_service_certificates`. Can also be called with `get_service_certificate`.

```
salt-cloud -f show_service_certificate my-azure name=my_service_certificate \
    thumbalgorithm=sha1 thumbprint=0123456789ABCDEF
```

add_service_certificate

Add a service certificate to the account. This requires that a certificate already exists, which is then added to the account. For more information on creating the certificate itself, see:

- Create a Service Certificate for Azure

The following options are available.

name

Required. The name of the hosted service that the certificate will belong to.

data

Required. The base-64 encoded form of the pfx file.

certificate_format

Required. The service certificate format. The only supported value is pfx.

password

The certificate password.

```
salt-cloud -f add_service_certificate my-azure name=my-cert \
    data='...CERT_DATA...' certificate_format=pfx password=verybadpass
```

delete_service_certificate

Delete a service certificate from the account. The `name`, `thumbprint`, and `thumbalgorithm` can be obtained from `list_service_certificates`.

```
salt-cloud -f delete_service_certificate my-azure \
    name=my_service_certificate \
    thumbalgorithm=sha1 thumbprint=0123456789ABCDEF
```

Managing Management Certificates

New in version 2015.8.0.

A Azure management certificate is an X.509 v3 certificate used to authenticate an agent, such as Visual Studio Tools for Windows Azure or a client application that uses the Service Management API, acting on behalf of the subscription owner to manage subscription resources. Azure management certificates are uploaded to Azure and stored at the subscription level. The management certificate store can hold up to 100 certificates per subscription. These certificates are used to authenticate your Windows Azure deployment.

For more information on management certificates, see the following link.

- Manage Certificates

The following functions are available.

list_management_certificates

List management certificates associated with the account.

```
salt-cloud -f list_management_certificates my-azure
```

show_management_certificate

Show the data for a specific management certificate associated with the account. The `name`, `thumbprint`, and `thumbalgorithm` can be obtained from `list_management_certificates`. Can also be called with `get_management_certificate`.

```
salt-cloud -f show_management_certificate my-azure name=my_management_certificate \
    thumbalgorithm=sha1 thumbprint=0123456789ABCDEF
```

add_management_certificate

Management certificates must have a key length of at least 2048 bits and should reside in the Personal certificate store. When the certificate is installed on the client, it should contain the private key of the certificate. To upload to the certificate to the Microsoft Azure Management Portal, you must export it as a .cer format file that does not contain the private key. For more information on creating management certificates, see the following link:

- Create and Upload a Management Certificate for Azure

The following options are available.

public_key

A base64 representation of the management certificate public key.

thumbprint

The thumb print that uniquely identifies the management certificate.

data

The certificate's raw data in base-64 encoded .cer format.

```
salt-cloud -f add_management_certificate my-azure public_key='...PUBKEY...' \
    thumbprint=0123456789ABCDEF data='...CERT_DATA...'
```

delete_management_certificate

Delete a management certificate from the account. The `thumbprint` can be obtained from `list_management_certificates`.

```
salt-cloud -f delete_management_certificate my-azure thumbprint=0123456789ABCDEF
```

Virtual Network Management

New in version 2015.8.0.

The following are functions for managing virtual networks.

list_virtual_networks

List input endpoints associated with the deployment.

```
salt-cloud -f list_virtual_networks my-azure service=myservice deployment=mydeployment
```

Managing Input Endpoints

New in version 2015.8.0.

Input endpoints are used to manage port access for roles. Because endpoints cannot be managed by the Azure Python SDK, Salt Cloud uses the API directly. With versions of Python before 2.7.9, the `requests-python` package needs to be installed in order for this to work. Additionally, the following needs to be set in the master's configuration file:

```
backend: requests
```

The following functions are available.

list_input_endpoints

List input endpoints associated with the deployment

```
salt-cloud -f list_input_endpoints my-azure service=myservice deployment=mydeployment
```

show_input_endpoint

Show an input endpoint associated with the deployment

```
salt-cloud -f show_input_endpoint my-azure service=myservice \
    deployment=mydeployment name=SSH
```

add_input_endpoint

Add an input endpoint to the deployment. Please note that there may be a delay before the changes show up. The following options are available.

service

Required. The name of the hosted service which the VM belongs to.

deployment

Required. The name of the deployment that the VM belongs to. If the VM was created with Salt Cloud, the deployment name probably matches the VM name.

role

Required. The name of the role that the VM belongs to. If the VM was created with Salt Cloud, the role name probably matches the VM name.

name

Required. The name of the input endpoint. This typically matches the port that the endpoint is set to. For instance, port 22 would be called SSH.

port

Required. The public (Internet-facing) port that is used for the endpoint.

local_port

Optional. The private port on the VM itself that will be matched with the port. This is typically the same as the `port`. If this value is not specified, it will be copied from `port`.

protocol

Required. Either `tcp` or `udp`.

enable_direct_server_return

Optional. If an internal load balancer exists in the account, it can be used with a direct server return. The default value is `False`. Please see the following article for an explanation of this option.

- Load Balancing for Azure Infrastructure Services

timeout_for_tcp_idle_connection

Optional. The default value is 4. Please see the following article for an explanation of this option.

- Configurable Idle Timeout for Azure Load Balancer

CLI Example

The following example illustrates adding an input endpoint.

```
salt-cloud -f add_input_endpoint my-azure service=myservice \
    deployment=mydeployment role=myrole name=HTTP local_port=80 \
    port=80 protocol=tcp enable_direct_server_return=False \
    timeout_for_tcp_idle_connection=4
```

update_input_endpoint

Updates the details for a specific input endpoint. All options from `add_input_endpoint` are supported.

```
salt-cloud -f update_input_endpoint my-azure service=myservice \
    deployment=mydeployment role=myrole name=HTTP local_port=80 \
    port=80 protocol=tcp enable_direct_server_return=False \
    timeout_for_tcp_idle_connection=4
```

delete_input_endpoint

Delete an input endpoint from the deployment. Please note that there may be a delay before the changes show up. The following items are required.

CLI Example

The following example illustrates deleting an input endpoint.

service

The name of the hosted service which the VM belongs to.

deployment

The name of the deployment that the VM belongs to. If the VM was created with Salt Cloud, the deployment name probably matches the VM name.

role

The name of the role that the VM belongs to. If the VM was created with Salt Cloud, the role name probably matches the VM name.

name

The name of the input endpoint. This typically matches the port that the endpoint is set to. For instance, port 22 would be called SSH.

```
salt-cloud -f delete_input_endpoint my-azure service=myservice \
    deployment=mydeployment role=myrole name=HTTP
```

Managing Affinity Groups

New in version 2015.8.0.

Affinity groups allow you to group your Azure services to optimize performance. All services and VMs within an affinity group will be located in the same region. For more information on Affinity groups, see the following link:

- Create an Affinity Group in the Management Portal

The following functions are available.

list_affinity_groups

List input endpoints associated with the account

```
salt-cloud -f list_affinity_groups my-azure
```

show_affinity_group

Show an affinity group associated with the account

```
salt-cloud -f show_affinity_group my-azure service=myservice \
    deployment=mydeployment name=SSH
```

create_affinity_group

Create a new affinity group. The following options are supported.

name

Required. The name of the new affinity group.

location

Required. The region in which the affinity group lives.

label

Required. A label describing the new affinity group.

description

Optional. A longer description of the affinity group.

```
salt-cloud -f create_affinity_group my-azure name=my_affinity_group \
   label=my-affinity-group location='West US'
```

update_affinity_group

Update an affinity group's properties

```
salt-cloud -f update_affinity_group my-azure name=my_group label=my_group
```

delete_affinity_group

Delete a specific affinity group associated with the account

```
salt-cloud -f delete_affinity_group my-azure name=my_affinity_group
```

Managing Blob Storage

New in version 2015.8.0.

Azure storage containers and their contents can be managed with Salt Cloud. This is not as elegant as using one of the other available clients in Windows, but it benefits Linux and Unix users, as there are fewer options available on those platforms.

Blob Storage Configuration

Blob storage must be configured differently than the standard Azure configuration. Both a `storage_account` and a `storage_key` must be specified either through the Azure provider configuration (in addition to the other Azure configuration) or via the command line.

```
storage_account: mystorage
storage_key: ffhj334fDSGFEGDFGFDewr34fwfsFSDFwe==
```

storage_account

This is one of the storage accounts that is available via the `list_storage` function.

storage_key

Both a primary and a secondary `storage_key` can be obtained by running the `show_storage_keys` function. Either key may be used.

Blob Functions

The following functions are made available through Salt Cloud for managing blog storage.

make_blob_url

Creates the URL to access a blob

```
salt-cloud -f make_blob_url my-azure container=mycontainer blob=myblob
```

container

Name of the container.

blob

Name of the blob.

account

Name of the storage account. If not specified, derives the host base from the provider configuration.

protocol

Protocol to use: `http' or `https'. If not specified, derives the host base from the provider configuration.

host_base

Live host base URL. If not specified, derives the host base from the provider configuration.

list_storage_containers

List containers associated with the storage account

```
salt-cloud -f list_storage_containers my-azure
```

create_storage_container

Create a storage container

```
salt-cloud -f create_storage_container my-azure name=mycontainer
```

name

Name of container to create.

meta_name_values

Optional. A dict with name_value pairs to associate with the container as metadata. Example:{`Category':'test'}

blob_public_access

Optional. Possible values include: container, blob

fail_on_exist

Specify whether to throw an exception when the container exists.

show_storage_container

Show a container associated with the storage account

```
salt-cloud -f show_storage_container my-azure name=myservice
```

name

Name of container to show.

show_storage_container_metadata

Show a storage container's metadata

```
salt-cloud -f show_storage_container_metadata my-azure name=myservice
```

name

Name of container to show.

lease_id

If specified, show_storage_container_metadata only succeeds if the container's lease is active and matches this ID.

set_storage_container_metadata

Set a storage container's metadata

```
salt-cloud -f set_storage_container my-azure name=mycontainer \
    x_ms_meta_name_values='{"my_name": "my_value"}'
```

name

Name of existing container. meta_name_values ````````````` A dict containing name, value for metadata. Example: {`category':'test'} lease_id ```` If specified, set_storage_container_metadata only succeeds if the container's lease is active and matches this ID.

show_storage_container_acl

Show a storage container's acl

```
salt-cloud -f show_storage_container_acl my-azure name=myservice
```

name

Name of existing container.

lease_id

If specified, show_storage_container_acl only succeeds if the container's lease is active and matches this ID.

set_storage_container_acl

Set a storage container's acl

```
salt-cloud -f set_storage_container my-azure name=mycontainer
```

name

Name of existing container.

signed_identifiers

SignedIdentifers instance

blob_public_access

Optional. Possible values include: container, blob

lease_id

If specified, set_storage_container_acl only succeeds if the container's lease is active and matches this ID.

delete_storage_container

Delete a container associated with the storage account

```
salt-cloud -f delete_storage_container my-azure name=mycontainer
```

name

Name of container to create.

fail_not_exist

Specify whether to throw an exception when the container exists.

lease_id

If specified, delete_storage_container only succeeds if the container's lease is active and matches this ID.

lease_storage_container

Lease a container associated with the storage account

```
salt-cloud -f lease_storage_container my-azure name=mycontainer
```

name

Name of container to create.

lease_action

Required. Possible values: acquire|renew|release|break|change

lease_id

Required if the container has an active lease.

lease_duration

Specifies the duration of the lease, in seconds, or negative one (-1) for a lease that never expires. A non-infinite lease can be between 15 and 60 seconds. A lease duration cannot be changed using renew or change. For backwards compatibility, the default is 60, and the value is only used on an acquire operation.

lease_break_period

Optional. For a break operation, this is the proposed duration of seconds that the lease should continue before it is broken, between 0 and 60 seconds. This break period is only used if it is shorter than the time remaining on the lease. If longer, the time remaining on the lease is used. A new lease will not be available before the break period has expired, but the lease may be held for longer than the break period. If this header does not appear with a break operation, a fixed-duration lease breaks after the remaining lease period elapses, and an infinite lease breaks immediately.

proposed_lease_id

Optional for acquire, required for change. Proposed lease ID, in a GUID string format.

list_blobs

List blobs associated with the container

```
salt-cloud -f list_blobs my-azure container=mycontainer
```

container

The name of the storage container

prefix

Optional. Filters the results to return only blobs whose names begin with the specified prefix.

marker

Optional. A string value that identifies the portion of the list to be returned with the next list operation. The operation returns a marker value within the response body if the list returned was not complete. The marker value may then be used in a subsequent call to request the next set of list items. The marker value is opaque to the client.

maxresults

Optional. Specifies the maximum number of blobs to return, including all BlobPrefix elements. If the request does not specify maxresults or specifies a value greater than 5,000, the server will return up to 5,000 items. Setting maxresults to a value less than or equal to zero results in error response code 400 (Bad Request).

include

Optional. Specifies one or more datasets to include in the response. To specify more than one of these options on the URI, you must separate each option with a comma. Valid values are:

```
snapshots:
    Specifies that snapshots should be included in the
    enumeration. Snapshots are listed from oldest to newest in
    the response.
metadata:
    Specifies that blob metadata be returned in the response.
uncommittedblobs:
    Specifies that blobs for which blocks have been uploaded,
    but which have not been committed using Put Block List
    (REST API), be included in the response.
copy:
    Version 2012-02-12 and newer. Specifies that metadata
    related to any current or previous Copy Blob operation
    should be included in the response.
```

delimiter

Optional. When the request includes this parameter, the operation returns a BlobPrefix element in the response body that acts as a placeholder for all blobs whose names begin with the same substring up to the appearance of the delimiter character. The delimiter may be a single character or a string.

show_blob_service_properties

Show a blob's service properties

```
salt-cloud -f show_blob_service_properties my-azure
```

set_blob_service_properties

Sets the properties of a storage account's Blob service, including Windows Azure Storage Analytics. You can also use this operation to set the default request version for all incoming requests that do not have a version specified.

```
salt-cloud -f set_blob_service_properties my-azure
```

properties

a StorageServiceProperties object.

timeout

Optional. The timeout parameter is expressed in seconds.

show_blob_properties

Returns all user-defined metadata, standard HTTP properties, and system properties for the blob.

```
salt-cloud -f show_blob_properties my-azure container=mycontainer blob=myblob
```

container

Name of existing container.

blob

Name of existing blob.

lease_id

Required if the blob has an active lease.

set_blob_properties

Set a blob's properties

```
salt-cloud -f set_blob_properties my-azure
```

container

Name of existing container.

blob

Name of existing blob.

blob_cache_control

Optional. Modifies the cache control string for the blob.

blob_content_type

Optional. Sets the blob's content type.

blob_content_md5

Optional. Sets the blob's MD5 hash.

blob_content_encoding

Optional. Sets the blob's content encoding.

blob_content_language

Optional. Sets the blob's content language.

lease_id

Required if the blob has an active lease.

blob_content_disposition

Optional. Sets the blob's Content-Disposition header. The Content-Disposition response header field conveys additional information about how to process the response payload, and also can be used to attach additional metadata. For example, if set to attachment, it indicates that the user-agent should not display the response, but instead show a Save As dialog with a filename other than the blob name specified.

put_blob

Upload a blob

```
salt-cloud -f put_blob my-azure container=base name=top.sls blob_path=/srv/salt/top.sls
salt-cloud -f put_blob my-azure container=base name=content.txt blob_content='Some
↪content'
```

container

Name of existing container.

name

Name of existing blob.

blob_path

The path on the local machine of the file to upload as a blob. Either this or blob_content must be specified.

blob_content

The actual content to be uploaded as a blob. Either this or blob_path must me specified.

cache_control

Optional. The Blob service stores this value but does not use or modify it.

content_language

Optional. Specifies the natural languages used by this resource.

content_md5

Optional. An MD5 hash of the blob content. This hash is used to verify the integrity of the blob during transport. When this header is specified, the storage service checks the hash that has arrived with the one that was sent. If the two hashes do not match, the operation will fail with error code 400 (Bad Request).

blob_content_type

Optional. Set the blob's content type.

blob_content_encoding

Optional. Set the blob's content encoding.

blob_content_language

Optional. Set the blob's content language.

blob_content_md5

Optional. Set the blob's MD5 hash.

blob_cache_control

Optional. Sets the blob's cache control.

meta_name_values

A dict containing name, value for metadata.

lease_id

Required if the blob has an active lease.

get_blob

Download a blob

```
salt-cloud -f get_blob my-azure container=base name=top.sls local_path=/srv/salt/top.
↪sls
salt-cloud -f get_blob my-azure container=base name=content.txt return_content=True
```

container

Name of existing container.

name

Name of existing blob.

local_path

The path on the local machine to download the blob to. Either this or return_content must be specified.

return_content

Whether or not to return the content directly from the blob. If specified, must be True or False. Either this or the local_path must be specified.

snapshot

Optional. The snapshot parameter is an opaque DateTime value that, when present, specifies the blob snapshot to retrieve.

lease_id

Required if the blob has an active lease.

progress_callback

callback for progress with signature function(current, total) where current is the number of bytes transferred so far, and total is the size of the blob.

max_connections

Maximum number of parallel connections to use when the blob size exceeds 64MB. Set to 1 to download the blob chunks sequentially. Set to 2 or more to download the blob chunks in parallel. This uses more system resources but will download faster.

max_retries

Number of times to retry download of blob chunk if an error occurs.

retry_wait

Sleep time in secs between retries.

13.7.3 Getting Started With Azure ARM

New in version 2016.11.0.

Azure is a cloud service by Microsoft providing virtual machines, SQL services, media services, and more. Azure ARM (aka, the Azure Resource Manager) is a next generation version of the Azure portal and API. This document describes how to use Salt Cloud to create a virtual machine on Azure ARM, with Salt installed.

More information about Azure is located at http://www.windowsazure.com/.

Dependencies

- Azure Cli `` `pip install 'azure-cli>=2.0.12'` ``
- A Microsoft Azure account
- Salt

Installation Tips

Because the `azure` library requires the `cryptography` library, which is compiled on-the-fly by `pip`, you may need to install the development tools for your operating system.

Before you install `azure` with `pip`, you should make sure that the required libraries are installed.

Debian

For Debian and Ubuntu, the following command will ensure that the required dependencies are installed:

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

Red Hat

For Fedora and RHEL-derivatives, the following command will ensure that the required dependencies are installed:

```
sudo yum install gcc libffi-devel python-devel openssl-devel
```

Configuration

Set up the provider config at `/etc/salt/cloud.providers.d/azurearm.conf`:

```
# Note: This example is for /etc/salt/cloud.providers.d/azurearm.conf

my-azurearm-config:
  driver: azurearm
  master: salt.example.com
  subscription_id: 01234567-890a-bcde-f012-34567890abdc

  # https://apps.dev.microsoft.com/#/appList
  username: <username>@<subdomain>.onmicrosoft.com
  password: verybadpass
  location: westus
  resource_group: my_rg

  # Optional
  network_resource_group: my_net_rg
  cleanup_disks: True
  cleanup_vhds: True
  cleanup_data_disks: True
  cleanup_interfaces: True
  custom_data: 'This is custom data'
  expire_publisher_cache: 604800  # 7 days
  expire_offer_cache: 518400  # 6 days
  expire_sku_cache: 432000  # 5 days
  expire_version_cache: 345600  # 4 days
  expire_group_cache: 14400  # 4 hours
  expire_interface_cache: 3600  # 1 hour
  expire_network_cache: 3600  # 1 hour
```

Cloud Profiles

Set up an initial profile at /etc/salt/cloud.profiles:

```
azure-ubuntu:
  provider: my-azure-config
  image: Canonical|UbuntuServer|14.04.5-LTS|14.04.201612050
  size: Standard_D1_v2
  location: eastus
  ssh_username: azureuser
  ssh_password: verybadpass

azure-win2012:
  provider: my-azure-config
  image: MicrosoftWindowsServer|WindowsServer|2012-R2-Datacenter|latest
  size: Standard_D1_v2
  location: westus
  win_username: azureuser
  win_password: verybadpass
```

These options are described in more detail below. Once configured, the profile can be realized with a salt command:

```
salt-cloud -p azure-ubuntu newinstance
```

This will create an salt minion instance named newinstance in Azure. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
salt newinstance test.ping
```

Profile Options

The following options are currently available for Azure ARM.

provider

The name of the provider as configured in */etc/salt/cloud.providers.d/azure.conf*.

image

Required. The name of the image to use to create a VM. Available images can be viewed using the following command:

```
salt-cloud --list-images my-azure-config
```

As you will see in --list-images, image names are comprised of the following fields, separated by the pipe (|) character:

```
publisher: For example, Canonical or MicrosoftWindowsServer
offer: For example, UbuntuServer or WindowsServer
sku: Such as 14.04.5-LTS or 2012-R2-Datacenter
version: Such as 14.04.201612050 or latest
```

It is possible to specify the URL of a custom image that you have access to, such as:

```
https://<mystorage>.blob.core.windows.net/system/Microsoft.Compute/Images/<mystorage>/
↪template-osDisk.01234567-890a-bcdef0123-4567890abcde.vhd
```

size

Required. The name of the size to use to create a VM. Available sizes can be viewed using the following command:

```
salt-cloud --list-sizes my-azure-config
```

location

Required. The name of the location to create a VM in. Available locations can be viewed using the following command:

```
salt-cloud --list-locations my-azure-config
```

ssh_username

Required for Linux. The user to use to log into the newly-created Linux VM to install Salt.

ssh_password

Required for Linux. The password to use to log into the newly-created Linux VM to install Salt.

win_username

Required for Windows. The user to use to log into the newly-created Windows VM to install Salt.

win_password

Required for Windows. The password to use to log into the newly-created Windows VM to install Salt.

win_installer

Required for Windows. The path to the Salt installer to be uploaded.

resource_group

Required. The resource group that all VM resources (VM, network interfaces, etc) will be created in.

network_resource_group

Optional. If specified, then the VM will be connected to the network resources in this group, rather than the group that it was created in. The VM interfaces and IPs will remain in the configured `resource_group` with the VM.

network

Required. The virtual network that the VM will be spun up in.

subnet

Optional. The subnet inside the virtual network that the VM will be spun up in. Default is `default`.

load_balancer

Optional. The load-balancer for the VM's network interface to join. If specified the backend_pool option need to be set.

backend_pool

Optional. Required if the load_balancer option is set. The load-balancer's Backend Pool the VM's network interface will join.

iface_name

Optional. The name to apply to the VM's network interface. If not supplied, the value will be set to `<VM name>-iface0`.

dns_servers

Optional. A **list** of the DNS servers to configure for the network interface (will be set on the VM by the DHCP of the VNET).

```
my-azurearm-profile:
  provider: azurearm-provider
  network: mynetwork
  dns_servers:
    - 10.1.1.4
    - 10.1.1.5
```

availability_set

Optional. If set, the VM will be added to the specified availability set.

cleanup_disks

Optional. Default is `False`. If set to `True`, disks will be cleaned up when the VM that they belong to is deleted.

cleanup_vhds

Optional. Default is `False`. If set to `True`, VHDs will be cleaned up when the VM and disk that they belong to are deleted. Requires `cleanup_disks` to be set to `True`.

cleanup_data_disks

Optional. Default is `False`. If set to `True`, data disks (non-root volumes) will be cleaned up whtn the VM that they are attached to is deleted. Requires `cleanup_disks` to be set to `True`.

cleanup_interfaces

Optional. Default is `False`. Normally when a VM is deleted, its associated interfaces and IPs are retained. This is useful if you expect the deleted VM to be recreated with the same name and network settings. If you would like interfaces and IPs to be deleted when their associated VM is deleted, set this to `True`.

userdata

Optional. Any custom cloud data that needs to be specified. How this data is used depends on the operating system and image that is used. For instance, Linux images that use `cloud-init` will import this data for use with that program. Some Windows images will create a file with a copy of this data, and others will ignore it. If a Windows image creates a file, then the location will depend upon the version of Windows. This will be ignored if the `userdata_file` is specified.

userdata_file

Optional. The path to a file to be read and submitted to Azure as user data. How this is used depends on the operating system that is being deployed. If used, any `userdata` setting will be ignored.

wait_for_ip_timeout

Optional. Default is `600`. When waiting for a VM to be created, Salt Cloud will attempt to connect to the VM's IP address until it starts responding. This setting specifies the maximum time to wait for a response.

wait_for_ip_interval

Optional. Default is `10`. How long to wait between attempts to connect to the VM's IP.

wait_for_ip_interval_multiplier

Optional. Default is `1`. Increase the interval by this multiplier after each request; helps with throttling.

expire_publisher_cache

Optional. Default is `604800`. When fetching image data using `--list-images`, a number of web calls need to be made to the Azure ARM API. This is normally very fast when performed using a VM that exists inside Azure itself, but can be very slow when made from an external connection.

By default, the publisher data will be cached, and only updated every `604800` seconds (7 days). If you need the publisher cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the publisher cache.

expire_offer_cache

Optional. Default is `518400`. See `expire_publisher_cache` for details on why this exists.

By default, the offer data will be cached, and only updated every `518400` seconds (6 days). If you need the offer cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the publiser cache.

expire_sku_cache

Optional. Default is `432000`. See `expire_publisher_cache` for details on why this exists.

By default, the sku data will be cached, and only updated every `432000` seconds (5 days). If you need the sku cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the sku cache.

expire_version_cache

Optional. Default is `345600`. See `expire_publisher_cache` for details on why this exists.

By default, the version data will be cached, and only updated every `345600` seconds (4 days). If you need the version cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the version cache.

expire_group_cache

Optional. Default is `14400`. See `expire_publisher_cache` for details on why this exists.

By default, the resource group data will be cached, and only updated every `14400` seconds (4 hours). If you need the resource group cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the resource group cache.

expire_interface_cache

Optional. Default is `3600`. See `expire_publisher_cache` for details on why this exists.

By default, the interface data will be cached, and only updated every `3600` seconds (1 hour). If you need the interface cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the interface cache.

expire_network_cache

Optional. Default is `3600`. See `expire_publisher_cache` for details on why this exists.

By default, the network data will be cached, and only updated every `3600` seconds (1 hour). If you need the network cache to be updated at a different frequency, change this setting. Setting it to `0` will turn off the network cache.

Other Options

Other options relevant to Azure ARM.

storage_account

Required for actions involving an Azure storage account.

storage_key

Required for actions involving an Azure storage account.

Show Instance

This action is a thin wrapper around `--full-query`, which displays details on a single instance only. In an environment with several machines, this will save a user from having to sort through all instance data, just to examine a single instance.

```
salt-cloud -a show_instance myinstance
```

13.7.4 Getting Started with CloudStack

CloudStack is one the most popular cloud projects. It's an open source project to build public and/or private clouds. You can use Salt Cloud to launch CloudStack instances.

Dependencies

- Libcloud >= 0.13.2

Configuration

Using Salt for CloudStack, requires an `API key` and a `secret key` along with the API address endpoint information.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

exoscale:
  driver: cloudstack
  host: api.exoscale.ch
  path: /compute
```

```
  apikey: EXOAPIKEY
  secretkey: EXOSECRETKEYINYOURACCOUNT
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory:

```
exoscale-ubuntu:
  provider: exoscale-config
  image: Ubuntu 16.04
  size: Small
  location: ch-gva-2
```

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
# salt-cloud --list-locations exoscale-config
exoscale:
    ----------
    cloudstack:
        ----------
        ch-dk-2:
            ----------
            country:
                Unknown
            driver:
            id:
                91e5e9e4-c9ed-4b76-bee4-427004b3baf9
            name:
                ch-dk-2
        ch-gva-2:
            ----------
            country:
                Unknown
            driver:
            id:
                1128bd56-b4d9-4ac6-a7b9-c715b187ce11
            name:
                ch-gva-2
```

Sizes can be obtained using the `--list-sizes` option for the `salt-cloud` command:

```
# salt-cloud --list-sizes exoscale
exoscale:
    ----------
```

```
    cloudstack:
        ----------
        Extra-large:
            ----------
            bandwidth:
                0
            disk:
                0
            driver:
            extra:
                ----------
                cpu:
                    4
            get_uuid:
            id:
                350dc5ea-fe6d-42ba-b6c0-efb8b75617ad
            name:
                Extra-large
            price:
                0
            ram:
                16384
            uuid:
                edb4cd4ae14bbf152d451b30c4b417ab095a5bfe
...SNIP...
```

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images exoscale
exoscale:
    ----------
    cloudstack:
        ----------
        Linux CentOS 6.6 64-bit:
            ----------
            driver:
            extra:
                ----------
                displaytext:
                    Linux CentOS 6.6 64-bit 10G Disk (2014-12-01-bac8e0)
                format:
                    QCOW2
                hypervisor:
                    KVM
                os:
                    Other PV (64-bit)
                size:
                    10737418240
            get_uuid:
            id:
                aa69ae64-1ea9-40af-8824-c2c3344e8d7c
            name:
                Linux CentOS 6.6 64-bit
            uuid:
                f26b4f54ec8591abdb6b5feb3b58f720aa438fee
...SNIP...
```

CloudStack specific settings

security_group

New in version next-release.

You can specify a list of security groups (by name or id) that should be assigned to the VM.

```
exoscale:
  provider: cloudstack
  security_group:
    - default
    - salt-master
```

13.7.5 Getting Started With DigitalOcean

DigitalOcean is a public cloud host that specializes in Linux instances.

Configuration

Using Salt for DigitalOcean requires a personal_access_token, an ssh_key_file, and at least one SSH key name in ssh_key_names. More ssh_key_names can be added by separating each key with a comma. The personal_access_token can be found in the DigitalOcean web interface in the ``Apps & API'' section. The SSH key name can be found under the ``SSH Keys'' section.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-digitalocean-config:
  driver: digitalocean
  personal_access_token: xxx
  ssh_key_file: /path/to/ssh/key/file
  ssh_key_names: my-key-name,my-key-name-2
  location: New York 1
```

Note: Changed in version 2015.8.0.

The provider parameter in cloud provider definitions was renamed to driver. This change was made to avoid confusion with the provider parameter that is used in cloud profile definitions. Cloud provider definitions now use driver to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use provider to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at /etc/salt/cloud.profiles or in the /etc/salt/cloud.profiles.d/ directory:

```
digitalocean-ubuntu:
  provider: my-digitalocean-config
  image: 14.04 x64
  size: 512MB
  location: New York 1
  private_networking: True
  backups_enabled: True
  ipv6: True
  create_dns_record: True
  userdata_file: /etc/salt/cloud.userdata.d/setup
  tags:
    - tag1
    - tag2
    - tag3
```

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
# salt-cloud --list-locations my-digitalocean-config
my-digitalocean-config:
    ----------
    digitalocean:
        ----------
        Amsterdam 1:
            ----------
            available:
                False
            features:
                [u'backups']
            name:
                Amsterdam 1
            sizes:
                []
            slug:
                ams1
...SNIP...
```

Sizes can be obtained using the `--list-sizes` option for the `salt-cloud` command:

```
# salt-cloud --list-sizes my-digitalocean-config
my-digitalocean-config:
    ----------
    digitalocean:
        ----------
        512MB:
            ----------
            cost_per_hour:
                0.00744
            cost_per_month:
                5.0
            cpu:
                1
            disk:
                20
            id:
                66
            memory:
                512
            name:
```

```
                512MB
            slug:
                None
...SNIP...
```

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images my-digitalocean-config
my-digitalocean-config:
    ----------
    digitalocean:
        ----------
        10.1:
            ----------
            created_at:
                2015-01-20T20:04:34Z
            distribution:
                FreeBSD
            id:
                10144573
            min_disk_size:
                20
            name:
                10.1
            public:
                True
...SNIP...
```

Profile Specifics:

ssh_username

If using a FreeBSD image from DigitalOcean, you'll need to set the `ssh_username` setting to `freebsd` in your profile configuration.

```
digitalocean-freebsd:
  provider: my-digitalocean-config
  image: 10.2
  size: 512MB
  ssh_username: freebsd
```

userdata_file

New in version 2016.11.6.

Use *userdata_file* to specify the userdata file to upload for use with cloud-init if available.

```
my-openstack-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/cloud-init/packages.yml
```

```
my-do-config:
  # Pass userdata to the instance to be created
```

```
  userdata_file: /etc/salt/cloud-init/packages.yml
  userdata_template: jinja
```

If no `userdata_template` is set in the cloud profile, then the master configuration will be checked for a *userdata_template* value. If this is not set, then no templating will be performed on the userdata_file.

To disable templating in a cloud profile when a *userdata_template* has been set in the master configuration file, simply set `userdata_template` to `False` in the cloud profile:

```
my-do-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/cloud-init/packages.yml
  userdata_template: False
```

Miscellaneous Information

Note: DigitalOcean's concept of `Applications` is nothing more than a pre-configured instance (same as a normal Droplet). You will find examples such `Docker 0.7 Ubuntu 13.04 x64` and `Wordpress on Ubuntu 12.10` when using the `--list-images` option. These names can be used just like the rest of the standard instances when specifying an image in the cloud profile configuration.

Note: If your domain's DNS is managed with DigitalOcean, and your minion name matches your DigitalOcean managed DNS domain, you can automatically create A and AAA records for newly created droplets. Use `create_dns_record: True` in your config to enable this. Adding `delete_dns_record: True` to also delete records when a droplet is destroyed is optional. Due to limitations in salt-cloud design, the destroy code does not have access to the VM config data. WHETHER YOU ADD `create_dns_record: True` OR NOT, salt-cloud WILL attempt to delete your DNS records if the minion name matches. This will prevent advertising any recycled IP addresses for destroyed minions.

Note: If you need to perform the bootstrap using the local interface for droplets, this can be done by setting `ssh_interface: private` in your config. By default the salt-cloud script would run on the public interface however if firewall is preventing the connection to the Droplet over the public interface you might need to set this option to connect via private interface. Also, to use this feature `private_networking: True` must be set in the config.

Note: Additional documentation is available from DigitalOcean.

13.7.6 Getting Started With Dimension Data Cloud

Dimension Data are a global IT Services company and form part of the NTT Group. Dimension Data provide IT-as-a-Service to customers around the globe on their cloud platform (Compute as a Service). The CaaS service is available either on one of the public cloud instances or as a private instance on premises.

http://cloud.dimensiondata.com/

CaaS has its own non-standard API , SaltStack provides a wrapper on top of this API with common methods with other IaaS solutions and Public cloud providers. Therefore, you can use the Dimension Data module to communicate with both the public and private clouds.

Dependencies

This driver requires the Python `apache-libcloud` and `netaddr` library to be installed.

Configuration

When you instantiate a driver you need to pass the following arguments to the driver constructor:

- `user_id` - Your Dimension Data Cloud username
- `key` - Your Dimension Data Cloud password
- `region` - The region key, one of the possible region keys

Possible regions:

- `dd-na` : Dimension Data North America (USA)
- `dd-eu` : Dimension Data Europe
- `dd-af` : Dimension Data Africa
- `dd-au` : Dimension Data Australia
- `dd-latam` : Dimension Data Latin America
- `dd-ap` : Dimension Data Asia Pacific
- `dd-canada` : Dimension Data Canada region

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-dimensiondata-config:
  user_id: my_username
  key: myPassword!
  region: dd-na
  driver: dimensiondata
```

Note: In version 2015.8.0, the `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profiles

Cloud Profiles

Dimension Data images have an inbuilt size configuration, there is no list of sizes (although, if the command --list-sizes is run a default will be returned).

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images my-dimensiondata-config
my-dimensiondata-config:
----------
dimensiondata:
    ----------
    CSfM SharePoint 2013 Trial:
        ----------
        driver:
        extra:
            ----------
            OS_displayName:
                WIN2012R2S/64
            OS_type:
                None
            cpu:
            created:
                2015-03-19T18:36:06.000Z
            description:
                Windows 2012 R2 Standard 64-bit installed with SharePoint 2013 and
↪Visual Studio 2013 Pro (Trial Version)
            location:
            memoryGb:
                12
            osImageKey:
                T-WIN-2012R2-STD-SP2013-VS2013-64-4-12-100
        get_uuid:
        id:
            0df4677e-d380-4e9b-9469-b529ee0214c5
        name:
            CSfM SharePoint 2013 Trial
        uuid:
            28c077f1be970ee904541407b377e3ff87a9ac69
    CentOS 5 32-bit 2 CPU:
        ----------
        driver:
        extra:
            ----------
            OS_displayName:
                CENTOS5/32
            OS_type:
                None
            cpu:
            created:
                2015-10-21T14:52:29.000Z
            description:
                CentOS Release 5.11 32-bit
            location:
            memoryGb:
                4
            osImageKey:
                T-CENT-5-32-2-4-10
        get_uuid:
        id:
            a8046bd1-04ea-4668-bf32-bf8d5540faed
        name:
            CentOS 5 32-bit 2 CPU
        uuid:
```

```
            4d7dd59929fed6f4228db861b609da64997773a7

...SNIP...
```

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
my-dimensiondata-config:
    ----------
    dimensiondata:
        ----------
        Australia - Melbourne:
            ----------
            country:
                Australia
            driver:
            id:
                AU2
            name:
                Australia - Melbourne
        Australia - Melbourne MCP2:
            ----------
            country:
                Australia
            driver:
            id:
                AU10
            name:
                Australia - Melbourne MCP2
        Australia - Sydney:
            ----------
            country:
                Australia
            driver:
            id:
                AU1
            name:
                Australia - Sydney
        Australia - Sydney MCP2:
            ----------
            country:
                Australia
            driver:
            id:
                AU9
            name:
                Australia - Sydney MCP2
        New Zealand:
            ----------
            country:
                New Zealand
            driver:
            id:
                AU8
            name:
                New Zealand
        New_Zealand:
            ----------
            country:
```

```
            New Zealand
        driver:
        id:
            AU11
        name:
            New_Zealand
```

Note: Dimension Data Cloud REST API documentation is available from Dimension Data MCP 2.

13.7.7 Getting Started With AWS EC2

Amazon EC2 is a very widely used public cloud platform and one of the core platforms Salt Cloud has been built to support.

Previously, the suggested driver for AWS EC2 was the `aws` driver. This has been deprecated in favor of the `ec2` driver. Configuration using the old `aws` driver will still function, but that driver is no longer in active development.

Dependencies

This driver requires the Python `requests` library to be installed.

Configuration

The following example illustrates some of the options that can be set. These parameters are discussed in more detail below.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-ec2-southeast-public-ips:
  # Set up the location of the salt master
  #
  minion:
    master: saltmaster.example.com

  # Set up grains information, which will be common for all nodes
  # using this provider
  grains:
    node_type: broker
    release: 1.0.1

  # Specify whether to use public or private IP for deploy script.
  #
  # Valid options are:
  #     private_ips - The salt-cloud command is run inside the EC2
  #     public_ips - The salt-cloud command is run outside of EC2
  #
  ssh_interface: public_ips

  # Optionally configure the Windows credential validation number of
  # retries and delay between retries.  This defaults to 10 retries
  # with a one second delay betwee retries
```

```
  win_deploy_auth_retries: 10
  win_deploy_auth_retry_delay: 1

  # Set the EC2 access credentials (see below)
  #
  id: 'use-instance-role-credentials'
  key: 'use-instance-role-credentials'

  # Make sure this key is owned by corresponding user (default 'salt') with
↪permissions 0400.
  #
  private_key: /etc/salt/my_test_key.pem
  keyname: my_test_key
  securitygroup: default

  # Optionally configure default region
  # Use salt-cloud --list-locations <provider> to obtain valid regions
  #
  location: ap-southeast-1
  availability_zone: ap-southeast-1b

  # Configure which user to use to run the deploy script. This setting is
  # dependent upon the AMI that is used to deploy. It is usually safer to
  # configure this individually in a profile, than globally. Typical users
  # are:
  #
  # Amazon Linux -> ec2-user
  # RHEL         -> ec2-user
  # CentOS       -> ec2-user
  # Ubuntu       -> ubuntu
  # Debian       -> admin
  #
  ssh_username: ec2-user

  # Optionally add an IAM profile
  iam_profile: 'arn:aws:iam::123456789012:instance-profile/ExampleInstanceProfile'

  driver: ec2


my-ec2-southeast-private-ips:
  # Set up the location of the salt master
  #
  minion:
    master: saltmaster.example.com

  # Specify whether to use public or private IP for deploy script.
  #
  # Valid options are:
  #     private_ips - The salt-master is also hosted with EC2
  #     public_ips - The salt-master is hosted outside of EC2
  #
  ssh_interface: private_ips

  # Optionally configure the Windows credential validation number of
  # retries and delay between retries.  This defaults to 10 retries
  # with a one second delay betwee retries
  win_deploy_auth_retries: 10
```

```
win_deploy_auth_retry_delay: 1

# Set the EC2 access credentials (see below)
#
id: 'use-instance-role-credentials'
key: 'use-instance-role-credentials'

# Make sure this key is owned by root with permissions 0400.
#
private_key: /etc/salt/my_test_key.pem
keyname: my_test_key

# This one should NOT be specified if VPC was not configured in AWS to be
# the default. It might cause an error message which says that network
# interfaces and an instance-level security groups may not be specified
# on the same request.
#
securitygroup: default

# Optionally configure default region
#
location: ap-southeast-1
availability_zone: ap-southeast-1b

# Configure which user to use to run the deploy script. This setting is
# dependent upon the AMI that is used to deploy. It is usually safer to
# configure this individually in a profile, than globally. Typical users
# are:
#
# Amazon Linux -> ec2-user
# RHEL         -> ec2-user
# CentOS       -> ec2-user
# Ubuntu       -> ubuntu
#
ssh_username: ec2-user

# Optionally add an IAM profile
iam_profile: 'my other profile name'

driver: ec2
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Access Credentials

The `id` and `key` settings may be found in the Security Credentials area of the AWS Account page:

https://portal.aws.amazon.com/gp/aws/securityCredentials

Both are located in the Access Credentials area of the page, under the Access Keys tab. The `id` setting is labeled Access Key ID, and the `key` setting is labeled Secret Access Key.

Note: if either `id` or `key` is set to `use-instance-role-credentials' it is assumed that Salt is running on an AWS instance, and the instance role credentials will be retrieved and used. Since both the `id` and `key` are required parameters for the AWS ec2 provider, it is recommended to set both to `use-instance-role-credentials' for this functionality.

A ``static'' and ``permanent'' Access Key ID and Secret Key can be specified, but this is not recommended. Instance role keys are rotated on a regular basis, and are the recommended method of specifying AWS credentials.

Windows Deploy Timeouts

For Windows instances, it may take longer than normal for the instance to be ready. In these circumstances, the provider configuration can be configured with a `win_deploy_auth_retries` and/or a `win_deploy_auth_retry_delay` setting, which default to 10 retries and a one second delay between retries. These retries and timeouts relate to validating the Administrator password once AWS provides the credentials via the AWS API.

Key Pairs

In order to create an instance with Salt installed and configured, a key pair will need to be created. This can be done in the EC2 Management Console, in the Key Pairs area. These key pairs are unique to a specific region. Keys in the us-east-1 region can be configured at:

https://console.aws.amazon.com/ec2/home?region=us-east-1#s=KeyPairs

Keys in the us-west-1 region can be configured at

https://console.aws.amazon.com/ec2/home?region=us-west-1#s=KeyPairs

...and so on. When creating a key pair, the browser will prompt to download a pem file. This file must be placed in a directory accessible by Salt Cloud, with permissions set to either 0400 or 0600.

Security Groups

An instance on EC2 needs to belong to a security group. Like key pairs, these are unique to a specific region. These are also configured in the EC2 Management Console. Security groups for the us-east-1 region can be configured at:

https://console.aws.amazon.com/ec2/home?region=us-east-1#s=SecurityGroups

...and so on.

A security group defines firewall rules which an instance will adhere to. If the salt-master is configured outside of EC2, the security group must open the SSH port (usually port 22) in order for Salt Cloud to install Salt.

IAM Profile

Amazon EC2 instances support the concept of an instance profile, which is a logical container for the IAM role. At the time that you launch an EC2 instance, you can associate the instance with an instance profile, which in turn corresponds to the IAM role. Any software that runs on the EC2 instance is able to access AWS using the permissions associated with the IAM role.

Scaffolding the profile is a 2-step configuration process:

1. Configure an IAM Role from the IAM Management Console.
2. Attach this role to a new profile. It can be done with the AWS CLI:

```
> aws iam create-instance-profile --instance-profile-name PROFILE_NAME
> aws iam add-role-to-instance-profile --instance-profile-name PROFILE_
↪NAME --role-name ROLE_NAME
```

Once the profile is created, you can use the **PROFILE_NAME** to configure your cloud profiles.

Cloud Profiles

Set up an initial profile at /etc/salt/cloud.profiles:

```
base_ec2_private:
  provider: my-ec2-southeast-private-ips
  image: ami-e565ba8c
  size: t2.micro
  ssh_username: ec2-user

base_ec2_public:
  provider: my-ec2-southeast-public-ips
  image: ami-e565ba8c
  size: t2.micro
  ssh_username: ec2-user

base_ec2_db:
  provider: my-ec2-southeast-public-ips
  image: ami-e565ba8c
  size: m1.xlarge
  ssh_username: ec2-user
  volumes:
    - { size: 10, device: /dev/sdf }
    - { size: 10, device: /dev/sdg, type: io1, iops: 1000 }
    - { size: 10, device: /dev/sdh, type: io1, iops: 1000 }
    - { size: 10, device: /dev/sdi, tags: {"Environment": "production"} }
  # optionally add tags to profile:
  tag: {'Environment': 'production', 'Role': 'database'}
  # force grains to sync after install
  sync_after_install: grains

base_ec2_vpc:
  provider: my-ec2-southeast-public-ips
  image: ami-a73264ce
  size: m1.xlarge
  ssh_username: ec2-user
  script:  /etc/salt/cloud.deploy.d/user_data.sh
  network_interfaces:
    - DeviceIndex: 0
      PrivateIpAddresses:
        - Primary: True
      #auto assign public ip (not EIP)
      AssociatePublicIpAddress: True
      SubnetId: subnet-813d4bbf
      SecurityGroupId:
        - sg-750af413
  del_root_vol_on_destroy: True
  del_all_vol_on_destroy: True
  volumes:
    - { size: 10, device: /dev/sdf }
    - { size: 10, device: /dev/sdg, type: io1, iops: 1000 }
```

```
    - { size: 10, device: /dev/sdh, type: io1, iops: 1000 }
  tag: {'Environment': 'production', 'Role': 'database'}
  sync_after_install: grains
```

The profile can now be realized with a salt command:

```
# salt-cloud -p base_ec2 ami.example.com
# salt-cloud -p base_ec2_public ami.example.com
# salt-cloud -p base_ec2_private ami.example.com
```

This will create an instance named `ami.example.com` in EC2. The minion that is installed on this instance will have an `id` of `ami.example.com`. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt 'ami.example.com' test.ping
```

Required Settings

The following settings are always required for EC2:

```
# Set the EC2 login data
my-ec2-config:
  id: HJGRYCILJLKJYG
  key: 'kdjgfsgm;woormgl/aserigjksjdhasdfgn'
  keyname: test
  securitygroup: quick-start
  private_key: /root/test.pem
  driver: ec2
```

Optional Settings

EC2 allows a userdata file to be passed to the instance to be created. This functionality was added to Salt in the 2015.5.0 release.

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/my-userdata-file
```

Note: From versions 2016.11.0 and 2016.11.3, this file was passed through the master's *renderer* to template it. However, this caused issues with non-YAML data, so templating is no longer performed by default. To template the userdata_file, add a `userdata_template` option to the cloud profile:

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/my-userdata-file
  userdata_template: jinja
```

If no `userdata_template` is set in the cloud profile, then the master configuration will be checked for a *userdata_template* value. If this is not set, then no templating will be performed on the userdata_file.

To disable templating in a cloud profile when a *userdata_template* has been set in the master configuration file, simply set `userdata_template` to `False` in the cloud profile:

```
my-ec2-config:
  # Pass userdata to the instance to be created
  userdata_file: /etc/salt/my-userdata-file
  userdata_template: False
```

EC2 allows a location to be set for servers to be deployed in. Availability zones exist inside regions, and may be added to increase specificity.

```
my-ec2-config:
  # Optionally configure default region
  location: ap-southeast-1
  availability_zone: ap-southeast-1b
```

EC2 instances can have a public or private IP, or both. When an instance is deployed, Salt Cloud needs to log into it via SSH to run the deploy script. By default, the public IP will be used for this. If the salt-cloud command is run from another EC2 instance, the private IP should be used.

```
my-ec2-config:
  # Specify whether to use public or private IP for deploy script
  # private_ips or public_ips
  ssh_interface: public_ips
```

Many EC2 instances do not allow remote access to the root user by default. Instead, another user must be used to run the deploy script using sudo. Some common usernames include ec2-user (for Amazon Linux), ubuntu (for Ubuntu instances), admin (official Debian) and bitnami (for images provided by Bitnami).

```
my-ec2-config:
  # Configure which user to use to run the deploy script
  ssh_username: ec2-user
```

Multiple usernames can be provided, in which case Salt Cloud will attempt to guess the correct username. This is mostly useful in the main configuration file:

```
my-ec2-config:
  ssh_username:
    - ec2-user
    - ubuntu
    - admin
    - bitnami
```

Multiple security groups can also be specified in the same fashion:

```
my-ec2-config:
  securitygroup:
    - default
    - extra
```

EC2 instances can be added to an AWS Placement Group by specifying the `placementgroup` option:

```
my-ec2-config:
  placementgroup: my-aws-placement-group
```

Your instances may optionally make use of EC2 Spot Instances. The following example will request that spot instances be used and your maximum bid will be $0.10. Keep in mind that different spot prices may be needed based on the current value of the various EC2 instance sizes. You can check current and past spot instance pricing via the EC2 API or AWS Console.

```
my-ec2-config:
  spot_config:
    spot_price: 0.10
```

By default, the spot instance type is set to `one-time', meaning it will be launched and, if it's ever terminated for whatever reason, it will not be recreated. If you would like your spot instances to be relaunched after a termination (by you or AWS), set the `type` to `persistent'.

NOTE: Spot instances are a great way to save a bit of money, but you do run the risk of losing your spot instances if the current price for the instance size goes above your maximum bid.

The following parameters may be set in the cloud configuration file to control various aspects of the spot instance launching:

- `wait_for_spot_timeout`: seconds to wait before giving up on spot instance launch (default=600)
- `wait_for_spot_interval`: seconds to wait in between polling requests to determine if a spot instance is available (default=30)
- `wait_for_spot_interval_multiplier`: a multiplier to add to the interval in between requests, which is useful if AWS is throttling your requests (default=1)
- `wait_for_spot_max_failures`: maximum number of failures before giving up on launching your spot instance (default=10)

If you find that you're being throttled by AWS while polling for spot instances, you can set the following in your core cloud configuration file that will double the polling interval after each request to AWS.

```
wait_for_spot_interval: 1
wait_for_spot_interval_multiplier: 2
```

See the AWS Spot Instances documentation for more information.

Block device mappings enable you to specify additional EBS volumes or instance store volumes when the instance is launched. This setting is also available on each cloud profile. Note that the number of instance stores varies by instance type. If more mappings are provided than are supported by the instance type, mappings will be created in the order provided and additional mappings will be ignored. Consult the AWS documentation for a listing of the available instance stores, and device names.

```
my-ec2-config:
  block_device_mappings:
    - DeviceName: /dev/sdb
      VirtualName: ephemeral0
    - DeviceName: /dev/sdc
      VirtualName: ephemeral1
```

You can also use block device mappings to change the size of the root device at the provisioning time. For example, assuming the root device is `/dev/sda', you can set its size to 100G by using the following configuration.

```
my-ec2-config:
  block_device_mappings:
    - DeviceName: /dev/sda
      Ebs.VolumeSize: 100
      Ebs.VolumeType: gp2
      Ebs.SnapshotId: dummy0
    - DeviceName: /dev/sdb
      # required for devices > 2TB
      Ebs.VolumeType: gp2
      Ebs.VolumeSize: 3001
```

Existing EBS volumes may also be attached (not created) to your instances or you can create new EBS volumes based on EBS snapshots. To simply attach an existing volume use the `volume_id` parameter.

```
device: /dev/xvdj
volume_id: vol-12345abcd
```

Or, to create a volume from an EBS snapshot, use the `snapshot` parameter.

```
device: /dev/xvdj
snapshot: snap-abcd12345
```

Note that `volume_id` will take precedence over the `snapshot` parameter.

Tags can be set once an instance has been launched.

```
my-ec2-config:
    tag:
        tag0: value
        tag1: value
```

Setting up a Master inside EC2

Salt Cloud can configure Salt Masters as well as Minions. Use the `make_master` setting to use this functionality.

```
my-ec2-config:
  # Optionally install a Salt Master in addition to the Salt Minion
  make_master: True
```

When creating a Salt Master inside EC2 with `make_master: True`, or when the Salt Master is already located and configured inside EC2, by default, minions connect to the master's public IP address during Salt Cloud's provisioning process. Depending on how your security groups are defined, the minions may or may not be able to communicate with the master. In order to use the master's private IP in EC2 instead of the public IP, set the `salt_interface` to `private_ips`.

```
my-ec2-config:
  # Optionally set the IP configuration to private_ips
  salt_interface: private_ips
```

Modify EC2 Tags

One of the features of EC2 is the ability to tag resources. In fact, under the hood, the names given to EC2 instances by salt-cloud are actually just stored as a tag called Name. Salt Cloud has the ability to manage these tags:

```
salt-cloud -a get_tags mymachine
salt-cloud -a set_tags mymachine tag1=somestuff tag2='Other stuff'
salt-cloud -a del_tags mymachine tag1,tag2,tag3
```

It is possible to manage tags on any resource in EC2 with a Resource ID, not just instances:

```
salt-cloud -f get_tags my_ec2 resource_id=af5467ba
salt-cloud -f set_tags my_ec2 resource_id=af5467ba tag1=somestuff
salt-cloud -f del_tags my_ec2 resource_id=af5467ba tags=tag1,tag2,tag3
```

Rename EC2 Instances

As mentioned above, EC2 instances are named via a tag. However, renaming an instance by renaming its tag will cause the salt keys to mismatch. A rename function exists which renames both the instance, and the salt keys.

```
salt-cloud -a rename mymachine newname=yourmachine
```

Rename on Destroy

When instances on EC2 are destroyed, there will be a lag between the time that the action is sent, and the time that Amazon cleans up the instance. During this time, the instance still retains a Name tag, which will cause a collision if the creation of an instance with the same name is attempted before the cleanup occurs. In order to avoid such collisions, Salt Cloud can be configured to rename instances when they are destroyed. The new name will look something like:

```
myinstance-DEL20f5b8ad4eb64ed88f2c428df80a1a0c
```

In order to enable this, add rename_on_destroy line to the main configuration file:

```
my-ec2-config:
  rename_on_destroy: True
```

Listing Images

Normally, images can be queried on a cloud provider by passing the `--list-images` argument to Salt Cloud. This still holds true for EC2:

```
salt-cloud --list-images my-ec2-config
```

However, the full list of images on EC2 is extremely large, and querying all of the available images may cause Salt Cloud to behave as if frozen. Therefore, the default behavior of this option may be modified, by adding an `owner` argument to the provider configuration:

```
owner: aws-marketplace
```

The possible values for this setting are `amazon`, `aws-marketplace`, `self`, `<AWS account ID>` or `all`. The default setting is `amazon`. Take note that `all` and `aws-marketplace` may cause Salt Cloud to appear as if it is freezing, as it tries to handle the large amount of data.

It is also possible to perform this query using different settings without modifying the configuration files. To do this, call the `avail_images` function directly:

```
salt-cloud -f avail_images my-ec2-config owner=aws-marketplace
```

EC2 Images

The following are lists of available AMI images, generally sorted by OS. These lists are on 3rd-party websites, are not managed by Salt Stack in any way. They are provided here as a reference for those who are interested, and contain no warranty (express or implied) from anyone affiliated with Salt Stack. Most of them have never been used, much less tested, by the Salt Stack team.

- Arch Linux
- FreeBSD

- Fedora
- CentOS
- Ubuntu
- Debian
- OmniOS
- All Images on Amazon

show_image

This is a function that describes an AMI on EC2. This will give insight as to the defaults that will be applied to an instance using a particular AMI.

```
$ salt-cloud -f show_image ec2 image=ami-fd20ad94
```

show_instance

This action is a thin wrapper around `--full-query`, which displays details on a single instance only. In an environment with several machines, this will save a user from having to sort through all instance data, just to examine a single instance.

```
$ salt-cloud -a show_instance myinstance
```

ebs_optimized

This argument enables switching of the EbsOptimized setting which default to `false'. Indicates whether the instance is optimized for EBS I/O. This optimization provides dedicated throughput to Amazon EBS and an optimized configuration stack to provide optimal Amazon EBS I/O performance. This optimization isn't available with all instance types. Additional usage charges apply when using an EBS-optimized instance.

This setting can be added to the profile or map file for an instance.

If set to True, this setting will enable an instance to be EbsOptimized

```
ebs_optimized: True
```

This can also be set as a cloud provider setting in the EC2 cloud configuration:

```
my-ec2-config:
  ebs_optimized: True
```

del_root_vol_on_destroy

This argument overrides the default DeleteOnTermination setting in the AMI for the EBS root volumes for an instance. Many AMIs contain `false' as a default, resulting in orphaned volumes in the EC2 account, which may unknowingly be charged to the account. This setting can be added to the profile or map file for an instance.

If set, this setting will apply to the root EBS volume

```
del_root_vol_on_destroy: True
```

This can also be set as a cloud provider setting in the EC2 cloud configuration:

```
my-ec2-config:
  del_root_vol_on_destroy: True
```

del_all_vols_on_destroy

This argument overrides the default DeleteOnTermination setting in the AMI for the not-root EBS volumes for an instance. Many AMIs contain `false' as a default, resulting in orphaned volumes in the EC2 account, which may unknowingly be charged to the account. This setting can be added to the profile or map file for an instance.

If set, this setting will apply to any (non-root) volumes that were created by salt-cloud using the `volumes' setting.

The volumes will not be deleted under the following conditions * If a volume is detached before terminating the instance * If a volume is created without this setting and attached to the instance

```
del_all_vols_on_destroy: True
```

This can also be set as a cloud provider setting in the EC2 cloud configuration:

```
my-ec2-config:
  del_all_vols_on_destroy: True
```

The setting for this may be changed on all volumes of an existing instance using one of the following commands:

```
salt-cloud -a delvol_on_destroy myinstance
salt-cloud -a keepvol_on_destroy myinstance
salt-cloud -a show_delvol_on_destroy myinstance
```

The setting for this may be changed on a volume on an existing instance using one of the following commands:

```
salt-cloud -a delvol_on_destroy myinstance device=/dev/sda1
salt-cloud -a delvol_on_destroy myinstance volume_id=vol-1a2b3c4d
salt-cloud -a keepvol_on_destroy myinstance device=/dev/sda1
salt-cloud -a keepvol_on_destroy myinstance volume_id=vol-1a2b3c4d
salt-cloud -a show_delvol_on_destroy myinstance device=/dev/sda1
salt-cloud -a show_delvol_on_destroy myinstance volume_id=vol-1a2b3c4d
```

EC2 Termination Protection

EC2 allows the user to enable and disable termination protection on a specific instance. An instance with this protection enabled cannot be destroyed. The EC2 driver adds a show_term_protect action to the regular EC2 functionality.

```
salt-cloud -a show_term_protect mymachine
salt-cloud -a enable_term_protect mymachine
salt-cloud -a disable_term_protect mymachine
```

Alternate Endpoint

Normally, EC2 endpoints are build using the region and the service_url. The resulting endpoint would follow this pattern:

```
ec2.<region>.<service_url>
```

This results in an endpoint that looks like:

```
ec2.us-east-1.amazonaws.com
```

There are other projects that support an EC2 compatibility layer, which this scheme does not account for. This can be overridden by specifying the endpoint directly in the main cloud configuration file:

```
my-ec2-config:
  endpoint: myendpoint.example.com:1138/services/Cloud
```

Volume Management

The EC2 driver has several functions and actions for management of EBS volumes.

Creating Volumes

A volume may be created, independent of an instance. A zone must be specified. A size or a snapshot may be specified (in GiB). If neither is given, a default size of 10 GiB will be used. If a snapshot is given, the size of the snapshot will be used.

The following parameters may also be set (when providing a snapshot OR size):

- `type`: choose between standard (magnetic disk), gp2 (SSD), or io1 (provisioned IOPS). (default=standard)
- `iops`: the number of IOPS (only applicable to io1 volumes) (default varies on volume size)
- `encrypted`: enable encryption on the volume (default=false)

```
salt-cloud -f create_volume ec2 zone=us-east-1b
salt-cloud -f create_volume ec2 zone=us-east-1b size=10
salt-cloud -f create_volume ec2 zone=us-east-1b snapshot=snap12345678
salt-cloud -f create_volume ec2 size=10 type=standard
salt-cloud -f create_volume ec2 size=10 type=gp2
salt-cloud -f create_volume ec2 size=10 type=io1 iops=1000
```

Attaching Volumes

Unattached volumes may be attached to an instance. The following values are required; name or instance_id, volume_id, and device.

```
salt-cloud -a attach_volume myinstance volume_id=vol-12345 device=/dev/sdb1
```

Show a Volume

The details about an existing volume may be retrieved.

```
salt-cloud -a show_volume myinstance volume_id=vol-12345
salt-cloud -f show_volume ec2 volume_id=vol-12345
```

Detaching Volumes

An existing volume may be detached from an instance.

```
salt-cloud -a detach_volume myinstance volume_id=vol-12345
```

Deleting Volumes

A volume that is not attached to an instance may be deleted.

```
salt-cloud -f delete_volume ec2 volume_id=vol-12345
```

Managing Key Pairs

The EC2 driver has the ability to manage key pairs.

Creating a Key Pair

A key pair is required in order to create an instance. When creating a key pair with this function, the return data will contain a copy of the private key. This private key is not stored by Amazon, will not be obtainable past this point, and should be stored immediately.

```
salt-cloud -f create_keypair ec2 keyname=mykeypair
```

Importing a Key Pair

```
salt-cloud -f import_keypair ec2 keyname=mykeypair file=/path/to/id_rsa.pub
```

Show a Key Pair

This function will show the details related to a key pair, not including the private key itself (which is not stored by Amazon).

```
salt-cloud -f show_keypair ec2 keyname=mykeypair
```

Delete a Key Pair

This function removes the key pair from Amazon.

```
salt-cloud -f delete_keypair ec2 keyname=mykeypair
```

Launching instances into a VPC

Simple launching into a VPC

In the amazon web interface, identify the id or the name of the subnet into which your image should be created. Then, edit your cloud.profiles file like so:-

```
profile-id:
  provider: provider-name
  subnetid: subnet-XXXXXXXX
  image: ami-XXXXXXXX
  size: m1.medium
  ssh_username: ubuntu
  securitygroupid:
    - sg-XXXXXXXX
  securitygroupname:
    - AnotherSecurityGroup
    - AndThirdSecurityGroup
```

Note that `subnetid' takes precedence over `subnetname', but `securitygroupid' and `securitygroupname' are merged together to generate a single list for SecurityGroups of instances.

Specifying interface properties

New in version 2014.7.0.

Launching into a VPC allows you to specify more complex configurations for the network interfaces of your virtual machines, for example:-

```
profile-id:
  provider: provider-name
  image: ami-XXXXXXXX
  size: m1.medium
  ssh_username: ubuntu

  # Do not include either 'subnetid', 'subnetname', 'securitygroupid' or
  # 'securitygroupname' here if you are going to manually specify
  # interface configuration
  #
  network_interfaces:
    - DeviceIndex: 0
      SubnetId: subnet-XXXXXXXX
      SecurityGroupId:
        - sg-XXXXXXXX

      # Uncomment this line if you would like to set an explicit private
      # IP address for the ec2 instance
      #
      # PrivateIpAddress: 192.168.1.66

      # Uncomment this to associate an existing Elastic IP Address with
      # this network interface:
      #
      # associate_eip: eipalloc-XXXXXXXX

      # You can allocate more than one IP address to an interface. Use the
```

```
        # 'ip addr list' command to see them.
        #
        # SecondaryPrivateIpAddressCount: 2

        # Uncomment this to allocate a new Elastic IP Address to this
        # interface (will be associated with the primary private ip address
        # of the interface
        #
        # allocate_new_eip: True

        # Uncomment this instead to allocate a new Elastic IP Address to
        # both the primary private ip address and each of the secondary ones
        #
        allocate_new_eips: True

        # Uncomment this if you're creating NAT instances. Allows an instance
        # to accept IP packets with destinations other than itself.
        # SourceDestCheck: False

      - DeviceIndex: 1
        subnetname: XXXXXXXX-Subnet
        securitygroupname:
          - XXXXXXXX-SecurityGroup
          - YYYYYYYY-SecurityGroup
```

Note that it is an error to assign a `subnetid', `subnetname', `securitygroupid' or `securitygroupname' to a profile where the interfaces are manually configured like this. These are both really properties of each network interface, not of the machine itself.

13.7.8 Getting Started With GoGrid

GoGrid is a public cloud host that supports Linux and Windows.

Configuration

To use Salt Cloud with GoGrid log into the GoGrid web interface and create an API key. Do this by clicking on ``My Account'' and then going to the API Keys tab.

The `apikey` and the `sharedsecret` configuration parameters need to be set in the configuration file to enable interfacing with GoGrid:

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-gogrid-config:
  driver: gogrid
  apikey: asdff7896asdh789
  sharedsecret: saltybacon
```

Note: A Note about using Map files with GoGrid:

Due to limitations in the GoGrid API, instances cannot be provisioned in parallel with the GoGrid driver. Map files will work with GoGrid, but the `-P` argument should not be used on maps referencing GoGrid instances.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory:

```
gogrid_512:
  provider: my-gogrid-config
  size: 512MB
  image: CentOS 6.2 (64-bit) w/ None
```

Sizes can be obtained using the `--list-sizes` option for the `salt-cloud` command:

```
# salt-cloud --list-sizes my-gogrid-config
my-gogrid-config:
    ----------
    gogrid:
        ----------
        512MB:
            ----------
            bandwidth:
                None
            disk:
                30
            driver:
            get_uuid:
            id:
                512MB
            name:
                512MB
            price:
                0.095
            ram:
                512
            uuid:
                bde1e4d7c3a643536e42a35142c7caac34b060e9
...SNIP...
```

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images my-gogrid-config
my-gogrid-config:
    ----------
    gogrid:
        ----------
        CentOS 6.4 (64-bit) w/ None:
            ----------
```

```
            driver:
            extra:
                ----------
            get_uuid:
            id:
                18094
            name:
                CentOS 6.4 (64-bit) w/ None
            uuid:
                bfd4055389919e01aa6261828a96cf54c8dcc2c4
...SNIP...
```

Assigning IPs

New in version 2015.8.0.

The GoGrid API allows IP addresses to be manually assigned. Salt Cloud supports this functionality by allowing an IP address to be specified using the `assign_public_ip` argument. This likely makes the most sense inside a map file, but it may also be used inside a profile.

```
gogrid_512:
  provider: my-gogrid-config
  size: 512MB
  image: CentOS 6.2 (64-bit) w/ None
  assign_public_ip: 11.38.257.42
```

13.7.9 Getting Started With Google Compute Engine

Google Compute Engine (GCE) is Google-infrastructure as a service that lets you run your large-scale computing workloads on virtual machines. This document covers how to use Salt Cloud to provision and manage your virtual machines hosted within Google's infrastructure.

You can find out more about GCE and other Google Cloud Platform services at https://cloud.google.com.

Dependencies

- LibCloud >= 1.0.0

Changed in version 2017.7.0.

- A Google Cloud Platform account with Compute Engine enabled
- A registered Service Account for authorization
- Oh, and obviously you'll need salt

Google Compute Engine Setup

1. Sign up for Google Cloud Platform

   Go to https://cloud.google.com and use your Google account to sign up for Google Cloud Platform and complete the guided instructions.

2. Create a Project

   Next, go to the console at https://cloud.google.com/console and create a new Project. Make sure to select your new Project if you are not automatically directed to the Project.

   Projects are a way of grouping together related users, services, and billing. You may opt to create multiple Projects and the remaining instructions will need to be completed for each Project if you wish to use GCE and Salt Cloud to manage your virtual machines.

3. Enable the Google Compute Engine service

   In your Project, either just click *Compute Engine* to the left, or go to the *APIs & auth* section and *APIs* link and enable the Google Compute Engine service.

4. Create a Service Account

   To set up authorization, navigate to *APIs & auth* section and then the *Credentials* link and click the *CREATE NEW CLIENT ID* button. Select *Service Account* and click the *Create Client ID* button. This will automatically download a `.json` file, which may or may not be used in later steps, depending on your version of `libcloud`.

   Look for a new *Service Account* section in the page and record the generated email address for the matching key/fingerprint. The email address will be used in the `service_account_email_address` of the `/etc/salt/cloud.providers` or the `/etc/salt/cloud.providers.d/*.conf` file.

5. Key Format

   Note: If you are using `libcloud >= 0.17.0` it is recommended that you use the `JSON format` file you downloaded above and skip to the *Provider Configuration* section below, using the JSON file **in place of `NEW.pem'** in the documentation.

   If you are using an older version of libcloud or are unsure of the version you have, please follow the instructions below to generate and format a new P12 key.

   In the new *Service Account* section, click *Generate new P12 key*, which will automatically download a `.p12` private key file. The `.p12` private key needs to be converted to a format compatible with libcloud. This new Google-generated private key was encrypted using *notasecret* as a passphrase. Use the following command and record the location of the converted private key and record the location for use in the `service_account_private_key` of the `/etc/salt/cloud` file:

```
openssl pkcs12 -in ORIG.p12 -passin pass:notasecret \
-nodes -nocerts | openssl rsa -out NEW.pem
```

Provider Configuration

Set up the provider cloud config at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/*.conf`:

```
gce-config:
  # Set up the Project name and Service Account authorization
  project: "your-project-id"
  service_account_email_address: "123-a5gt@developer.gserviceaccount.com"
  service_account_private_key: "/path/to/your/NEW.pem"

  # Set up the location of the salt master
  minion:
    master: saltmaster.example.com
```

```
# Set up grains information, which will be common for all nodes
# using this provider
grains:
  node_type: broker
  release: 1.0.1

driver: gce
```

Note: Empty strings as values for `service_account_private_key` and `service_account_email_address` can be used on GCE instances. This will result in the service account assigned to the GCE instance being used.

Note: The value provided for `project` must not contain underscores or spaces and is labeled as ``Project ID'' on the Google Developers Console.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profile Configuration

Set up an initial profile at `/etc/salt/cloud.profiles` or `/etc/salt/cloud.profiles.d/*.conf`:

```
my-gce-profile:
  image: centos-6
  size: n1-standard-1
  location: europe-west1-b
  network: default
  subnetwork: default
  tags: '["one", "two", "three"]'
  metadata: '{"one": "1", "2": "two"}'
  use_persistent_disk: True
  delete_boot_pd: False
  deploy: True
  make_master: False
  provider: gce-config
```

The profile can be realized now with a salt command:

```
salt-cloud -p my-gce-profile gce-instance
```

This will create an salt minion instance named `gce-instance` in GCE. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with a salt-minion installed, connectivity to it can be verified with Salt:

```
salt gce-instance test.ping
```

GCE Specific Settings

Consult the sample profile below for more information about GCE specific settings. Some of them are mandatory and are properly labeled below but typically also include a hard-coded default.

Initial Profile

Set up an initial profile at `/etc/salt/cloud.profiles` or `/etc/salt/cloud.profiles.d/gce.conf`:

```
my-gce-profile:
  image: centos-6
  size: n1-standard-1
  location: europe-west1-b
  network: default
  subnetwork: default
  tags: '["one", "two", "three"]'
  metadata: '{"one": "1", "2": "two"}'
  use_persistent_disk: True
  delete_boot_pd: False
  ssh_interface: public_ips
  external_ip: "ephemeral"
```

image

Image is used to define what Operating System image should be used to for the instance. Examples are Debian 7 (wheezy) and CentOS 6. Required.

size

A `size', in GCE terms, refers to the instance's `machine type'. See the on-line documentation for a complete list of GCE machine types. Required.

location

A `location', in GCE terms, refers to the instance's `zone'. GCE has the notion of both Regions (e.g. us-central1, europe-west1, etc) and Zones (e.g. us-central1-a, us-central1-b, etc). Required.

network

Use this setting to define the network resource for the instance. All GCE projects contain a network named `default' but it's possible to use this setting to create instances belonging to a different network resource.

subnetwork

Use this setting to define the subnetwork an instance will be created in. This requires that the network your instance is created under has a mode of `custom' or `auto'. Additionally, the subnetwork your instance is created under is associated with the location you provide.

New in version 2017.7.0.

tags

GCE supports instance/network tags and this setting allows you to set custom tags. It should be a list of strings and must be parse-able by the python ast.literal_eval() function to convert it to a python list.

metadata

GCE supports instance metadata and this setting allows you to set custom metadata. It should be a hash of key/value strings and parse-able by the python ast.literal_eval() function to convert it to a python dictionary.

use_persistent_disk

Use this setting to ensure that when new instances are created, they will use a persistent disk to preserve data between instance terminations and re-creations.

delete_boot_pd

In the event that you wish the boot persistent disk to be permanently deleted when you destroy an instance, set delete_boot_pd to True.

ssh_interface

New in version 2015.5.0.

Specify whether to use public or private IP for deploy script.

Valid options are:

- private_ips: The salt-master is also hosted with GCE
- public_ips: The salt-master is hosted outside of GCE

external_ip

Per instance setting: Used a named fixed IP address to this host.

Valid options are:

- ephemeral: The host will use a GCE ephemeral IP
- None: No external IP will be configured on this host.

Optionally, pass the name of a GCE address to use a fixed IP address. If the address does not already exist, it will be created.

ex_disk_type

GCE supports two different disk types, pd-standard and pd-ssd. The default disk type setting is pd-standard. To specify using an SSD disk, set pd-ssd as the value.

New in version 2014.7.0.

ip_forwarding

GCE instances can be enabled to use IP Forwarding. When set to True, this options allows the instance to send/receive non-matching src/dst packets. Default is False.

New in version 2015.8.1.

Profile with scopes

Scopes can be specified by setting the optional ex_service_accounts key in your cloud profile. The following example enables the bigquery scope.

```
my-gce-profile:
 image: centos-6
  ssh_username: salt
  size: f1-micro
  location: us-central1-a
  network: default
  subnetwork: default
  tags: '["one", "two", "three"]'
  metadata: '{"one": "1", "2": "two",
              "sshKeys": ""}'
  use_persistent_disk: True
  delete_boot_pd: False
  deploy: False
  make_master: False
  provider: gce-config
  ex_service_accounts:
    - scopes:
      - bigquery
```

Email can also be specified as an (optional) parameter.

```
my-gce-profile:
...snip
  ex_service_accounts:
    - scopes:
      - bigquery
      email: default
```

There can be multiple entries for scopes since ex-service_accounts accepts a list of dictionaries. For more information refer to the libcloud documentation on specifying service account scopes.

SSH Remote Access

GCE instances do not allow remote access to the root user by default. Instead, another user must be used to run the deploy script using sudo. Append something like this to /etc/salt/cloud.profiles or /etc/salt/cloud.profiles.d/*.conf:

```
my-gce-profile:
    ...

    # SSH to GCE instances as gceuser
    ssh_username: gceuser

    # Use the local private SSH key file located here
    ssh_keyfile: /etc/cloud/google_compute_engine
```

If you have not already used this SSH key to login to instances in this GCE project you will also need to add the public key to your projects metadata at https://cloud.google.com/console. You could also add it via the metadata setting too:

```
my-gce-profile:
    ...

    metadata: '{"one": "1", "2": "two",
                "sshKeys": "gceuser:ssh-rsa <Your SSH Public Key> gceuser@host"}'
```

Single instance details

This action is a thin wrapper around `--full-query`, which displays details on a single instance only. In an environment with several machines, this will save a user from having to sort through all instance data, just to examine a single instance.

```
salt-cloud -a show_instance myinstance
```

Destroy, persistent disks, and metadata

As noted in the provider configuration, it's possible to force the boot persistent disk to be deleted when you destroy the instance. The way that this has been implemented is to use the instance metadata to record the cloud profile used when creating the instance. When `destroy` is called, if the instance contains a `salt-cloud-profile` key, it's value is used to reference the matching profile to determine if `delete_boot_pd` is set to `True`.

Be aware that any GCE instances created with salt cloud will contain this custom `salt-cloud-profile` metadata entry.

List various resources

It's also possible to list several GCE resources similar to what can be done with other providers. The following commands can be used to list GCE zones (locations), machine types (sizes), and images.

```
salt-cloud --list-locations gce
salt-cloud --list-sizes gce
salt-cloud --list-images gce
```

Persistent Disk

The Compute Engine provider provides functions via salt-cloud to manage your Persistent Disks. You can create and destroy disks as well as attach and detach them from running instances.

Create

When creating a disk, you can create an empty disk and specify its size (in GB), or specify either an `image' or `snapshot'.

```
salt-cloud -f create_disk gce disk_name=pd location=us-central1-b size=200
```

Delete

Deleting a disk only requires the name of the disk to delete

```
salt-cloud -f delete_disk gce disk_name=old-backup
```

Attach

Attaching a disk to an existing instance is really an `action' and requires both an instance name and disk name. It's possible to use this ation to create bootable persistent disks if necessary. Compute Engine also supports attaching a persistent disk in READ_ONLY mode to multiple instances at the same time (but then cannot be attached in READ_WRITE to any instance).

```
salt-cloud -a attach_disk myinstance disk_name=pd mode=READ_WRITE boot=yes
```

Detach

Detaching a disk is also an action against an instance and only requires the name of the disk. Note that this does *not* safely sync and umount the disk from the instance. To ensure no data loss, you must first make sure the disk is unmounted from the instance.

```
salt-cloud -a detach_disk myinstance disk_name=pd
```

Show disk

It's also possible to look up the details for an existing disk with either a function or an action.

```
salt-cloud -a show_disk myinstance disk_name=pd
salt-cloud -f show_disk gce disk_name=pd
```

Create snapshot

You can take a snapshot of an existing disk's content. The snapshot can then in turn be used to create other persistent disks. Note that to prevent data corruption, it is strongly suggested that you unmount the disk prior to taking a snapshot. You must name the snapshot and provide the name of the disk.

```
salt-cloud -f create_snapshot gce name=backup-20140226 disk_name=pd
```

Delete snapshot

You can delete a snapshot when it's no longer needed by specifying the name of the snapshot.

```
salt-cloud -f delete_snapshot gce name=backup-20140226
```

Show snapshot

Use this function to look up information about the snapshot.

```
salt-cloud -f show_snapshot gce name=backup-20140226
```

Networking

Compute Engine supports multiple private networks per project. Instances within a private network can easily communicate with each other by an internal DNS service that resolves instance names. Instances within a private network can also communicate with either directly without needing special routing or firewall rules even if they span different regions/zones.

Networks also support custom firewall rules. By default, traffic between instances on the same private network is open to all ports and protocols. Inbound SSH traffic (port 22) is also allowed but all other inbound traffic is blocked.

Create network

New networks require a name and CIDR range if they don't have a `mode'. Optionally, `mode' can be provided. Supported modes are `auto', `custom', `legacy'. Optionally, `description' can be provided to add an extra note to your network. New instances can be created and added to this network by setting the network name during create. It is not possible to add/remove existing instances to a network.

```
salt-cloud -f create_network gce name=mynet cidr=10.10.10.0/24
salt-cloud -f create_network gce name=mynet mode=auto description=some optional info.
```

Changed in version 2017.7.0.

Destroy network

Destroy a network by specifying the name. If a resource is currently using the target network an exception will be raised.

```
salt-cloud -f delete_network gce name=mynet
```

Show network

Specify the network name to view information about the network.

```
salt-cloud -f show_network gce name=mynet
```

Create subnetwork

New subnetworks require a name, region, and CIDR range. Optionally, `description' can be provided to add an extra note to your subnetwork. New instances can be created and added to this subnetwork by setting the subnetwork name during create. It is not possible to add/remove existing instances to a subnetwork.

```
salt-cloud -f create_subnetwork gce name=mynet network=mynet region=us-central1
↪cidr=10.0.10.0/24
salt-cloud -f create_subnetwork gce name=mynet network=mynet region=us-central1
↪cidr=10.10.10.0/24 description=some info about my subnet.
```

New in version 2017.7.0.

Destroy subnetwork

Destroy a subnetwork by specifying the name and region. If a resource is currently using the target subnetwork an exception will be raised.

```
salt-cloud -f delete_subnetwork gce name=mynet region=us-central1
```

New in version 2017.7.0.

Show subnetwork

Specify the subnetwork name to view information about the subnetwork.

```
salt-cloud -f show_subnetwork gce name=mynet
```

New in version 2017.7.0.

Create address

Create a new named static IP address in a region.

```
salt-cloud -f create_address gce name=my-fixed-ip region=us-central1
```

Delete address

Delete an existing named fixed IP address.

```
salt-cloud -f delete_address gce name=my-fixed-ip region=us-central1
```

Show address

View details on a named address.

```
salt-cloud -f show_address gce name=my-fixed-ip region=us-central1
```

Create firewall

You'll need to create custom firewall rules if you want to allow other traffic than what is described above. For instance, if you run a web service on your instances, you'll need to explicitly allow HTTP and/or SSL traffic. The firewall rule must have a name and it will use the `default' network unless otherwise specified with a `network' attribute. Firewalls also support instance tags for source/destination

```
salt-cloud -f create_fwrule gce name=web allow=tcp:80,tcp:443,icmp
```

Delete firewall

Deleting a firewall rule will prevent any previously allowed traffic for the named firewall rule.

```
salt-cloud -f delete_fwrule gce name=web
```

Show firewall

Use this function to review an existing firewall rule's information.

```
salt-cloud -f show_fwrule gce name=web
```

Load Balancer

Compute Engine possess a load-balancer feature for splitting traffic across multiple instances. Please reference the documentation for a more complete description.

The load-balancer functionality is slightly different than that described in Google's documentation. The concept of *TargetPool* and *ForwardingRule* are consolidated in salt-cloud/libcloud. HTTP Health Checks are optional.

HTTP Health Check

HTTP Health Checks can be used as a means to toggle load-balancing across instance members, or to detect if an HTTP site is functioning. A common use-case is to set up a health check URL and if you want to toggle traffic on/off to an instance, you can temporarily have it return a non-200 response. A non-200 response to the load-balancer's health check will keep the LB from sending any new traffic to the ``down'' instance. Once the instance's health check URL beings returning 200-responses, the LB will again start to send traffic to it. Review Compute Engine's documentation for allowable parameters. You can use the following salt-cloud functions to manage your HTTP health checks.

```
salt-cloud -f create_hc gce name=myhc path=/ port=80
salt-cloud -f delete_hc gce name=myhc
salt-cloud -f show_hc gce name=myhc
```

Load-balancer

When creating a new load-balancer, it requires a name, region, port range, and list of members. There are other optional parameters for protocol, and list of health checks. Deleting or showing details about the LB only requires the name.

```
salt-cloud -f create_lb gce name=lb region=... ports=80 members=w1,w2,w3
salt-cloud -f delete_lb gce name=lb
salt-cloud -f show_lb gce name=lb
```

You can also create a load balancer using a named fixed IP addressby specifying the name of the address. If the address does not exist yet it will be created.

```
salt-cloud -f create_lb gce name=my-lb region=us-central1 ports=234 members=s1,s2,s3␣
↪address=my-lb-ip
```

Attach and Detach LB

It is possible to attach or detach an instance from an existing load-balancer. Both the instance and load-balancer must exist before using these functions.

```
salt-cloud -f attach_lb gce name=lb member=w4
salt-cloud -f detach_lb gce name=lb member=oops
```

13.7.10 Getting Started With HP Cloud

HP Cloud is a major public cloud platform and uses the libcloud *openstack* driver. The current version of OpenStack that HP Cloud uses is Havana. When an instance is booted, it must have a floating IP added to it in order to connect to it and further below you will see an example that adds context to this statement.

Set up a cloud provider configuration file

To use the *openstack* driver for HP Cloud, set up the cloud provider configuration file as in the example shown below:

`/etc/salt/cloud.providers.d/hpcloud.conf`:

```
hpcloud-config:
  # Set the location of the salt-master
  #
  minion:
    master: saltmaster.example.com

  # Configure HP Cloud using the OpenStack plugin
  #
  identity_url: https://region-b.geo-1.identity.hpcloudsvc.com:35357/v2.0/tokens
  compute_name: Compute
  protocol: ipv4

  # Set the compute region:
  #
  compute_region: region-b.geo-1

  # Configure HP Cloud authentication credentials
  #
  user: myname
  tenant: myname-project1
  password: xxxxxxxxx

  # keys to allow connection to the instance launched
```

```
  #
  ssh_key_name: yourkey
  ssh_key_file: /path/to/key/yourkey.priv

  driver: openstack
```

The subsequent example that follows is using the openstack driver.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Compute Region

Originally, HP Cloud, in its OpenStack Essex version (1.0), had 3 availability zones in one region, US West (region-a.geo-1), which each behaved each as a region.

This has since changed, and the current OpenStack Havana version of HP Cloud (1.1) now has simplified this and now has two regions to choose from:

```
region-a.geo-1 -> US West
region-b.geo-1 -> US East
```

Authentication

The `user` is the same user as is used to log into the HP Cloud management UI. The `tenant` can be found in the upper left under ``Project/Region/Scope''. It is often named the same as `user` albeit with a `-project1` appended. The `password` is of course what you created your account with. The management UI also has other information such as being able to select US East or US West.

Set up a cloud profile config file

The profile shown below is a know working profile for an Ubuntu instance. The profile configuration file is stored in the following location:

`/etc/salt/cloud.profiles.d/hp_ae1_ubuntu.conf`:

```
hp_ae1_ubuntu:
    provider: hp_ae1
    image: 9302692b-b787-4b52-a3a6-daebb79cb498
    ignore_cidr: 10.0.0.1/24
    networks:
      - floating: Ext-Net
    size: standard.small
    ssh_key_file: /root/keys/test.key
    ssh_key_name: test
    ssh_username: ubuntu
```

Some important things about the example above:

- The image parameter can use either the image name or image ID which you can obtain by running in the example below (this case US East):

```
# salt-cloud --list-images hp_ae1
```

- The parameter ignore_cidr specifies a range of addresses to ignore when trying to connect to the instance. In this case, it's the range of IP addresses used for an private IP of the instance.
- The parameter networks is very important to include. In previous versions of Salt Cloud, this is what made it possible for salt-cloud to be able to attach a floating IP to the instance in order to connect to the instance and set up the minion. The current version of salt-cloud doesn't require it, though having it is of no harm either. Newer versions of salt-cloud will use this, and without it, will attempt to find a list of floating IP addresses to use regardless.
- The ssh_key_file and ssh_key_name are the keys that will make it possible to connect to the instance to set up the minion
- The ssh_username parameter, in this case, being that the image used will be ubuntu, will make it possible to not only log in but install the minion

Launch an instance

To instantiate a machine based on this profile (example):

```
# salt-cloud -p hp_ae1_ubuntu ubuntu_instance_1
```

After several minutes, this will create an instance named ubuntu_instance_1 running in HP Cloud in the US East region and will set up the minion and then return information about the instance once completed.

Manage the instance

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt ubuntu_instance_1 ping
```

SSH to the instance

Additionally, the instance can be accessed via SSH using the floating IP assigned to it

```
# ssh ubuntu@<floating ip>
```

Using a private IP

Alternatively, in the cloud profile, using the private IP to log into the instance to set up the minion is another option, particularly if salt-cloud is running within the cloud on an instance that is on the same network with all the other instances (minions)

The example below is a modified version of the previous example. Note the use of ssh_interface:

```
hp_ae1_ubuntu:
    provider: hp_ae1
    image: 9302692b-b787-4b52-a3a6-daebb79cb498
    size: standard.small
    ssh_key_file: /root/keys/test.key
```

```
    ssh_key_name: test
    ssh_username: ubuntu
    ssh_interface: private_ips
```

With this setup, salt-cloud will use the private IP address to ssh into the instance and set up the salt-minion

13.7.11 Getting Started With Joyent

Joyent is a public cloud host that supports SmartOS, Linux, FreeBSD, and Windows.

Dependencies

This driver requires the Python `requests` library to be installed.

Configuration

The Joyent cloud requires three configuration parameters. The user name and password that are used to log into the Joyent system, and the location of the private ssh key associated with the Joyent account. The ssh key is needed to send the provisioning commands up to the freshly created virtual machine.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-joyent-config:
  driver: joyent
  user: fred
  password: saltybacon
  private_key: /root/mykey.pem
  keyname: mykey
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory:

```
joyent_512:
  provider: my-joyent-config
  size: g4-highcpu-512M
  image: ubuntu-16.04
```

Sizes can be obtained using the `--list-sizes` option for the `salt-cloud` command:

```
# salt-cloud --list-sizes my-joyent-config
my-joyent-config:
    ----------
    joyent:
        ----------
        g4-highcpu-512M:
            ----------
            default:
                False
            description:
                Compute Optimized 512M RAM - 1 vCPU - 10 GB Disk
            disk:
                10240
            group:
                Compute Optimized
            id:
                14aea8fc-d0f8-11e5-bfe4-a7458dbc6c99
            lwps:
                4000
            memory:
                512
            name:
                g4-highcpu-512M
            swap:
                2048
            vcpus:
                0
            version:
                1.0.3
...SNIP...
```

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images my-joyent-config
my-joyent-config:
    ----------
    joyent:
        ----------
        base:
            ----------
            description:
                A 32-bit SmartOS image with just essential packages
                installed. Ideal for users who are comfortabl e with
                setting up their own environment and tools.
            files:
                |_
                  ----------
                  compression:
                      gzip
                  sha1:
                      b00a77408ddd9aeac85085b68b1cd22a07353956
                  size:
                      106918297
            homepage:
                http://wiki.joyent.com/jpc2/Base+Instance
            id:
                00aec452-6e81-11e4-8474-ebfec9a1a911
            name:
```

```
            base
        os:
            smartos
        owner:
            9dce1460-0c4c-4417-ab8b-25ca478c5a78
        public:
            True
        published_at:
            2014-11-17T17:41:46Z
        requirements:
            ----------
        state:
            active
        type:
            smartmachine
        version:
            14.3.0

...SNIP...
```

SmartDataCenter

This driver can also be used with the Joyent SmartDataCenter project. More details can be found at:

Using SDC requires that an api_host_suffix is set. The default value for this is *.api.joyentcloud.com*. All characters, including the leading ., should be included:

```
api_host_suffix: .api.myhostname.com
```

Miscellaneous Configuration

The following configuration items can be set in either `provider` or `profile` confuration files.

use_ssl

When set to `True` (the default), attach `https://` to any URL that does not already have `http://` or `https://` included at the beginning. The best practice is to leave the protocol out of the URL, and use this setting to manage it.

verify_ssl

When set to `True` (the default), the underlying web library will verify the SSL certificate. This should only be set to `False` for debugging.`

13.7.12 Getting Started With Libvirt

Libvirt is a toolkit to interact with the virtualization capabilities of recent versions of Linux (and other OSes). This driver Salt cloud provider is currently geared towards libvirt with qemu-kvm.

http://www.libvirt.org/

Host Dependencies

- libvirt >= 1.2.18 (older might work)

Salt-Cloud Dependencies

- libvirt-python

Provider Configuration

For every KVM host a provider needs to be set up. The provider currently maps to one libvirt daemon (e.g. one KVM host).

Set up the provider cloud configuration file at /etc/salt/cloud.providers or /etc/salt/cloud.providers.d/*.conf.

```
# Set up a provider with qemu+ssh protocol
kvm-via-ssh:
  driver: libvirt
  url: qemu+ssh://user@kvm.company.com/system?socket=/var/run/libvirt/libvirt-sock

# Or connect to a local libvirt instance
local-kvm:
  driver: libvirt
  url: qemu:///system
  # work around flag for XML validation errors while cloning
  validate_xml: no
```

Cloud Profiles

Virtual machines get cloned from so called Cloud Profiles. Profiles can be set up at /etc/salt/cloud.profiles or /etc/salt/cloud.profiles.d/*.conf:

- Configure a profile to be used:

```
centos7:
  # points back at provider configuration
  provider: local-kvm
  base_domain: base-centos7-64
  ip_source: ip-learning
  ssh_username: root
  password: my-very-secret-password
  # /tmp is mounted noexec.. do workaround
  deploy_command: sh /tmp/.saltcloud/deploy.sh
  script_args: -F
  # grains to add to the minion
  grains:
    clones-are-awesome: true
  # override minion settings
  minion:
    master: 192.168.16.1
    master_port: 5506
```

The profile can be realized now with a salt command:

```
# salt-cloud -p centos7 my-centos7-clone
```

This will create an instance named `my-centos7-clone` on the cloud host. Also the minion id will be set to `my-centos7-clone`.

If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt my-centos7-clone test.ping
```

Required Settings

The following settings are always required for libvirt:

```
centos7:
  provider: local-kvm
  # the domain to clone
  base_domain: base-centos7-64
  # how to obtain the IP address of the cloned instance
  # ip-learning or qemu-agent
  ip_source: ip-learning
```

The `ip_source` setting controls how the IP address of the cloned instance is determined. When using `ip-learning` the IP is requested from libvirt. This needs a recent libvirt version and may only work for NAT networks. Another option is to use `qemu-agent` this requires that the qemu-agent is installed and configured to run at startup in the base domain.

Optional Settings

```
# Username and password
ssh_username: root
password: my-secret-password

# Cloning strategy: full or quick
clone_strategy: quick
```

The `clone_strategy` controls how the clone is done. In case of `full` the disks are copied creating a standalone clone. If `quick` is used the disks of the base domain are used as backing disks for the clone. This results in nearly instantaneous clones at the expense of slower write performance. The quick strategy has a number of requirements:

- The disks must be of type qcow2
- The base domain must be turned off
- The base domain must not change after creating the clone

13.7.13 Getting Started With Linode

Linode is a public cloud host with a focus on Linux instances.

Starting with the 2015.8.0 release of Salt, the Linode driver uses Linode's native REST API. There are no external dependencies required to use the Linode driver, other than a Linode account.

Provider Configuration

Linode requires a single API key, but the default root password for new instances also needs to be set. The password needs to be eight characters and contain lowercase, uppercase, and numbers.

Set up the provider cloud configuration file at /etc/salt/cloud.providers or /etc/salt/cloud.providers.d/*.conf.

```
my-linode-config:
  apikey: 'asldkgfakl;sdfjsjaslfjaklsdjf;askldjfaaklsjdfhasldsadfghdkf'
  password: 'F00barbaz'
  driver: linode
```

Note: Changed in version 2015.8.0.

The provider parameter in cloud provider definitions was renamed to driver. This change was made to avoid confusion with the provider parameter that is used in cloud profile definitions. Cloud provider definitions now use driver to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use provider to refer to provider configurations that you define.

Profile Configuration

Linode profiles require a provider, size, image, and location. Set up an initial profile at /etc/salt/cloud.profiles or in the /etc/salt/cloud.profiles.d/ directory:

```
linode_1024:
  provider: my-linode-config
  size: Linode 2GB
  image: CentOS 7
  location: London, England, UK
```

The profile can be realized now with a salt command:

```
salt-cloud -p linode_1024 linode-instance
```

This will create an salt minion instance named linode-instance in Linode. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with a salt-minion installed, connectivity to it can be verified with Salt:

```
salt linode-instance test.ping
```

Listing Sizes

Sizes can be obtained using the --list-sizes option for the salt-cloud command:

```
# salt-cloud --list-sizes my-linode-config
my-linode-config:
    ----------
    linode:
        ----------
        Linode 2GB:
            ----------
            AVAIL:
```

```
                ----------
                10:
                    500
                11:
                    500
                2:
                    500
                3:
                    500
                4:
                    500
                6:
                    500
                7:
                    500
                8:
                    500
                9:
                    500
            CORES:
                1
            DISK:
                50
            HOURLY:
                0.015
            LABEL:
                Linode 2GB
            PLANID:
                2
            PRICE:
                10.0
            RAM:
                2048
            XFER:
                2000
...SNIP...
```

Listing Images

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images my-linode-config
my-linode-config:
    ----------
    linode:
        ----------
        Arch Linux 2015.02:
            ----------
            CREATE_DT:
                2015-02-20 14:17:16.0
            DISTRIBUTIONID:
                138
            IS64BIT:
                1
            LABEL:
                Arch Linux 2015.02
```

```
            MINIMAGESIZE:
                800
            REQUIRESPVOPSKERNEL:
                1
...SNIP...
```

Listing Locations

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
# salt-cloud --list-locations my-linode-config
my-linode-config:
    ----------
    linode:
        ----------
        Atlanta, GA, USA:
            ----------
            ABBR:
                atlanta
            DATACENTERID:
                4
            LOCATION:
                Atlanta, GA, USA
...SNIP...
```

Linode Specific Settings

There are several options outlined below that can be added to either the Linode provider of profile configuration files. Some options are mandatory and are properly labeled below but typically also include a hard-coded default.

image

Image is used to define what Operating System image should be used for the instance. Examples are `Ubuntu 14.04 LTS` and `CentOS 7`. This option should be specified in the profile config. Required.

location

Location is used to define which Linode data center the instance will reside in. Required.

size

Size is used to define the instance's ``plan type'' which includes memory, storage, and price. Required.

assign_private_ip

New in version 2016.3.0.

Assigns a private IP address to a Linode when set to True. Default is False.

ssh_interface

New in version 2016.3.0.

Specify whether to use a public or private IP for the deploy script. Valid options are:

- public_ips: The salt-master is hosted outside of Linode. Default.
- private_ips: The salt-master is also hosted within Linode.

If specifying `private_ips`, the Linodes must be hosted within the same data center and have the Network Helper enabled on your entire account. The instance that is running the Salt-Cloud provisioning command must also have a private IP assigned to it.

Newer accounts created on Linode have the Network Helper setting enabled by default, account-wide. Legacy accounts do not have this setting enabled by default. To enable the Network Helper on your Linode account, please see Linode's Network Helper documentation.

If you're running into problems, be sure to restart the instance that is running Salt Cloud after adding its own private IP address or enabling the Network Helper.

clonefrom

Setting the clonefrom option to a specified instance enables the new instance to be cloned from the named instance instead of being created from scratch. If using the clonefrom option, it is likely a good idea to also specify `script_args: -C` if a minion is already installed on the to-be-cloned instance. See the *Cloning* section below for more information.

Cloning

To clone a Linode, add a profile with a `clonefrom` key, and a `script_args: -C`. `clonefrom` should be the name of the Linode that is the source for the clone. `script_args: -C` passes a -C to the salt-bootstrap script, which only configures the minion and doesn't try to install a new copy of salt-minion. This way the minion gets new keys and the keys get pre-seeded on the master, and the `/etc/salt/minion` file has the right minion `id:' declaration.

Cloning requires a post 2015-02-01 salt-bootstrap.

It is safest to clone a stopped machine. To stop a machine run

```
salt-cloud -a stop machine_to_clone
```

To create a new machine based on another machine, add an entry to your linode cloud profile that looks like this:

```
li-clone:
  provider: my-linode-config
  clonefrom: machine_to_clone
  script_args: -C -F
```

Then run salt-cloud as normal, specifying `-p li-clone`. The profile name can be anything; It doesn't have to be `li-clone`.

`clonefrom:` is the name of an existing machine in Linode from which to clone. `Script_args: -C -F` is necessary to avoid re-deploying Salt via salt-bootstrap. `-C` will just re-deploy keys so the new minion will not have a duplicate key or minion_id on the Master, and `-F` will force a rewrite of the Minion config file on the new Minion. If `-F` isn't provided, the new Minion will have the `machine_to_clone`s Minion ID, instead of its own Minion ID, which can cause problems.

Note: Pull Request #733 to the salt-bootstrap repo makes the `-F` argument non-necessary. Once that change is released into a stable version of the Bootstrap Script, the `-C` argument will be sufficient for the `script_args` setting.

If the `machine_to_clone` does not have Salt installed on it, refrain from using the `script_args: -C -F` altogether, because the new machine will need to have Salt installed.

13.7.14 Getting Started With LXC

The LXC module is designed to install Salt in an LXC container on a controlled and possibly remote minion.

In other words, Salt will connect to a minion, then from that minion:

- Provision and configure a container for networking access
- Use those modules to deploy salt and re-attach to master.
  - *lxc runner*
  - *lxc module*
  - *seed*

Limitations

- You can only act on one minion and one provider at a time.
- Listing images must be targeted to a particular LXC provider (nothing will be outputted with `all`)

Operation

Salt's LXC support does use *lxc.init* via the *lxc.cloud_init_interface* and seeds the minion via *seed.mkconfig*.

You can provide to those lxc VMs a profile and a network profile like if you were directly using the minion module.

Order of operation:

- Create the LXC container on the desired minion (clone or template)
- Change LXC config options (if any need to be changed)
- Start container
- Change base passwords if any
- Change base DNS configuration if necessary
- Wait for LXC container to be up and ready for ssh
- Test SSH connection and bailout in error
- Upload deploy script and seeds, then re-attach the minion.

Provider configuration

Here is a simple provider configuration:

```
# Note: This example goes in /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.
devhost10-lxc:
  target: devhost10
  driver: lxc
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profile configuration

Please read *LXC Management with Salt* before anything else. And specially *Profiles*.

Here are the options to configure your containers:

target Host minion id to install the lxc Container into

lxc_profile Name of the profile or inline options for the LXC vm creation/cloning, please see *Container Profiles*.

network_profile Name of the profile or inline options for the LXC vm network settings, please see *Network Profiles*.

nic_opts Totally optional. Per interface new-style configuration options mappings which will override any profile default option:

```
eth0: {'mac': '00:16:3e:01:29:40',
              'gateway': None, (default)
              'link': 'br0', (default)
              'gateway': None, (default)
              'netmask': '', (default)
              'ip': '22.1.4.25'}}
```

password password for root and sysadmin users

dnsservers List of DNS servers to use. This is optional.

minion minion configuration (see *Minion Configuration in Salt Cloud*)

bootstrap_delay specify the time to wait (in seconds) between container creation and salt bootstrap execution. It is useful to ensure that all essential services have started before the bootstrap script is executed. By default there's no wait time between container creation and bootstrap unless you are on systemd where we wait that the system is no more in starting state.

bootstrap_shell shell for bootstraping script (default: /bin/sh)

script defaults to salt-boostrap

script_args arguments which are given to the bootstrap script. the {0} placeholder will be replaced by the path which contains the minion config and key files, eg:

```
script_args="-c {0}"
```

Using profiles:

```
# Note: This example would go in /etc/salt/cloud.profiles or any file in the
# /etc/salt/cloud.profiles.d/ directory.
devhost10-lxc:
  provider: devhost10-lxc
  lxc_profile: foo
  network_profile: bar
  minion:
    master: 10.5.0.1
    master_port: 4506
```

Using inline profiles (eg to override the network bridge):

```
devhost11-lxc:
  provider: devhost10-lxc
  lxc_profile:
    clone_from: foo
  network_profile:
    etho:
      link: lxcbr0
  minion:
    master: 10.5.0.1
    master_port: 4506
```

Using a lxc template instead of a clone:

```
devhost11-lxc:
  provider: devhost10-lxc
  lxc_profile:
    template: ubuntu
    # options:
    #   release: trusty
  network_profile:
    etho:
      link: lxcbr0
  minion:
    master: 10.5.0.1
    master_port: 4506
```

Static ip:

```
# Note: This example would go in /etc/salt/cloud.profiles or any file in the
# /etc/salt/cloud.profiles.d/ directory.
devhost10-lxc:
  provider: devhost10-lxc
  nic_opts:
    eth0:
      ipv4: 10.0.3.9
  minion:
    master: 10.5.0.1
    master_port: 4506
```

DHCP:

```
# Note: This example would go in /etc/salt/cloud.profiles or any file in the
# /etc/salt/cloud.profiles.d/ directory.
devhost10-lxc:
  provider: devhost10-lxc
  minion:
    master: 10.5.0.1
    master_port: 4506
```

Driver Support

- Container creation
- Image listing (LXC templates)
- Running container information (IP addresses, etc.)

13.7.15 Getting Started With 1and1

1&1 is one of the world's leading Web hosting providers. 1&1 currently offers a wide range of Web hosting products, including email solutions and high-end servers in 10 different countries including Germany, Spain, Great Britain and the United States. From domains to 1&1 MyWebsite to eBusiness solutions like Cloud Hosting and Web servers for complex tasks, 1&1 is well placed to deliver a high quality service to its customers. All 1&1 products are hosted in 1&1's high-performance, green data centers in the USA and Europe.

Dependencies

- 1and1 >= 1.2.0

Configuration

- Using the new format, set up the cloud configuration at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/oneandone.conf`:

```
my-oneandone-config:
  driver: oneandone

  # Set the location of the salt-master
  #
  minion:
    master: saltmaster.example.com

  # Configure oneandone authentication credentials
  #
  api_token: <api_token>
  ssh_private_key: /path/to/id_rsa
  ssh_public_key: /path/to/id_rsa.pub
```

Authentication

The `api_key` is used for API authorization. This token can be obtained from the CloudPanel in the Management section below Users.

Profiles

Here is an example of a profile:

```
oneandone_fixed_size:
  provider: my-oneandone-config
  description: Small instance size server
  fixed_instance_size: S
  appliance_id: 8E3BAA98E3DFD37857810E0288DD8FBA

oneandone_custom_size:
  provider: my-oneandone-config
  description: Custom size server
  vcore: 2
  cores_per_processor: 2
  ram: 8
  appliance_id: 8E3BAA98E3DFD37857810E0288DD8FBA
  hdds:
  -
    is_main: true
    size: 20
  -
    is_main: false
    size: 20
```

The following list explains some of the important properties.

fixed_instance_size_id When creating a server, either `fixed_instance_size_id` or custom hardware params containing `vcore`, `cores_per_processor`, `ram`, and `hdds` must be provided. Can be one of the IDs listed among the output of the following command:

```
salt-cloud --list-sizes oneandone
```

vcore Total amount of processors.

cores_per_processor Number of cores per processor.

ram RAM memory size in GB.

hdds Hard disks.

appliance_id ID of the image that will be installed on server. Can be one of the IDs listed in the output of the following command:

```
salt-cloud --list-images oneandone
```

datacenter_id ID of the datacenter where the server will be created. Can be one of the IDs listed in the output of the following command:

```
salt-cloud --list-locations oneandone
```

description Description of the server.

password Password of the server. Password must contain more than 8 characters using uppercase letters, numbers and other special symbols.

power_on Power on server after creation. Default is set to true.

firewall_policy_id Firewall policy ID. If it is not provided, the server will assign the best firewall policy, creating a new one if necessary. If the parameter is sent with a 0 value, the server will be created with all ports blocked.

ip_id IP address ID.

load_balancer_id Load balancer ID.

monitoring_policy_id Monitoring policy ID.

deploy Set to False if Salt should not be installed on the node.

wait_for_timeout The timeout to wait in seconds for provisioning resources such as servers. The default wait_for_timeout is 15 minutes.

For more information concerning cloud profiles, see *here*.

13.7.16 Getting Started with OpenNebula

OpenNebula is an open-source solution for the comprehensive management of virtualized data centers to enable the mixed use of private, public, and hybrid IaaS clouds.

Dependencies

The driver requires Python's `lxml` library to be installed. It also requires an OpenNebula installation running version `4.12` or greater.

Configuration

The following example illustrates some of the options that can be set. These parameters are discussed in more detail below.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-opennebula-provider:
  # Set up the location of the salt master
  #
  minion:
    master: saltmaster.example.com

  # Define xml_rpc setting which Salt-Cloud uses to connect to the OpenNebula API.
↪Required.
  #
  xml_rpc: http://localhost:2633/RPC2

  # Define the OpenNebula access credentials. This can be the main "oneadmin" user
↪that OpenNebula uses as the
  # OpenNebula main admin, or it can be a user defined in the OpenNebula instance.
↪Required.
  #
  user: oneadmin
  password: JHGhgsayu32jsa

  # Define the private key location that is used by OpenNebula to access new VMs. This
↪setting is required if
  # provisioning new VMs or accessing VMs previously created with the associated
↪public key.
  #
  private_key: /path/to/private/key
```

```
  driver: opennebula
```

Access Credentials

The Salt Cloud driver for OpenNebula was written using OpenNebula's native XML RPC API. Every interaction with OpenNebula's API requires a `username` and `password` to make the connection from the machine running Salt Cloud to API running on the OpenNebula instance. Based on the access credentials passed in, OpenNebula filters the commands that the user can perform or the information for which the user can query. For example, the images that a user can view with a `--list-images` command are the images that the connected user and the connected user's groups can access.

Key Pairs

Salt Cloud needs to be able to access a virtual machine in order to install the Salt Minion by using a public/private key pair. The virtual machine will need to be seeded with the public key, which is laid down by the OpenNebula template. Salt Cloud then uses the corresponding private key, provided by the `private_key` setting in the cloud provider file, to SSH into the new virtual machine.

To seed the virtual machine with the public key, the public key must be added to the OpenNebula template. If using the OpenNebula web interface, navigate to the template, then click `Update`. Click the `Context` tab. Under the `Network & SSH` section, click `Add SSH Contextualization` and paste the public key in the `Public Key` box. Don't forget to save your changes by clicking the green `Update` button.

Note: The key pair must not have a pass-phrase.

Cloud Profiles

Set up an initial profile at either `/etc/salt/cloud.profiles` or the `/etc/salt/cloud.profiles.d/` directory.

```
my-opennebula-profile:
  provider: my-opennebula-provider
  image: Ubuntu-14.04
```

The profile can now be realized with a salt command:

```
salt-cloud -p my-opennebula-profile my-new-vm
```

This will create a new instance named `my-new-vm` in OpenNebula. The minion that is installed on this instance will have a minion id of `my-new-vm`. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
salt my-new-vm test.ping
```

OpenNebula uses an image --> template --> virtual machine paradigm where the template draws on the image, or disk, and virtual machines are created from templates. Because of this, there is no need to define a `size` in the cloud profile. The size of the virtual machine is defined in the template.

Change Disk Size

You can now change the size of a VM on creation by cloning an image and expanding the size. You can accomplish this by the following cloud profile settings below.

```
my-opennebula-profile:
  provider: my-opennebula-provider
  image: Ubuntu-14.04
  disk:
    disk0:
      disk_type: clone
      size: 8096
      image: centos7-base-image-v2
    disk1:
      disk_type: volatile
      type: swap
      size: 4096
    disk2:
      disk_type: volatile
      size: 4096
      type: fs
      format: ext3
```

There are currently two different disk_types a user can use: volatile and clone. Clone which is required when specifying devices will clone an image in open nebula and will expand it to the size specified in the profile settings. By default this will clone the image attached to the template specified in the profile but a user can add the *image* argument under the disk definition.

For example the profile below will not use Ubuntu-14.04 for the cloned disk image. It will use the centos7-base-image image:

```
my-opennebula-profile:
  provider: my-opennebula-provider
  image: Ubuntu-14.04
  disk:
    disk0:
      disk_type: clone
      size: 8096
      image: centos7-base-image
```

If you want to use the image attached to the template set in the profile you can simply remove the image argument as show below. The profile below will clone the image Ubuntu-14.04 and expand the disk to 8GB.:

```
my-opennebula-profile:
  provider: my-opennebula-provider
  image: Ubuntu-14.04
  disk:
    disk0:
      disk_type: clone
      size: 8096
```

A user can also currently specify swap or fs disks. Below is an example of this profile setting:

```
my-opennebula-profile:
  provider: my-opennebula-provider
  image: Ubuntu-14.04
  disk:
    disk0:
```

```
      disk_type: clone
      size: 8096
    disk1:
      disk_type: volatile
      type: swap
      size: 4096
    disk2:
      disk_type: volatile
      size: 4096
      type: fs
      format: ext3
```

The example above will attach both a swap disk and a ext3 filesystem with a size of 4GB. To note if you define other disks you have to define the image disk to clone because the template will write over the entire `DISK=[]' template definition on creation.

Required Settings

The following settings are always required for OpenNebula:

```
my-opennebula-config:
  xml_rpc: http://localhost:26633/RPC2
  user: oneadmin
  password: JHGhgsayu32jsa
  driver: opennebula
```

Required Settings for VM Deployment

The settings defined in the *Required Settings* section are required for all interactions with OpenNebula. However, when deploying a virtual machine via Salt Cloud, an additional setting, `private_key`, is also required:

```
my-opennebula-config:
  private_key: /path/to/private/key
```

Listing Images

Images can be queried on OpenNebula by passing the `--list-images` argument to Salt Cloud:

```
salt-cloud --list-images opennebula
```

Listing Locations

In OpenNebula, locations are defined as `hosts`. Locations, or ``hosts'', can be querried on OpenNebula by passing the `--list-locations` argument to Salt Cloud:

```
salt-cloud --list-locations opennebula
```

Listing Sizes

Sizes are defined by templates in OpenNebula. As such, the `--list-sizes` call returns an empty dictionary since there are no sizes to return.

Additional OpenNebula API Functionality

The Salt Cloud driver for OpenNebula was written using OpenNebula's native XML RPC API. As such, many `--function` and `--action` calls were added to the OpenNebula driver to enhance support for an OpenNebula infrastructure with additional control from Salt Cloud. See the *OpenNebula function definitions* for more information.

Access via DNS entry instead of IP

Some OpenNebula installations do not assign IP addresses to new VMs, instead they establish the new VM's hostname based on OpenNebula's name of the VM, and then allocate an IP out of DHCP with dynamic DNS attaching the hostname. This driver supports this behavior by adding the entry *fqdn_base* to the driver configuration or the OpenNebula profile with a value matching the base fully-qualified domain. For example:

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-opennebula-provider:
  [...]
  fqdn_base: corp.example.com
  [...]
```

13.7.17 Getting Started with Openstack

Openstack Cloud Driver

depends shade

OpenStack is an open source project that is in use by a number a cloud providers, each of which have their own ways of using it.

This OpenStack driver uses a the shade python module which is managed by the OpenStack Infra team. This module is written to handle all the different versions of different OpenStack tools for salt, so most commands are just passed over to the module to handle everything.

Provider

There are two ways to configure providers for this driver. The first one is to just let shade handle everything, and configure using os-client-config and setting up */etc/openstack/clouds.yml*.

```
clouds:
  democloud:
    region_name: RegionOne
    auth:
      username: 'demo'
      password: secret
      project_name: 'demo'
      auth_url: 'http://openstack/identity'
```

And then this can be referenced in the salt provider based on the *democloud* name.

```
myopenstack:
  driver: openstack
```

```
  cloud: democloud
  region_name: RegionOne
```

This allows for just using one configuration for salt-cloud and for any other openstack tools which are all using */etc/openstack/clouds.yml*

The other method allows for specifying everything in the provider config, instead of using the extra configuration file. This will allow for passing salt-cloud configs only through pillars for minions without having to write a clouds.yml file on each minion.abs

```
myopenstack:
  driver: openstack
  region_name: RegionOne
  auth:
    username: 'demo'
    password: secret
    project_name: 'demo'
    auth_url: 'http://openstack/identity'
```

Or if you need to use a profile to setup some extra stuff, it can be passed as a *profile* to use any of the vendor config options.

```
myrackspace:
  driver: openstack
  profile: rackspace
  auth:
    username: rackusername
    api_key: myapikey
  region_name: ORD
  auth_type: rackspace_apikey
```

And this will pull in the profile for rackspace and setup all the correct options for the auth_url and different api versions for services.

Profile

Most of the options for building servers are just passed on to the create_server function from shade.

The salt specific ones are:

- ssh_key_file: The path to the ssh key that should be used to login to the machine to bootstrap it
- ssh_key_file: The name of the keypair in openstack
- userdata_template: The renderer to use if the userdata is a file that is templated. Default: False
- ssh_interface: The interface to use to login for bootstrapping: public_ips, private_ips, floating_ips, fixed_ips

```
centos:
  provider: myopenstack
  image: CentOS 7
  size: ds1G
  ssh_key_name: mykey
  ssh_key_file: /root/.ssh/id_rsa
```

This is the minimum setup required.

If metadata is set to make sure that the host has finished setting up the *wait_for_metadata* can be set.

```
centos:
  provider: myopenstack
  image: CentOS 7
  size: ds1G
  ssh_key_name: mykey
  ssh_key_file: /root/.ssh/id_rsa
  meta:
    build_config: rack_user_only
  wait_for_metadata:
    rax_service_level_automation: Complete
    rackconnect_automation_status: DEPLOYED
```

Anything else from the create_server docs can be passed through here.

- **image: Image dict, name or ID to boot with. image is required** unless boot_volume is given.
- **flavor**: Flavor dict, name or ID to boot onto.
- **auto_ip: Whether to take actions to find a routable IP for** the server. (defaults to True)
- **ips**: List of IPs to attach to the server (defaults to None)
- **ip_pool: Name of the network or floating IP pool to get an** address from. (defaults to None)
- **root_volume: Name or ID of a volume to boot from** (defaults to None - deprecated, use boot_volume)
- **boot_volume: Name or ID of a volume to boot from** (defaults to None)
- **terminate_volume: If booting from a volume, whether it should** be deleted when the server is destroyed. (defaults to False)
- **volumes**: (optional) A list of volumes to attach to the server
- **meta: (optional) A dict of arbitrary key/value metadata to** store for this server. Both keys and values must be <=255 characters.
- **files: (optional, deprecated) A dict of files to overwrite** on the server upon boot. Keys are file names (i.e. `/etc/passwd`) and values are the file contents (either as a string or as a file-like object). A maximum of five entries is allowed, and each file must be 10k or less.
- **reservation_id**: a UUID for the set of servers being requested.
- **min_count: (optional extension) The minimum number of** servers to launch.
- **max_count: (optional extension) The maximum number of** servers to launch.
- **security_groups**: A list of security group names
- **userdata: user data to pass to be exposed by the metadata** server this can be a file type object as well or a string.
- **key_name: (optional extension) name of previously created** keypair to inject into the instance.
- **availability_zone: Name of the availability zone for instance** placement.
- **block_device_mapping: (optional) A dict of block** device mappings for this server.
- **block_device_mapping_v2: (optional) A dict of block** device mappings for this server.
- **nics: (optional extension) an ordered list of nics to be** added to this server, with information about connected networks, fixed IPs, port etc.
- **scheduler_hints: (optional extension) arbitrary key-value pairs** specified by the client to help boot an instance
- **config_drive: (optional extension) value for config drive** either boolean, or volume-id

- **disk_config: (optional extension) control how the disk is** partitioned when the server is created. possible values are `AUTO' or `MANUAL'.
- **admin_pass: (optional extension) add a user supplied admin** password.
- **timeout: (optional) Seconds to wait, defaults to 60.** See the `wait` parameter.
- **reuse_ips: (optional) Whether to attempt to reuse pre-existing** floating ips should a floating IP be needed (defaults to True)
- **network: (optional) Network dict or name or ID to attach the** server to. Mutually exclusive with the nics parameter. Can also be be a list of network names or IDs or network dicts.
- **boot_from_volume: Whether to boot from volume. `boot_volume'** implies True, but boot_from_volume=True with no boot_volume is valid and will create a volume from the image and use that.
- **volume_size: When booting an image from volume, how big should** the created volume be? Defaults to 50.
- **nat_destination: Which network should a created floating IP** be attached to, if it's not possible to infer from the cloud's configuration. (Optional, defaults to None)
- **group: ServerGroup dict, name or id to boot the server in.** If a group is provided in both scheduler_hints and in the group param, the group param will win. (Optional, defaults to None)

Note: If there is anything added, that is not in this list, it can be added to an *extras* dictionary for the profile, and that will be to the create_server function.

13.7.18 Getting Started With Parallels

Parallels Cloud Server is a product by Parallels that delivers a cloud hosting solution. The PARALLELS module for Salt Cloud enables you to manage instances hosted using PCS. Further information can be found at:

http://www.parallels.com/products/pcs/

- Using the old format, set up the cloud configuration at `/etc/salt/cloud`:

```
# Set up the location of the salt master
#
minion:
    master: saltmaster.example.com

# Set the PARALLELS access credentials (see below)
#
PARALLELS.user: myuser
PARALLELS.password: badpass

# Set the access URL for your PARALLELS host
#
PARALLELS.url: https://api.cloud.xmission.com:4465/paci/v1.0/
```

- Using the new format, set up the cloud configuration at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/parallels.conf`:

```
my-parallels-config:
  # Set up the location of the salt master
  #
  minion:
```

```
    master: saltmaster.example.com

  # Set the PARALLELS access credentials (see below)
  #
  user: myuser
  password: badpass

  # Set the access URL for your PARALLELS provider
  #
  url: https://api.cloud.xmission.com:4465/paci/v1.0/
  driver: parallels
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Access Credentials

The `user`, `password`, and `url` will be provided to you by your cloud host. These are all required in order for the PARALLELS driver to work.

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or `/etc/salt/cloud.profiles.d/parallels.conf`:

```
parallels-ubuntu:
    provider: my-parallels-config
    image: ubuntu-12.04-x86_64
```

The profile can be realized now with a salt command:

```
# salt-cloud -p parallels-ubuntu myubuntu
```

This will create an instance named `myubuntu` on the cloud host. The minion that is installed on this instance will have an `id` of `myubuntu`. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt myubuntu test.ping
```

Required Settings

The following settings are always required for PARALLELS:

- Using the old cloud configuration format:

```
PARALLELS.user: myuser
PARALLELS.password: badpass
PARALLELS.url: https://api.cloud.xmission.com:4465/paci/v1.0/
```

- Using the new cloud configuration format:

```
my-parallels-config:
  user: myuser
  password: badpass
  url: https://api.cloud.xmission.com:4465/paci/v1.0/
  driver: parallels
```

Optional Settings

Unlike other cloud providers in Salt Cloud, Parallels does not utilize a `size` setting. This is because Parallels allows the end-user to specify a more detailed configuration for their instances than is allowed by many other cloud hosts. The following options are available to be used in a profile, with their default settings listed.

```
# Description of the instance. Defaults to the instance name.
desc: <instance_name>

# How many CPU cores, and how fast they are (in MHz)
cpu_number: 1
cpu_power: 1000

# How many megabytes of RAM
ram: 256

# Bandwidth available, in kbps
bandwidth: 100

# How many public IPs will be assigned to this instance
ip_num: 1

# Size of the instance disk (in GiB)
disk_size: 10

# Username and password
ssh_username: root
password: <value from PARALLELS.password>

# The name of the image, from ``salt-cloud --list-images parallels``
image: ubuntu-12.04-x86_64
```

13.7.19 Getting Started With ProfitBricks

ProfitBricks provides an enterprise-grade Infrastructure as a Service (IaaS) solution that can be managed through a browser-based ``Data Center Designer'' (DCD) tool or via an easy to use API. A unique feature of the ProfitBricks platform is that it allows you to define your own settings for cores, memory, and disk size without being tied to a particular server size.

Dependencies

- profitbricks >= 3.0.0

Configuration

- Using the new format, set up the cloud configuration at /etc/salt/cloud.providers or /etc/salt/cloud.providers.d/profitbricks.conf:

```
my-profitbricks-config:
  driver: profitbricks

  # Set the location of the salt-master
  #
  minion:
    master: saltmaster.example.com

  # Configure ProfitBricks authentication credentials
  #
  username: user@domain.com
  password: 123456
  # datacenter_id is the UUID of a pre-existing virtual data center.
  datacenter_id: 9e6709a0-6bf9-4bd6-8692-60349c70ce0e
  # Connect to public LAN ID 1.
  public_lan: 1
  ssh_public_key: /path/to/id_rsa.pub
  ssh_private_key: /path/to/id_rsa
```

Note: Changed in version 2015.8.0.

The provider parameter in cloud provider definitions was renamed to driver. This change was made to avoid confusion with the provider parameter that is used in cloud profile definitions. Cloud provider definitions now use driver to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use provider to refer to provider configurations that you define.

Virtual Data Center

ProfitBricks uses the concept of Virtual Data Centers. These are logically separated from one another and allow you to have a self-contained environment for all servers, volumes, networking, snapshots, and so forth.

A list of existing virtual data centers can be retrieved with the following command:

```
salt-cloud -f list_datacenters my-profitbricks-config
```

Authentication

The username and password are the same as those used to log into the ProfitBricks ``Data Center Designer''.

Profiles

Here is an example of a profile:

```
profitbricks_staging
  provider: my-profitbricks-config
  size: Micro Instance
  image: 2f98b678-6e7e-11e5-b680-52540066fee9
  cores: 2
```

```
  ram: 4096
  public_lan: 1
  private_lan: 2
  ssh_public_key: /path/to/id_rsa.pub
  ssh_private_key: /path/to/id_rsa
  ssh_interface: private_lan

profitbricks_production:
  provider: my-profitbricks-config
  image: Ubuntu-15.10-server-2016-05-01
  disk_type: SSD
  disk_size: 40
  cores: 8
  cpu_family: INTEL_XEON
  ram: 32768
  public_lan: 1
  private_lan: 2
  public_firewall_rules:
    Allow SSH:
      protocol: TCP
      source_ip: 1.2.3.4
      port_range_start: 22
      port_range_end: 22
    Allow Ping:
      protocol: ICMP
      icmp_type: 8
  ssh_public_key: /path/to/id_rsa.pub
  ssh_private_key: /path/to/id_rsa
  ssh_interface: private_lan
  volumes:
    db_data:
      disk_size: 500
    db_log:
      disk_size: 50
      disk_type: HDD
      disk_availability_zone: ZONE_3
```

The following list explains some of the important properties.

size Can be one of the options listed in the output of the following command:

```
salt-cloud --list-sizes my-profitbricks
```

image Can be one of the options listed in the output of the following command:

```
salt-cloud --list-images my-profitbricks
```

disk_size This option allows you to override the size of the disk as defined by the size. The disk size is set in gigabytes (GB).

disk_type This option allow the disk type to be set to HDD or SSD. The default is HDD.

disk_availability_zone This option will provision the volume in the specified availability_zone.

cores This option allows you to override the number of CPU cores as defined by the size.

ram This option allows you to override the amount of RAM defined by the size. The value must be a multiple of 256, e.g. 256, 512, 768, 1024, and so forth.

availability_zone This options specifies in which availability zone the server should be built. Zones include ZONE_1 and ZONE_2. The default is AUTO.

public_lan This option will connect the server to the specified public LAN. If no LAN exists, then a new public LAN will be created. The value accepts a LAN ID (integer).

public_firewall_rules This option allows for a list of firewall rules assigned to the public network interface.

> **Firewall Rule Name**: protocol: <protocol> (TCP, UDP, ICMP) source_mac: <source-mac> source_ip: <source-ip> target_ip: <target-ip> port_range_start: <port-range-start> port_range_end: <port-range-end> icmp_type: <icmp-type> icmp_code: <icmp-code>

nat This option will enable NAT on the private NIC.

private_lan This option will connect the server to the specified private LAN. If no LAN exists, then a new private LAN will be created. The value accepts a LAN ID (integer).

private_firewall_rules This option allows for a list of firewall rules assigned to the private network interface.

> **Firewall Rule Name**: protocol: <protocol> (TCP, UDP, ICMP) source_mac: <source-mac> source_ip: <source-ip> target_ip: <target-ip> port_range_start: <port-range-start> port_range_end: <port-range-end> icmp_type: <icmp-type> icmp_code: <icmp-code>

ssh_private_key Full path to the SSH private key file.

ssh_public_key Full path to the SSH public key file.

ssh_interface This option will use the private LAN IP for node connections (such as bootstrapping the node) instead of the public LAN IP. The value accepts `private_lan'.

cpu_family This option allow the CPU family to be set to AMD_OPTERON or INTEL_XEON. The default is AMD_OPTERON.

volumes: This option allows a list of additional volumes by name that will be created and attached to the server. Each volume requires `disk_size' and, optionally, `disk_type'. The default is HDD.

deploy Set to False if Salt should not be installed on the node.

wait_for_timeout The timeout to wait in seconds for provisioning resources such as servers. The default wait_for_timeout is 15 minutes.

For more information concerning cloud profiles, see *here*.

13.7.20 Getting Started With Proxmox

Proxmox Virtual Environment is a complete server virtualization management solution, based on OpenVZ(in Proxmox up to 3.4)/LXC(from Proxmox 4.0 and up) and full virtualization with KVM. Further information can be found at:

http://www.proxmox.org/

Dependencies

- IPy >= 0.81
- requests >= 2.2.1

Please note: This module allows you to create OpenVZ/LXC containers and KVM VMs, but installing Salt on it will only be done on containers rather than a KVM virtual machine.

- Set up the cloud configuration at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/proxmox.conf`:

```
my-proxmox-config:
  # Set up the location of the salt master
  #
  minion:
    master: saltmaster.example.com

  # Set the PROXMOX access credentials (see below)
  #
  user: myuser@pve
  password: badpass

  # Set the access URL for your PROXMOX host
  #
  url: your.proxmox.host
  driver: proxmox
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Access Credentials

The `user`, `password`, and `url` will be provided to you by your cloud host. These are all required in order for the PROXMOX driver to work.

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or `/etc/salt/cloud.profiles.d/proxmox.conf`:

- Configure a profile to be used:

```
proxmox-ubuntu:
    provider: my-proxmox-config
    image: local:vztmpl/ubuntu-12.04-standard_12.04-1_amd64.tar.gz
    technology: lxc

    # host needs to be set to the configured name of the proxmox host
    # and not the ip address or FQDN of the server
    host: myvmhost
    ip_address: 192.168.100.155
    password: topsecret
```

The profile can be realized now with a salt command:

```
# salt-cloud -p proxmox-ubuntu myubuntu
```

This will create an instance named `myubuntu` on the cloud host. The minion that is installed on this instance will have a `hostname` of `myubuntu`. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt myubuntu test.ping
```

Required Settings

The following settings are always required for PROXMOX:

- Using the new cloud configuration format:

```
my-proxmox-config:
  driver: proxmox
  user: saltcloud@pve
  password: xyzzy
  url: your.proxmox.host
```

Optional Settings

Unlike other cloud providers in Salt Cloud, Proxmox does not utilize a `size` setting. This is because Proxmox allows the end-user to specify a more detailed configuration for their instances, than is allowed by many other cloud providers. The following options are available to be used in a profile, with their default settings listed.

```
# Description of the instance.
desc: <instance_name>

# How many CPU cores, and how fast they are (in MHz)
cpus: 1
cpuunits: 1000

# How many megabytes of RAM
memory: 256

# How much swap space in MB
swap: 256

# Whether to auto boot the vm after the host reboots
onboot: 1

# Size of the instance disk (in GiB)
disk: 10

# Host to create this vm on
host: myvmhost

# Nameservers. Defaults to host
nameserver: 8.8.8.8 8.8.4.4

# Username and password
ssh_username: root
password: <value from PROXMOX.password>

# The name of the image, from ``salt-cloud --list-images proxmox``
image: local:vztmpl/ubuntu-12.04-standard_12.04-1_amd64.tar.gz

# Whether or not to verify the SSL cert on the Proxmox host
verify_ssl: False
```

```
# Network interfaces, netX
net0: name=eth0,bridge=vmbr0,ip=dhcp

# Public key to add to /root/.ssh/authorized_keys.
pubkey: 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABA...'
```

QEMU

Some functionnalities works differently if you use `qemu' as technology. In order to create a new VM with qemu, you need to specificy some more information. You can also clone a qemu template which already is on your Proxmox server.

QEMU profile file (for a new VM):

```
proxmox-win7:
  # Image of the new VM
  image: image.iso # You can get all your available images using 'salt-cloud --list-
↪images provider_name' (Ex: 'salt-cloud --list-images my-proxmox-config')

  # Technology used to create the VM ('qemu', 'openvz'(on Proxmox <4.x) or 'lxc'(on
↪Proxmox 4.x+))
  technology: qemu

  # Proxmox node name
  host: node_name

  # Proxmox password
  password: your_password

  # Workaround https://github.com/saltstack/salt/issues/27821
  size: ''

  # RAM size (MB)
  memory: 2048

  # OS Type enum (other / wxp / w2k / w2k3 / w2k8 / wvista / win7 / win8 / l24 / l26 /
↪solaris)
  ostype: win7

  # Hard disk location
  sata0: <location>:<size>, format=<qcow2/vmdk/raw>, size=<size>GB #Example: local:
↪120,format=qcow2,size=120GB

  #CD/DVD Drive
  ide2: <content_location>,media=cdrom #Example: local:iso/name.iso,media=cdrom

  # Network Device
  net0:<model>,bridge=<bridge> #Example: e1000,bridge=vmbr0

  # Enable QEMU Guest Agent (0 / 1)
  agent: 1

  # VM name
  name: Test
```

More information about these parameters can be found on Proxmox API (http://pve.proxmox.com/pve2-api-doc/) under the `POST' method of nodes/{node}/qemu

QEMU profile file (for a clone):

```
proxmox-win7:
  # Enable Clone
  clone: True

  # New VM description
  clone_description: 'description'

  # New VM name
  clone_name: 'name'

  # New VM format (qcow2 / raw / vmdk)
  clone_format: qcow2

  # Full clone (1) or Link clone (0)
  clone_full: 0

  # VMID of Template to clone
  clone_from: ID

  # Technology used to create the VM ('qemu' or 'lxc')
  technology: qemu

  # Proxmox node name
  host: node_name

  # Proxmox password
  password: your_password

  # Workaround https://github.com/saltstack/salt/issues/27821
  size: ''
```

More information can be found on Proxmox API under the `POST' method of /nodes/{node}/qemu/{vmid}/clone

Note: The Proxmox API offers a lot more options and parameters, which are not yet supported by this salt-cloud `overlay'. Feel free to add your contribution by forking the github repository and modifying the following file: `salt/cloud/clouds/proxmox.py`

An easy way to support more parameters for VM creation would be to add the names of the optional parameters in the `create_nodes(vm_)' function, under the `qemu' technology. But it requires you to dig into the code …

13.7.21 Getting Started With Scaleway

Scaleway is the first IaaS host worldwide to offer an ARM based cloud. It's the ideal platform for horizontal scaling with BareMetal SSD servers. The solution provides on demand resources: it comes with on-demand SSD storage, movable IPs , images, security group and an Object Storage solution. https://scaleway.com

Configuration

Using Salt for Scaleway, requires an `access key` and an `API token`. `API tokens` are unique identifiers associated with your Scaleway account. To retrieve your `access key` and `API token`, log-in to the Scaleway control panel, open the pull-down menu on your account name and click on ``My Credentials'' link.

If you do not have API token you can create one by clicking the ``Create New Token'' button on the right corner.

```
# Note: This example is for /etc/salt/cloud.providers or any file in the
# /etc/salt/cloud.providers.d/ directory.

my-scaleway-config:
  access_key: 15cf404d-4560-41b1-9a0c-21c3d5c4ff1f
  token: a7347ec8-5de1-4024-a5e3-24b77d1ba91d
  driver: scaleway
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profiles

Cloud Profiles

Set up an initial profile at /etc/salt/cloud.profiles or in the /etc/salt/cloud.profiles.d/ directory:

```
scaleway-ubuntu:
  provider: my-scaleway-config
  image: Ubuntu Trusty (14.04 LTS)
```

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
#salt-cloud --list-images my-scaleway-config
my-scaleway-config:
  ----------
  scaleway:
      ----------
      069fd876-eb04-44ab-a9cd-47e2fa3e5309:
          ----------
          arch:
              arm
          creation_date:
              2015-03-12T09:35:45.764477+00:00
          default_bootscript:
              {u'kernel': {u'dtb': u'', u'title': u'Pimouss 3.2.34-30-std', u'id': u
↪'cfda4308-cd6f-4e51-9744-905fc0da370f', u'path': u'kernel/pimouss-uImage-3.2.34-30-
↪std'}, u'title': u'3.2.34-std #30 (stable)', u'id': u'c5af0215-2516-4316-befc-
↪5da1cfad609c', u'initrd': {u'path': u'initrd/c1-uInitrd', u'id': u'1be14b1b-e24c-
↪48e5-b0b6-7ba452e42b92', u'title': u'C1 initrd'}, u'bootcmdargs': {u'id': u
↪'d22c4dde-e5a4-47ad-abb9-d23b54d542ff', u'value': u'ip=dhcp boot=local root=/dev/
↪nbd0 USE_XNBD=1 nbd.max_parts=8'}, u'organization': u'11111111-1111-4111-8111-
↪111111111111', u'public': True}
          extra_volumes:
              []
          id:
              069fd876-eb04-44ab-a9cd-47e2fa3e5309
          modification_date:
              2015-04-24T12:02:16.820256+00:00
          name:
```

```
            Ubuntu Vivid (15.04)
        organization:
            a283af0b-d13e-42e1-a43f-855ffbf281ab
        public:
            True
        root_volume:
            {u'name': u'distrib-ubuntu-vivid-2015-03-12_10:32-snapshot', u'id': u
↪'a6d02e63-8dee-4bce-b627-b21730f35a05', u'volume_type': u'l_ssd', u'size':
↪50000000000L}
...
```

Execute a query and return all information about the nodes running on configured cloud providers using the -Q option for the salt-cloud command:

```
# salt-cloud -F
[INFO    ] salt-cloud starting
[INFO    ] Starting new HTTPS connection (1): api.scaleway.com
my-scaleway-config:
  ----------
  scaleway:
      ----------
      salt-manager:
          ----------
          creation_date:
              2015-06-03T08:17:38.818068+00:00
          hostname:
              salt-manager
...
```

Note: Additional documentation about Scaleway can be found at https://www.scaleway.com/docs.

13.7.22 Getting Started With Saltify

The Saltify driver is a driver for installing Salt on existing machines (virtual or bare metal).

Dependencies

The Saltify driver has no external dependencies.

Configuration

Because the Saltify driver does not use an actual cloud provider host, it can have a simple provider configuration. The only thing that is required to be set is the driver name, and any other potentially useful information, like the location of the salt-master:

```
# Note: This example is for /etc/salt/cloud.providers file or any file in
# the /etc/salt/cloud.providers.d/ directory.

my-saltify-config:
  minion:
    master: 111.222.333.444
  driver: saltify
```

However, if you wish to use the more advanced capabilities of salt-cloud, such as rebooting, listing, and disconnecting machines, then the salt master must fill the role usually performed by a vendor's cloud management system. The salt master must be running on the salt-cloud machine, and created nodes must be connected to the master.

Additional information about which configuration options apply to which actions can be studied in the *Saltify Module documentation* and the *Miscellaneous Salt Cloud Options* document.

Profiles

Saltify requires a separate profile to be configured for each machine that needs Salt installed [1]. The initial profile can be set up at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory. Each profile requires both an `ssh_host` and an `ssh_username` key parameter as well as either an `key_filename` or a `password`.

Profile configuration example:

```
# /etc/salt/cloud.profiles.d/saltify.conf

salt-this-machine:
  ssh_host: 12.34.56.78
  ssh_username: root
  key_filename: '/etc/salt/mysshkey.pem'
  provider: my-saltify-config
```

The machine can now be ``Salted'' with the following command:

```
salt-cloud -p salt-this-machine my-machine
```

This will install salt on the machine specified by the cloud profile, `salt-this-machine`, and will give the machine the minion id of `my-machine`. If the command was executed on the salt-master, its Salt key will automatically be accepted by the master.

Once a salt-minion has been successfully installed on the instance, connectivity to it can be verified with Salt:

```
salt my-machine test.version
```

Destroy Options

New in version 2018.3.0.

For obvious reasons, the `destroy` action does not actually vaporize hardware. If the salt master is connected, it can tear down parts of the client machines. It will remove the client's key from the salt master, and can execute the following options:

```
- remove_config_on_destroy: true
  # default: true
  # Deactivate salt-minion on reboot and
  # delete the minion config and key files from its "/etc/salt" directory,
  #   NOTE: If deactivation was unsuccessful (older Ubuntu machines) then when
  #   salt-minion restarts it will automatically create a new, unwanted, set
  #   of key files. Use the "force_minion_config" option to replace them.

- shutdown_on_destroy: false
  # default: false
  # last of all, send a "shutdown" command to the client.
```

[1] Unless you are using a map file to provide the unique parameters.

Wake On LAN

New in version 2018.3.0.

In addition to connecting a hardware machine to a Salt master, you have the option of sending a wake-on-LAN magic packet to start that machine running.

The ``magic packet'' must be sent by an existing salt minion which is on the same network segment as the target machine. (Or your router must be set up especially to route WoL packets.) Your target machine must be set up to listen for WoL and to respond appropriately.

You must provide the Salt node id of the machine which will send the WoL packet (parameter `wol_sender_node`), and the hardware MAC address of the machine you intend to wake, (parameter `wake_on_lan_mac`). If both parameters are defined, the WoL will be sent. The cloud master will then sleep a while (parameter `wol_boot_wait`) to give the target machine time to boot up before we start probing its SSH port to begin deploying Salt to it. The default sleep time is 30 seconds.

```
# /etc/salt/cloud.profiles.d/saltify.conf

salt-this-machine:
  ssh_host: 12.34.56.78
  ssh_username: root
  key_filename: '/etc/salt/mysshkey.pem'
  provider: my-saltify-config
  wake_on_lan_mac: '00:e0:4c:70:2a:b2'  # found with ifconfig
  wol_sender_node: bevymaster  # its on this network segment
  wol_boot_wait: 45  # seconds to sleep
```

Using Map Files

The settings explained in the section above may also be set in a map file. An example of how to use the Saltify driver with a map file follows:

```
# /etc/salt/saltify-map

make_salty:
  - my-instance-0:
      ssh_host: 12.34.56.78
      ssh_username: root
      password: very-bad-password
  - my-instance-1:
      ssh_host: 44.33.22.11
      ssh_username: root
      password: another-bad-pass
```

Note: When using a cloud map with the Saltify driver, the name of the profile to use, in this case `make_salty`, must be defined in a profile config. For example:

```
# /etc/salt/cloud.profiles.d/saltify.conf

make_salty:
  provider: my-saltify-config
```

The machines listed in the map file can now be ``Salted'' by applying the following salt map command:

```
salt-cloud -m /etc/salt/saltify-map
```

This command will install salt on the machines specified in the map and will give each machine their minion id of `my-instance-0` and `my-instance-1`, respectively. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Connectivity to the new ``Salted'' instances can now be verified with Salt:

```
salt 'my-instance-*' test.ping
```

Credential Verification

Because the Saltify driver does not actually create VM's, unlike other salt-cloud drivers, it has special behaviour when the `deploy` option is set to `False`. When the cloud configuration specifies `deploy: False`, the Saltify driver will attept to authenticate to the target node(s) and return `True` for each one that succeeds. This can be useful to verify ports, protocols, services and credentials are correctly configured before a live deployment.

Return values:

- `True`: Credential verification succeeded
- `False`: Credential verification succeeded
- `None`: Credential verification was not attempted.

13.7.23 Getting Started With SoftLayer

SoftLayer is a public cloud host, and baremetal hardware hosting service.

Dependencies

The SoftLayer driver for Salt Cloud requires the softlayer package, which is available at PyPI:

https://pypi.python.org/pypi/SoftLayer

This package can be installed using `pip` or `easy_install`:

```
# pip install softlayer
# easy_install softlayer
```

Configuration

Set up the cloud config at `/etc/salt/cloud.providers`:

```
# Note: These examples are for /etc/salt/cloud.providers

  my-softlayer:
    # Set up the location of the salt master
    minion:
      master: saltmaster.example.com

    # Set the SoftLayer access credentials (see below)
    user: MYUSER1138
    apikey: 'e3b68aa711e6deadc62d5b76355674beef7cc3116062ddbacafe5f7e465bfdc9'
```

```
    driver: softlayer


  my-softlayer-hw:
    # Set up the location of the salt master
    minion:
      master: saltmaster.example.com

    # Set the SoftLayer access credentials (see below)
    user: MYUSER1138
    apikey: 'e3b68aa711e6deadc62d5b76355674beef7cc3116062ddbacafe5f7e465bfdc9'

    driver: softlayer_hw
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Access Credentials

The `user` setting is the same user as is used to log into the SoftLayer Administration area. The `apikey` setting is found inside the Admin area after logging in:

- Hover over the `Account` menu item.
- Click the `Users` link.
- Find the `API Key` column and click `View`.

Profiles

Cloud Profiles

Set up an initial profile at `/etc/salt/cloud.profiles`:

```
base_softlayer_ubuntu:
  provider: my-softlayer
  image: UBUNTU_LATEST
  cpu_number: 1
  ram: 1024
  disk_size: 100
  local_disk: True
  hourly_billing: True
  domain: example.com
  location: sjc01
  # Optional
  max_net_speed: 1000
  private_vlan: 396
  private_network: True
  private_ssh: True
  # Use a dedicated host instead of cloud
```

```
  dedicated_host_id: 1234
  # May be used _instead_of_ image
  global_identifier: 320d8be5-46c0-dead-cafe-13e3c51
```

Most of the above items are required; optional items are specified below.

image

Images to build an instance can be found using the `--list-images` option:

```
# salt-cloud --list-images my-softlayer
```

The setting used will be labeled as `template`.

cpu_number

This is the number of CPU cores that will be used for this instance. This number may be dependent upon the image that is used. For instance:

```
Red Hat Enterprise Linux 6 - Minimal Install (64 bit) (1 - 4 Core):
    ----------
    name:
        Red Hat Enterprise Linux 6 - Minimal Install (64 bit) (1 - 4 Core)
    template:
        REDHAT_6_64
Red Hat Enterprise Linux 6 - Minimal Install (64 bit) (5 - 100 Core):
    ----------
    name:
        Red Hat Enterprise Linux 6 - Minimal Install (64 bit) (5 - 100 Core)
    template:
        REDHAT_6_64
```

Note that the template (meaning, the *image* option) for both of these is the same, but the names suggests how many CPU cores are supported.

ram

This is the amount of memory, in megabytes, that will be allocated to this instance.

disk_size

The amount of disk space that will be allocated to this image, in gigabytes.

```
base_softlayer_ubuntu:
  disk_size: 100
```

Using Multiple Disks

New in version 2015.8.1.

SoftLayer allows up to 5 disks to be specified for a virtual machine upon creation. Multiple disks can be specified either as a list or a comma-delimited string. The first `disk_size` specified in the string or list will be the first disk size assigned to the VM.

List Example: .. code-block:: yaml

> **base_softlayer_ubuntu:** disk_size: [`100', `20', `20']

String Example: .. code-block:: yaml

> **base_softlayer_ubuntu:** disk_size: `100, 20, 20'

local_disk

When true the disks for the computing instance will be provisioned on the host which it runs, otherwise SAN disks will be provisioned.

hourly_billing

When true the computing instance will be billed on hourly usage, otherwise it will be billed on a monthly basis.

domain

The domain name that will be used in the FQDN (Fully Qualified Domain Name) for this instance. The *domain* setting will be used in conjunction with the instance name to form the FQDN.

use_fqdn

If set to True, the Minion will be identified by the FQDN (Fully Qualified Domain Name) which is a result of combining the `domain` configuration value and the Minion name specified either via the CLI or a map file rather than only using the short host name, or Minion ID. Default is False.

New in version 2016.3.0.

For example, if the value of `domain` is `example.com` and a new VM was created via the CLI with `salt-cloud -p base_softlayer_ubuntu my-vm`, the resulting Minion ID would be `my-vm.example.com`.

Note: When enabling the `use_fqdn` setting, the Minion ID will be the FQDN and will interact with salt commands with the FQDN instead of the short hostname. However, due to the way the SoftLayer API is constructed, some Salt Cloud functions such as listing nodes or destroying VMs will only list the short hostname of the VM instead of the FQDN.

Example output displaying the SoftLayer hostname quirk mentioned in the note above (note the Minion ID is `my-vm.example.com`, but the VM to be destroyed is listed with its short hostname, `my-vm`):

```
# salt-key -L
Accepted Keys:
my-vm.example.com
Denied Keys:
Unaccepted Keys:
Rejected Keys:
#
#
```

```
# salt my-vm.example.com test.ping
my-vm.example.com:
    True
#
#
# salt-cloud -d my-vm.example.com
[INFO    ] salt-cloud starting
[INFO    ] POST https://api.softlayer.com/xmlrpc/v3.1/SoftLayer_Account
The following virtual machines are set to be destroyed:
  softlayer-config:
    softlayer:
      my-vm

Proceed? [N/y] y
... proceeding
[INFO    ] Destroying in non-parallel mode.
[INFO    ] POST https://api.softlayer.com/xmlrpc/v3.1/SoftLayer_Account
[INFO    ] POST https://api.softlayer.com/xmlrpc/v3.1/SoftLayer_Virtual_Guest
softlayer-config:
    ----------
    softlayer:
        ----------
        my-vm:
            True
```

location

Images to build an instance can be found using the *--list-locations* option:

```
# salt-cloud --list-location my-softlayer
```

max_net_speed

Specifies the connection speed for the instance's network components. This setting is optional. By default, this is set to 10.

post_uri

Specifies the uri location of the script to be downloaded and run after the instance is provisioned.

New in version 2015.8.1.

Example: .. code-block:: yaml

> **base_softlayer_ubuntu:** post_uri: `https://SOMESERVERIP:8000/myscript.sh`

public_vlan

If it is necessary for an instance to be created within a specific frontend VLAN, the ID for that VLAN can be specified in either the provider or profile configuration.

This ID can be queried using the *list_vlans* function, as described below. This setting is optional.

If this setting is set to *None*, salt-cloud will connect to the private ip of the server.

Note: If this setting is not provided and the server is not built with a public vlan, *private_ssh* or *private_wds* will need to be set to make sure that salt-cloud attempts to connect to the private ip.

private_vlan

If it is necessary for an instance to be created within a specific backend VLAN, the ID for that VLAN can be specified in either the provider or profile configuration.

This ID can be queried using the *list_vlans* function, as described below. This setting is optional.

private_network

If a server is to only be used internally, meaning it does not have a public VLAN associated with it, this value would be set to True. This setting is optional. The default is False.

private_ssh or private_wds

Whether to run the deploy script on the server using the public IP address or the private IP address. If set to True, Salt Cloud will attempt to SSH or WinRM into the new server using the private IP address. The default is False. This settiong is optional.

global_identifier

When creating an instance using a custom template, this option is set to the corresponding value obtained using the *list_custom_images* function. This option will not be used if an *image* is set, and if an *image* is not set, it is required.

The profile can be realized now with a salt command:

```
# salt-cloud -p base_softlayer_ubuntu myserver
```

Using the above configuration, this will create *myserver.example.com*.

Once the instance has been created with salt-minion installed, connectivity to it can be verified with Salt:

```
# salt 'myserver.example.com' test.ping
```

Dedicated Host

Soflayer allows the creation of new VMs in a dedicated host. This means that you can order and pay a fixed amount for a bare metal dedicated host and use it to provision as many VMs as you can fit in there. If you want your VMs to be launched in a dedicated host, instead of Sofltayer's cloud, set the `dedicated_host_id` parameter in your profile.

dedicated_host_id

The id of the dedicated host where the VMs should be created. If not set, VMs will be created in Softlayer's cloud instead.

Bare metal Profiles

Set up an initial profile at /etc/salt/cloud.profiles:

```
base_softlayer_hw_centos:
  provider: my-softlayer-hw
  # CentOS 6.0 - Minimal Install (64 bit)
  image: 13963
  # 2 x 2.0 GHz Core Bare Metal Instance - 2 GB Ram
  size: 1921
  # 500GB SATA II
  hdd: 1267
  # San Jose 01
  location: 168642
  domain: example.com
  # Optional
  vlan: 396
  port_speed: 273
  banwidth: 248
```

Most of the above items are required; optional items are specified below.

image

Images to build an instance can be found using the *--list-images* option:

```
# salt-cloud --list-images my-softlayer-hw
```

A list of *id`s and names will be provided. The `name* will describe the operating system and architecture. The *id* will be the setting to be used in the profile.

size

Sizes to build an instance can be found using the *--list-sizes* option:

```
# salt-cloud --list-sizes my-softlayer-hw
```

A list of *id`s and names will be provided. The `name* will describe the speed and quantity of CPU cores, and the amount of memory that the hardware will contain. The *id* will be the setting to be used in the profile.

hdd

There is currently only one size of hard disk drive (HDD) that is available for hardware instances on SoftLayer:

```
1267: 500GB SATA II
```

The *hdd* setting in the profile should be 1267. Other sizes may be added in the future.

location

Locations to build an instance can be found using the *--list-images* option:

```
# salt-cloud --list-locations my-softlayer-hw
```

A list of IDs and names will be provided. The *location* will describe the location in human terms. The *id* will be the setting to be used in the profile.

domain

The domain name that will be used in the FQDN (Fully Qualified Domain Name) for this instance. The *domain* setting will be used in conjunction with the instance name to form the FQDN.

vlan

If it is necessary for an instance to be created within a specific VLAN, the ID for that VLAN can be specified in either the provider or profile configuration.

This ID can be queried using the *list_vlans* function, as described below.

port_speed

Specifies the speed for the instance's network port. This setting refers to an ID within the SoftLayer API, which sets the port speed. This setting is optional. The default is 273, or, 100 Mbps Public & Private Networks. The following settings are available:

- 273: 100 Mbps Public & Private Networks
- 274: 1 Gbps Public & Private Networks
- 21509: 10 Mbps Dual Public & Private Networks (up to 20 Mbps)
- 21513: 100 Mbps Dual Public & Private Networks (up to 200 Mbps)
- 2314: 1 Gbps Dual Public & Private Networks (up to 2 Gbps)
- 272: 10 Mbps Public & Private Networks

bandwidth

Specifies the network bandwidth available for the instance. This setting refers to an ID within the SoftLayer API, which sets the bandwidth. This setting is optional. The default is 248, or, 5000 GB Bandwidth. The following settings are available:

- 248: 5000 GB Bandwidth
- 129: 6000 GB Bandwidth
- 130: 8000 GB Bandwidth
- 131: 10000 GB Bandwidth
- 36: Unlimited Bandwidth (10 Mbps Uplink)
- 125: Unlimited Bandwidth (100 Mbps Uplink)

Actions

The following actions are currently supported by the SoftLayer Salt Cloud driver.

show_instance

This action is a thin wrapper around *--full-query*, which displays details on a single instance only. In an environment with several machines, this will save a user from having to sort through all instance data, just to examine a single instance.

```
$ salt-cloud -a show_instance myinstance
```

Functions

The following functions are currently supported by the SoftLayer Salt Cloud driver.

list_vlans

This function lists all VLANs associated with the account, and all known data from the SoftLayer API concerning those VLANs.

```
$ salt-cloud -f list_vlans my-softlayer
$ salt-cloud -f list_vlans my-softlayer-hw
```

The *id* returned in this list is necessary for the *vlan* option when creating an instance.

list_custom_images

This function lists any custom templates associated with the account, that can be used to create a new instance.

```
$ salt-cloud -f list_custom_images my-softlayer
```

The *globalIdentifier* returned in this list is necessary for the *global_identifier* option when creating an image using a custom template.

Optional Products for SoftLayer HW

The softlayer_hw driver supports the ability to add optional products, which are supported by SoftLayer's API. These products each have an ID associated with them, that can be passed into Salt Cloud with the *optional_products* option:

```
softlayer_hw_test:
  provider: my-softlayer-hw
  # CentOS 6.0 - Minimal Install (64 bit)
  image: 13963
  # 2 x 2.0 GHz Core Bare Metal Instance - 2 GB Ram
  size: 1921
  # 500GB SATA II
  hdd: 1267
  # San Jose 01
  location: 168642
  domain: example.com
```

```
optional_products:
  # MySQL for Linux
  - id: 28
  # Business Continuance Insurance
  - id: 104
```

These values can be manually obtained by looking at the source of an order page on the SoftLayer web interface. For convenience, many of these values are listed here:

Public Secondary IP Addresses

- 22: 4 Public IP Addresses
- 23: 8 Public IP Addresses

Primary IPv6 Addresses

- 17129: 1 IPv6 Address

Public Static IPv6 Addresses

- 1481: /64 Block Static Public IPv6 Addresses

OS-Specific Addon

- 17139: XenServer Advanced for XenServer 6.x
- 17141: XenServer Enterprise for XenServer 6.x
- 2334: XenServer Advanced for XenServer 5.6
- 2335: XenServer Enterprise for XenServer 5.6
- 13915: Microsoft WebMatrix
- 21276: VMware vCenter 5.1 Standard

Control Panel Software

- 121: cPanel/WHM with Fantastico and RVskin
- 20778: Parallels Plesk Panel 11 (Linux) 100 Domain w/ Power Pack
- 20786: Parallels Plesk Panel 11 (Windows) 100 Domain w/ Power Pack
- 20787: Parallels Plesk Panel 11 (Linux) Unlimited Domain w/ Power Pack
- 20792: Parallels Plesk Panel 11 (Windows) Unlimited Domain w/ Power Pack
- 2340: Parallels Plesk Panel 10 (Linux) 100 Domain w/ Power Pack
- 2339: Parallels Plesk Panel 10 (Linux) Unlimited Domain w/ Power Pack
- 13704: Parallels Plesk Panel 10 (Windows) Unlimited Domain w/ Power Pack

Database Software

- 29: MySQL 5.0 for Windows
- 28: MySQL for Linux
- 21501: Riak 1.x
- 20893: MongoDB
- 30: Microsoft SQL Server 2005 Express
- 92: Microsoft SQL Server 2005 Workgroup
- 90: Microsoft SQL Server 2005 Standard
- 94: Microsoft SQL Server 2005 Enterprise
- 1330: Microsoft SQL Server 2008 Express
- 1340: Microsoft SQL Server 2008 Web
- 1337: Microsoft SQL Server 2008 Workgroup
- 1334: Microsoft SQL Server 2008 Standard
- 1331: Microsoft SQL Server 2008 Enterprise
- 2179: Microsoft SQL Server 2008 Express R2
- 2173: Microsoft SQL Server 2008 Web R2
- 2183: Microsoft SQL Server 2008 Workgroup R2
- 2180: Microsoft SQL Server 2008 Standard R2
- 2176: Microsoft SQL Server 2008 Enterprise R2

Anti-Virus & Spyware Protection

- 594: McAfee VirusScan Anti-Virus - Windows
- 414: McAfee Total Protection - Windows

Insurance

- 104: Business Continuance Insurance

Monitoring

- 55: Host Ping
- 56: Host Ping and TCP Service Monitoring

Notification

- 57: Email and Ticket

Advanced Monitoring

- 2302: Monitoring Package - Basic
- 2303: Monitoring Package - Advanced
- 2304: Monitoring Package - Premium Application

Response

- 58: Automated Notification
- 59: Automated Reboot from Monitoring
- 60: 24x7x365 NOC Monitoring, Notification, and Response

Intrusion Detection & Protection

- 413: McAfee Host Intrusion Protection w/Reporting

Hardware & Software Firewalls

- 411: APF Software Firewall for Linux
- 894: Microsoft Windows Firewall
- 410: 10Mbps Hardware Firewall
- 409: 100Mbps Hardware Firewall
- 408: 1000Mbps Hardware Firewall

13.7.24 Getting Started With Vagrant

The Vagrant driver is a new, experimental driver for spinning up a VagrantBox virtual machine, and installing Salt on it.

Dependencies

The Vagrant driver itself has no external dependencies.

The machine which will host the VagrantBox must be an already existing minion of the cloud server's Salt master. It must have Vagrant installed, and a Vagrant-compatible virtual machine engine, such as VirtualBox. (Note: The Vagrant driver does not depend on the salt-cloud VirtualBox driver in any way.)

[Caution: The version of Vagrant packaged for `apt install` in Ubuntu 16.04 will not connect a bridged network adapter correctly. Use a version downloaded directly from the web site.]

Include the Vagrant guest editions plugin: `vagrant plugin install vagrant-vbguest`.

Configuration

Configuration of the client virtual machine (using VirtualBox, VMware, etc) will be done by Vagrant as specified in the Vagrantfile on the host machine.

Salt-cloud will push the commands to install and provision a salt minion on the virtual machine, so you need not (perhaps **should** not) provision salt in your Vagrantfile, in most cases.

If, however, your cloud master cannot open an SSH connection to the child VM, you may **need** to let Vagrant provision the VM with Salt, and use some other method (such as passing a pillar dictionary to the VM) to pass the master's IP address to the VM. The VM can then attempt to reach the salt master in the usual way for non-cloud minions. Specify the profile configuration argument as `deploy: False` to prevent the cloud master from trying.

```
# Note: This example is for /etc/salt/cloud.providers file or any file in
# the /etc/salt/cloud.providers.d/ directory.

my-vagrant-config:
  minion:
    master: 111.222.333.444
  provider: vagrant
```

Because the Vagrant driver needs a place to store the mapping between the node name you use for Salt commands and the Vagrantfile which controls the VM, you must configure your salt minion as a Salt smb server. (See *host provisioning example* below.)

Profiles

Vagrant requires a profile to be configured for each machine that needs Salt installed. The initial profile can be set up at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory.

Each profile requires a `vagrantfile` parameter. If the Vagrantfile has definitions for multiple machines then you need a `machine` parameter,

Salt-cloud uses SSH to provision the minion. There must be a routable path from the cloud master to the VM. Usually, you will want to use a bridged network adapter for SSH. The address may not be known until DHCP assigns it. If `ssh_host` is not defined, and `target_network` is defined, the driver will attempt to read the address from the output of an `ifconfig` command. Lacking either setting, the driver will try to use the value Vagrant returns as its `ssh_host`, which will work only if the cloud master is running somewhere on the same host.

The `target_network` setting should be used to identify the IP network your bridged adapter is expected to appear on. Use CIDR notation, like `target_network: '2001:DB8::/32'` or `target_network: '192.0.2.0/24'`.

Profile configuration example:

```
# /etc/salt/cloud.profiles.d/vagrant.conf

vagrant-machine:
  host: my-vhost  # the Salt id of the virtual machine's host computer.
  provider: my-vagrant-config
  cwd: /srv/machines  # the path to your Virtualbox file.
  vagrant_runas: my-username  # the username who defined the Vagrantbox on the host
  # vagrant_up_timeout: 300 # (seconds) timeout for cmd.run of the "vagrant up" command
  # vagrant_provider: '' # option for "vagrant up" like: "--provider vmware_fusion"
  # ssh_host: None  # "None" means try to find the routable IP address from "ifconfig"
  # ssh_username: '' # also required when ssh_host is used.
```

```
  # target_network: None  # Expected CIDR address range of your bridged network
  # force_minion_config: false  # Set "true" to re-purpose an existing VM
```

The machine can now be created and configured with the following command:

```
salt-cloud -p vagrant-machine my-id
```

This will create the machine specified by the cloud profile `vagrant-machine`, and will give the machine the minion id of `my-id`. If the cloud master is also the salt-master, its Salt key will automatically be accepted on the master.

Once a salt-minion has been successfully installed on the instance, connectivity to it can be verified with Salt:

```
salt my-id test.ping
```

Provisioning a Vagrant cloud host (example)

In order to query or control minions it created, each host minion needs to track the Salt node names associated with any guest virtual machines on it. It does that using a Salt sdb database.

The Salt sdb is not configured by default. The following example shows a simple installation.

This example assumes:

- you are on a large network using the 10.x.x.x IP address space
- your Salt master's Salt id is ``bevymaster''
- it will also be your salt-cloud controller
- it is at hardware address 10.124.30.7
- it is running a recent Debian family Linux (raspbian)
- your workstation is a Salt minion of bevymaster
- your workstation's minion id is ``my_laptop''
- VirtualBox has been installed on ``my_laptop'' (apt install is okay)
- Vagrant was installed from vagrantup.com. (not the 16.04 Ubuntu apt)
- ``my_laptop'' has done ``vagrant plugin install vagrant-vbguest''
- the VM you want to start is on ``my_laptop'' at ``/home/my_username/Vagrantfile''

```
# file /etc/salt/minion.d/vagrant_sdb.conf on host computer "my_laptop"
#  -- this sdb database is required by the Vagrant module --
vagrant_sdb_data:  # The sdb database must have this name.
  driver: sqlite3  # Let's use SQLite to store the data ...
  database: /var/cache/salt/vagrant.sqlite  # ... in this file ...
  table: sdb  # ... using this table name.
  create_table: True  # if not present
```

Remember to re-start your minion after changing its configuration files...

```
sudo systemctl restart salt-minion
```

```
# -*- mode: ruby -*-
# file /home/my_username/Vagrantfile on host computer "my_laptop"
BEVY = "bevy1"
```

```
DOMAIN = BEVY + ".test"  # .test is an ICANN reserved non-public TLD

# must supply a list of names to avoid Vagrant asking for interactive input
def get_good_ifc()   # try to find a working Ubuntu network adapter name
  addr_infos = Socket.getifaddrs
  addr_infos.each do |info|
    a = info.addr
    if a and a.ip? and not a.ip_address.start_with?("127.")
     return info.name
     end
  end
  return "eth0"  # fall back to an old reliable name
end

Vagrant.configure(2) do |config|
  config.ssh.forward_agent = true  # so you can use git ssh://...

  # add a bridged network interface. (try to detect name, then guess MacOS names, too)
  interface_guesses = [get_good_ifc(), 'en0: Ethernet', 'en1: Wi-Fi (AirPort)']
  config.vm.network "public_network", bridge: interface_guesses
  if ARGV[0] == "up"
    puts "Trying bridge network using interfaces: #{interface_guesses}"
  end
  config.vm.provision "shell", inline: "ip address", run: "always"  # make user feel
 ↪good

  # . . . . . . . . . . . . Define machine QUAIL1 . . . . . . . . . . . . . .
  config.vm.define "quail1", primary: true do |quail_config|
    quail_config.vm.box = "boxesio/xenial64-standard"  # a public VMware & Virtualbox
 ↪box
    quail_config.vm.hostname = "quail1." + DOMAIN  # supply a name in our bevy
    quail_config.vm.provider "virtualbox" do |v|
        v.memory = 1024       # limit memory for the virtual box
        v.cpus = 1
        v.linked_clone = true # make a soft copy of the base Vagrant box
       v.customize ["modifyvm", :id, "--natnet1", "192.168.128.0/24"]  # do not use 10.
 ↪x network for NAT
    end
  end
end
```

```
# file /etc/salt/cloud.profiles.d/my_vagrant_profiles.conf on bevymaster
q1:
  host: my_laptop  # the Salt id of your virtual machine host
  machine: quail1   # a machine name in the Vagrantfile (if not primary)
  vagrant_runas: my_username  # owner of Vagrant box files on "my_laptop"
  cwd: '/home/my_username' # the path (on "my_laptop") of the Vagrantfile
  provider: my_vagrant_provider  # name of entry in provider.conf file
  target_network: '10.0.0.0/8'  # VM external address will be somewhere here
```

```
# file /etc/salt/cloud.providers.d/vagrant_provider.conf on bevymaster
my_vagrant_provider:
  driver: vagrant
  minion:
    master: 10.124.30.7  # the hard address of the master
```

Create and use your new Salt minion

- Typing on the Salt master computer `bevymaster`, tell it to create a new minion named `v1` using profile `q1`...

```
sudo salt-cloud -p q1 v1
sudo salt v1 network.ip_addrs
  [ you get a list of IP addresses, including the bridged one ]
```

- logged in to your laptop (or some other computer known to GitHub)...

  [NOTE:] if you are using MacOS, you need to type `ssh-add -K` after each boot, unless you use one of the methods in this gist.

```
ssh -A vagrant@< the bridged network address >
  # [ or, if you are at /home/my_username/ on my_laptop ]
vagrant ssh quail1
```

- then typing on your new node ``v1'' (a.k.a. quail1.bevy1.test)...

```
password: vagrant
  # [ stuff types out ... ]

ls -al /vagrant
  # [ should be shared /home/my_username from my_laptop ]

# you can access other network facilities using the ssh authorization
# as recorded in your ~.ssh/ directory on my_laptop ...

sudo apt update
sudo apt install git
git clone ssh://git@github.com/yourID/your_project
# etc...
```

13.7.25 Getting Started with VEXXHOST

VEXXHOST is a cloud computing host which provides Canadian cloud computing services which are based in Monteral and use the libcloud OpenStack driver. VEXXHOST currently runs the Havana release of OpenStack. When provisioning new instances, they automatically get a public IP and private IP address. Therefore, you do not need to assign a floating IP to access your instance after it's booted.

Cloud Provider Configuration

To use the *openstack* driver for the VEXXHOST public cloud, you will need to set up the cloud provider configuration file as in the example below:

`/etc/salt/cloud.providers.d/vexxhost.conf`: In order to use the VEXXHOST public cloud, you will need to setup a cloud provider configuration file as in the example below which uses the OpenStack driver.

```
my-vexxhost-config:
  # Set the location of the salt-master
  #
  minion:
    master: saltmaster.example.com
```

```
# Configure VEXXHOST using the OpenStack plugin
#
identity_url: http://auth.api.thenebulacloud.com:5000/v2.0/tokens
compute_name: nova

# Set the compute region:
#
compute_region: na-yul-nhs1

# Configure VEXXHOST authentication credentials
#
user: your-tenant-id
password: your-api-key
tenant: your-tenant-name

# keys to allow connection to the instance launched
#
ssh_key_name: yourkey
ssh_key_file: /path/to/key/yourkey.priv

driver: openstack
```

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Authentication

All of the authentication fields that you need can be found by logging into your VEXXHOST customer center. Once you've logged in, you will need to click on ``CloudConsole'' and then click on ``API Credentials''.

Cloud Profile Configuration

In order to get the correct image UUID and the instance type to use in the cloud profile, you can run the following command respectively:

```
# salt-cloud --list-images=vexxhost-config
# salt-cloud --list-sizes=vexxhost-config
```

Once you have that, you can go ahead and create a new cloud profile. This profile will build an Ubuntu 12.04 LTS *nb.2G* instance.

`/etc/salt/cloud.profiles.d/vh_ubuntu1204_2G.conf`:

```
vh_ubuntu1204_2G:
  provider: my-vexxhost-config
  image: 4051139f-750d-4d72-8ef0-074f2ccc7e5a
  size: nb.2G
```

Provision an instance

To create an instance based on the sample profile that we created above, you can run the following *salt-cloud* command.

```
# salt-cloud -p vh_ubuntu1204_2G vh_instance1
```

Typically, instances are provisioned in under 30 seconds on the VEXXHOST public cloud. After the instance provisions, it will be set up a minion and then return all the instance information once it's complete.

Once the instance has been setup, you can test connectivity to it by running the following command:

```
# salt vh_instance1 test.ping
```

You can now continue to provision new instances and they will all automatically be set up as minions of the master you've defined in the configuration file.

13.7.26 Getting Started With Virtualbox

The Virtualbox cloud module allows you to manage a **local** Virtualbox hypervisor. Remote hypervisors may come later on.

Dependencies

The virtualbox module for Salt Cloud requires the Virtualbox SDK which is contained in a virtualbox installation from

https://www.virtualbox.org/wiki/Downloads

Configuration

The Virtualbox cloud module just needs to use the virtualbox driver for now. Virtualbox will be run as the running user.

`/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/virtualbox.conf`:

```
virtualbox-config:
   driver: virtualbox
```

Profiles

Set up an initial profile at `/etc/salt/cloud.profiles` or `/etc/salt/cloud.profiles.d/virtualbox.conf`:

```
virtualbox-test:
    provider: virtualbox-config
    clonefrom: VM_to_clone_from
    # Optional
    power_on: True
    deploy: True
    ssh_username: a_username
    password: a_password
    sudo: a_username
    sudo_password: a_password
    # Example minion config
```

```
    minion:
        master: localhost
    make_master: True
```

clonefrom **Mandatory** Enter the name of the VM/template to clone from.

So far only machines can only be cloned and automatically provisioned by Salt Cloud.

Provisioning

In order to provision when creating a new machine `power_on` and `deploy` have to be `True`.

Furthermore to connect to the VM `ssh_username` and `password` will have to be set.

`sudo` and `sudo_password` are the credentials for getting root access in order to deploy salt

Actions

start Attempt to boot a VM by name. VMs should have unique names in order to boot the correct one.

stop Attempt to stop a VM. This is akin to a force shutdown or 5 second press.

Functions

show_image Show all available information about a VM given by the *image* parameter

```
$ salt-cloud -f show_image virtualbox image=my_vm_name
```

13.7.27 Getting Started With VMware

New in version 2015.5.4.

Author: Nitin Madhok <nmadhok@clemson.edu>

The VMware cloud module allows you to manage VMware ESX, ESXi, and vCenter.

Dependencies

The vmware module for Salt Cloud requires the `pyVmomi` package, which is available at PyPI:

https://pypi.python.org/pypi/pyvmomi

This package can be installed using *pip* or *easy_install*:

```
pip install pyvmomi
easy_install pyvmomi
```

Note: Version 6.0 of pyVmomi has some problems with SSL error handling on certain versions of Python. If using version 6.0 of pyVmomi, the machine that you are running the proxy minion process from must have either Python 2.7.9 or newer This is due to an upstream dependency in pyVmomi 6.0 that is not supported in Python version 2.6 to 2.7.8. If the version of Python running the salt-cloud command is not in the supported range, you will need to install an earlier version of pyVmomi. See Issue #29537 for more information.

Note: pyVmomi doesn't expose the ability to specify the locale when connecting to VMware. This causes parsing issues when connecting to an instance of VMware running under a non-English locale. Until this feature is added upstream Issue #38402 contains a workaround.

Configuration

The VMware cloud module needs the vCenter or ESX/ESXi URL, username and password to be set up in the cloud configuration at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/vmware.conf`:

```
my-vmware-config:
  driver: vmware
  user: 'DOMAIN\user'
  password: 'verybadpass'
  url: '10.20.30.40'

vcenter01:
  driver: vmware
  user: 'DOMAIN\user'
  password: 'verybadpass'
  url: 'vcenter01.domain.com'
  protocol: 'https'
  port: 443

vcenter02:
  driver: vmware
  user: 'DOMAIN\user'
  password: 'verybadpass'
  url: 'vcenter02.domain.com'
  protocol: 'http'
  port: 80

esx01:
  driver: vmware
  user: 'admin'
  password: 'verybadpass'
  url: 'esx01.domain.com'
```

Note: Optionally, `protocol` and `port` can be specified if the vCenter server is not using the defaults. Default is `protocol: https` and `port: 443`.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider configuration was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile configuration. Cloud provider configuration now uses `driver` to refer to the salt-cloud driver that provides the underlying functionality to connect to a cloud provider, while cloud profile configuration continues to use `provider` to refer to the cloud provider configuration that you define.

Profiles

Set up an initial profile at /etc/salt/cloud.profiles or /etc/salt/cloud.profiles.d/vmware.conf:

```
vmware-centos6.5:
  provider: vcenter01
  clonefrom: test-vm

  ## Optional arguments
  num_cpus: 4
  memory: 8GB
  devices:
    cd:
      CD/DVD drive 1:
        device_type: datastore_iso_file
        iso_path: "[nap004-1] vmimages/tools-isoimages/linux.iso"
      CD/DVD drive 2:
        device_type: client_device
        mode: atapi
        controller: IDE 2
      CD/DVD drive 3:
        device_type: client_device
        mode: passthrough
        controller: IDE 3
    disk:
      Hard disk 1:
        size: 30
      Hard disk 2:
        size: 20
        controller: SCSI controller 2
      Hard disk 3:
        size: 5
        controller: SCSI controller 3
        datastore: smalldiskdatastore
    network:
      Network adapter 1:
        name: 10.20.30-400-Test
        switch_type: standard
        ip: 10.20.30.123
        gateway: [10.20.30.110]
        subnet_mask: 255.255.255.128
        domain: example.com
      Network adapter 2:
        name: 10.30.40-500-Dev-DHCP
        adapter_type: e1000
        switch_type: distributed
        mac: '00:16:3e:e8:19:0f'
      Network adapter 3:
        name: 10.40.50-600-Prod
        adapter_type: vmxnet3
        switch_type: distributed
        ip: 10.40.50.123
        gateway: [10.40.50.110]
        subnet_mask: 255.255.255.128
        domain: example.com
    scsi:
      SCSI controller 1:
        type: lsilogic
```

```
    SCSI controller 2:
      type: lsilogic_sas
      bus_sharing: virtual
    SCSI controller 3:
      type: paravirtual
      bus_sharing: physical
  ide:
    IDE 2
    IDE 3

domain: example.com
dns_servers:
  - 123.127.255.240
  - 123.127.255.241
  - 123.127.255.242

resourcepool: Resources
cluster: Prod

datastore: HUGE-DATASTORE-Cluster
folder: Development
datacenter: DC1
host: c4212n-002.domain.com
template: False
power_on: True
extra_config:
  mem.hotadd: 'yes'
  guestinfo.foo: bar
  guestinfo.domain: foobar.com
  guestinfo.customVariable: customValue
annotation: Created by Salt-Cloud

deploy: True
customization: True
private_key: /root/.ssh/mykey.pem
ssh_username: cloud-user
password: veryVeryBadPassword
minion:
  master: 123.127.193.105

file_map:
  /path/to/local/custom/script: /path/to/remote/script
  /path/to/local/file: /path/to/remote/file
  /srv/salt/yum/epel.repo: /etc/yum.repos.d/epel.repo

hardware_version: 10
image: centos64Guest

#For Windows VM
win_username: Administrator
win_password: administrator
win_organization_name: ABC-Corp
plain_text: True
win_installer: /root/Salt-Minion-2015.8.4-AMD64-Setup.exe
win_user_fullname: Windows User
```

provider Enter the name that was specified when the cloud provider config was created.

clonefrom Enter the name of the VM/template to clone from. If not specified, the VM will be created without cloning.

num_cpus Enter the number of vCPUS that you want the VM/template to have. If not specified, the current VM/template's vCPU count is used.

cores_per_socket Enter the number of cores per vCPU that you want the VM/template to have. If not specified, this will default to 1.

> **Note:** Cores per socket should be less than or equal to the total number of vCPUs assigned to the VM/template.

New in version 2016.11.0.

memory Enter the memory size (in MB or GB) that you want the VM/template to have. If not specified, the current VM/template's memory size is used. Example `memory: 8GB` or `memory: 8192MB`.

devices Enter the device specifications here. Currently, the following devices can be created or reconfigured:

cd Enter the CD/DVD drive specification here. If the CD/DVD drive doesn't exist, it will be created with the specified configuration. If the CD/DVD drive already exists, it will be reconfigured with the specifications. The following options can be specified per CD/DVD drive:

device_type Specify how the CD/DVD drive should be used. Currently supported types are `client_device` and `datastore_iso_file`. Default is `device_type: client_device`

iso_path Enter the path to the iso file present on the datastore only if `device_type: datastore_iso_file`. The syntax to specify this is `iso_path: "[datastoreName] vmimages/tools-isoimages/linux.iso"`. This field is ignored if `device_type: client_device`

mode Enter the mode of connection only if `device_type: client_device`. Currently supported modes are `passthrough` and `atapi`. This field is ignored if `device_type: datastore_iso_file`. Default is `mode: passthrough`

controller Specify the IDE controller label to which this drive should be attached. This should be specified only when creating both the specified IDE controller as well as the CD/DVD drive at the same time.

disk Enter the disk specification here. If the hard disk doesn't exist, it will be created with the provided size. If the hard disk already exists, it will be expanded if the provided size is greater than the current size of the disk.

size Enter the size of disk in GB

thin_provision Specifies whether the disk should be thin provisioned or not. Default is `thin_provision: False`. .. versionadded:: 2016.3.0

eagerly_scrub Specifies whether the disk should be rewrite with zeros during thick provisioning or not. Default is `eagerly_scrub: False`. .. versionadded:: 2018.3.0

controller Specify the SCSI controller label to which this disk should be attached. This should be specified only when creating both the specified SCSI controller as well as the hard disk at the same time.

datastore The name of a valid datastore should you wish the new disk to be in a datastore other than the default for the VM.

network Enter the network adapter specification here. If the network adapter doesn't exist, a new network adapter will be created with the specified network name, type and other configuration. If the network

adapter already exists, it will be reconfigured with the specifications. The following additional options can be specified per network adapter (See example above):

name Enter the network name you want the network adapter to be mapped to.

adapter_type Enter the network adapter type you want to create. Currently supported types are `vmxnet`, `vmxnet2`, `vmxnet3`, `e1000` and `e1000e`. If no type is specified, by default `vmxnet3` will be used.

switch_type Enter the type of switch to use. This decides whether to use a standard switch network or a distributed virtual portgroup. Currently supported types are `standard` for standard portgroups and `distributed` for distributed virtual portgroups.

ip Enter the static IP you want the network adapter to be mapped to. If the network specified is DHCP enabled, you do not have to specify this.

gateway Enter the gateway for the network as a list. If the network specified is DHCP enabled, you do not have to specify this.

subnet_mask Enter the subnet mask for the network. If the network specified is DHCP enabled, you do not have to specify this.

domain Enter the domain to be used with the network adapter. If the network specified is DHCP enabled, you do not have to specify this.

mac Enter the MAC for this network adapter. If not specified an address will be selected automatically.

scsi Enter the SCSI controller specification here. If the SCSI controller doesn't exist, a new SCSI controller will be created of the specified type. If the SCSI controller already exists, it will be reconfigured with the specifications. The following additional options can be specified per SCSI controller:

type Enter the SCSI controller type you want to create. Currently supported types are `lsilogic`, `lsilogic_sas` and `paravirtual`. Type must be specified when creating a new SCSI controller.

bus_sharing Specify this if sharing of virtual disks between virtual machines is desired. The following can be specified:

virtual Virtual disks can be shared between virtual machines on the same server.

physical Virtual disks can be shared between virtual machines on any server.

no Virtual disks cannot be shared between virtual machines.

ide Enter the IDE controller specification here. If the IDE controller doesn't exist, a new IDE controller will be created. If the IDE controller already exists, no further changes to it will me made.

`domain` Enter the global domain name to be used for DNS. If not specified and if the VM name is a FQDN, `domain` is set to the domain from the VM name. Default is `local`.

`dns_servers` Enter the list of DNS servers to use in order of priority.

`resourcepool` Enter the name of the resourcepool to which the new virtual machine should be attached. This determines what compute resources will be available to the clone.

Note:

- For a clone operation from a virtual machine, it will use the same resourcepool as the original virtual machine unless specified.
- For a clone operation from a template to a virtual machine, specifying either this or cluster is required. If both are specified, the resourcepool value will be used.
- For a clone operation to a template, this argument is ignored.

---

cluster Enter the name of the cluster whose resource pool the new virtual machine should be attached to.

Note:

- For a clone operation from a virtual machine, it will use the same cluster's resourcepool as the original virtual machine unless specified.
- For a clone operation from a template to a virtual machine, specifying either this or resourcepool is required. If both are specified, the resourcepool value will be used.
- For a clone operation to a template, this argument is ignored.

datastore Enter the name of the datastore or the datastore cluster where the virtual machine should be located on physical storage. If not specified, the current datastore is used.

Note:

- If you specify a datastore cluster name, DRS Storage recommendation is automatically applied.
- If you specify a datastore name, DRS Storage recommendation is disabled.

folder Enter the name of the folder that will contain the new virtual machine.

Note:

- For a clone operation from a VM/template, the new VM/template will be added to the same folder that the original VM/template belongs to unless specified.
- If both folder and datacenter are specified, the folder value will be used.

datacenter Enter the name of the datacenter that will contain the new virtual machine.

Note:

- For a clone operation from a VM/template, the new VM/template will be added to the same folder that the original VM/template belongs to unless specified.
- If both folder and datacenter are specified, the folder value will be used.

host Enter the name of the target host where the virtual machine should be registered.

If not specified:

Note:

- If resource pool is not specified, current host is used.
- If resource pool is specified, and the target pool represents a stand-alone host, the host is used.
- If resource pool is specified, and the target pool represents a DRS-enabled cluster, a host selected by DRS is used.
- If resource pool is specified and the target pool represents a cluster without DRS enabled, an InvalidArgument exception be thrown.

template Specifies whether the new virtual machine should be marked as a template or not. Default is `template: False`.

power_on Specifies whether the new virtual machine should be powered on or not. If `template: True` is set, this field is ignored. Default is `power_on: True`.

extra_config Specifies the additional configuration information for the virtual machine. This describes a set of modifications to the additional options. If the key is already present, it will be reset with the new value provided. Otherwise, a new option is added. Keys with empty values will be removed.

annotation User-provided description of the virtual machine. This will store a message in the vSphere interface, under the annotations section in the Summary view of the virtual machine.

deploy Specifies if salt should be installed on the newly created VM. Default is `True` so salt will be installed using the bootstrap script. If `template: True` or `power_on: False` is set, this field is ignored and salt will not be installed.

wait_for_ip_timeout When `deploy: True`, this timeout determines the maximum time to wait for VMware tools to be installed on the virtual machine. If this timeout is reached, an attempt to determine the client's IP will be made by resolving the VM's name. By lowering this value a salt bootstrap can be fully automated for systems that are not built with VMware tools. Default is `wait_for_ip_timeout: 1200`.

customization Specify whether the new virtual machine should be customized or not. If `customization: False` is set, the new virtual machine will not be customized. Default is `customization: True`.

private_key Specify the path to the private key to use to be able to ssh to the VM.

ssh_username Specify the username to use in order to ssh to the VM. Default is `root`

password Specify a password to use in order to ssh to the VM. If `private_key` is specified, you do not need to specify this.

minion Specify custom minion configuration you want the salt minion to have. A good example would be to specify the `master` as the IP/DNS name of the master.

file_map Specify file/files you want to copy to the VM before the bootstrap script is run and salt is installed. A good example of using this would be if you need to put custom repo files on the server in case your server will be in a private network and cannot reach external networks.

hardware_version Specify the virtual hardware version for the vm/template that is supported by the host.

image Specify the guest id of the VM. For a full list of supported values see the VMware vSphere documentation:

http://pubs.vmware.com/vsphere-60/topic/com.vmware.wssdk.apiref.doc/vim.vm.GuestOsDescriptor.GuestOsIdentifier.html

Note: For a clone operation, this argument is ignored.

win_username Specify windows vm administrator account.

Note: Windows template should have ``administrator'' account.

win_password Specify windows vm administrator account password.

Note: During network configuration (if network specified), it is used to specify new administrator password for the machine.

win_organization_name

Specify windows vm user's organization. Default organization name is Organization VMware vSphere documentation:

https://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/vim.vm.customization.UserData.html

win_user_fullname

Specify windows vm user's fullname. Default fullname is ``Windows User'' VMware vSphere documentation:

https://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/vim.vm.customization.UserData.html

plain_text Flag to specify whether or not the password is in plain text, rather than encrypted. VMware vSphere documentation:

https://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/vim.vm.customization.Password.html

win_installer Specify windows minion client installer path

Cloning a VM

Cloning VMs/templates is the easiest and the preferred way to work with VMs using the VMware driver.

Note: Cloning operations are unsupported on standalone ESXi hosts, a vCenter server will be required.

Example of a minimal profile:

```
my-minimal-clone:
  provider: vcenter01
  clonefrom: 'test-vm'
```

When cloning a VM, all the profile configuration parameters are optional and the configuration gets inherited from the clone.

Example to add/resize a disk:

```
my-disk-example:
  provider: vcenter01
  clonefrom: 'test-vm'

  devices:
    disk:
      Hard disk 1:
        size: 30
```

Depending on the configuration of the VM that is getting cloned, the disk in the resulting clone will differ.

Note:

- If the VM has no disk named `Hard disk 1' an empty disk with the specified size will be added to the clone.
- If the VM has a disk named `Hard disk 1' and the size specified is larger than the original disk, an empty disk with the specified size will be added to the clone.
- If the VM has a disk named `Hard disk 1' and the size specified is smaller than the original disk, an empty disk with the original size will be added to the clone.

Example to reconfigure the memory and number of vCPUs:

```
my-disk-example:
  provider: vcenter01
  clonefrom: 'test-vm'

  memory: 16GB
  num_cpus: 8
```

Cloning a Template

Cloning a template works similar to cloning a VM except for the fact that a resource pool or cluster must be specified additionally in the profile.

Example of a minimal profile:

```
my-template-clone:
 provider: vcenter01
 clonefrom: 'test-template'
 cluster: 'Prod'
```

Cloning from a Snapshot

New in version 2016.3.5.

Cloning from a snapshot requires that one of the supported options be set in the cloud profile.

Supported options are `createNewChildDiskBacking`, `moveChildMostDiskBacking`, `moveAllDiskBackingsAndAllowSharing` and `moveAllDiskBackingsAndDisallowSharing`.

Example of a minimal profile:

```
my-template-clone:
  provider: vcenter01
  clonefrom: 'salt_vm'
  snapshot:
    disk_move_type: createNewChildDiskBacking
    # these types are also supported
    # disk_move_type: moveChildMostDiskBacking
    # disk_move_type: moveAllDiskBackingsAndAllowSharing
    # disk_move_type: moveAllDiskBackingsAndDisallowSharing
```

Creating a VM

New in version 2016.3.0.

Creating a VM from scratch means that more configuration has to be specified in the profile because there is no place to inherit configuration from.

Note: Unlike most cloud drivers that use prepared images, creating VMs using VMware cloud driver needs an installation method that requires no human interaction. For Example: preseeded ISO, kickstart URL or network PXE boot.

Example of a minimal profile:

```
my-minimal-profile:
  provider: esx01
  datastore: esx01-datastore
  resourcepool: Resources
  folder: vm
```

Note: The example above contains the minimum required configuration needed to create a VM from scratch. The resulting VM will only have 1 VCPU, 32MB of RAM and will not have any storage or networking.

Example of a complete profile:

```
my-complete-example:
  provider: esx01
  datastore: esx01-datastore
  resourcepool: Resources
  folder: vm

  num_cpus: 2
  memory: 8GB

  image: debian7_64Guest

  devices:
    scsi:
      SCSI controller 0:
        type: lsilogic_sas
    ide:
      IDE 0: {}
      IDE 1: {}
    disk:
      Hard disk 0:
        controller: 'SCSI controller 0'
        size: 20
        mode: 'independent_nonpersistent'
    cd:
      CD/DVD drive 0:
        controller: 'IDE 0'
        device_type: datastore_iso_file
        iso_path: '[esx01-datastore] debian-8-with-preseed.iso'
    network:
      Network adapter 0:
        name: 'VM Network'
        swith_type: standard
```

Note: Depending on VMware ESX/ESXi version, an exact match for `image` might not be available. In such cases, the closest match to another `image` should be used. In the example above, a Debian 8 VM is created using the image `debian7_64Guest` which is for a Debian 7 guest.

Specifying disk backing mode

New in version 2016.3.5.

Disk backing mode can now be specified when cloning a VM. This option can be set in the cloud profile as shown in example below:

```
my-vm:
  provider: esx01
  datastore: esx01-datastore
  resourcepool: Resources
  folder: vm


  devices:
    disk:
      Hard disk 1:
        mode: 'independent_nonpersistent'
        size: 42
      Hard disk 2:
        mode: 'independent_nonpersistent'
```

13.7.28 Getting Started With Xen

The Xen cloud driver works with Citrix XenServer.

It can be used with a single XenServer or a XenServer resource pool.

Setup Dependencies

This driver requires a copy of the freely available `XenAPI.py` Python module.

Information about the Xen API Python module in the XenServer SDK can be found at https://xenserver.org/partners/developing-products-for-xenserver.html

Place a copy of this module on your system. For example, it can be placed in the *site packages* location on your system.

The location of *site packages* can be determined by running:

```
python -m site --user-site
```

Provider Configuration

Xen requires login credentials to a XenServer.

Set up the provider cloud configuration file at `/etc/salt/cloud.providers` or `/etc/salt/cloud.providers.d/*.conf`.

```
# /etc/salt/cloud.providers.d/myxen.conf
myxen:
  driver: xen
  url: https://10.0.0.120
  user: root
  password: p@ssw0rd
```

url: The `url` option supports both `http` and `https` uri prefixes.

user: A valid user id to login to the XenServer host.

password: The associated password for the user.

Note: Changed in version 2015.8.0.

The `provider` parameter in cloud provider definitions was renamed to `driver`. This change was made to avoid confusion with the `provider` parameter that is used in cloud profile definitions. Cloud provider definitions now use `driver` to refer to the Salt cloud module that provides the underlying functionality to connect to a cloud host, while cloud profiles continue to use `provider` to refer to provider configurations that you define.

Profile Configuration

Xen profiles require a `provider` and `image`.

provider: This will be the name of your defined provider.

image: The name of the VM template used to clone or copy.

clone: The default behavior is to clone a template or VM. This is very fast, but requires the source template or VM to be in the same storage repository of the new target system. If the source and target are in different storage repositories then you must copy the source and not clone it by setting `clone: False`.

deploy: The provisioning process will attempt to install the Salt minion service on the new target system by default. This will require login credentials for Salt cloud to login via ssh to it. The `user` and `password` options are required. If `deploy` is set to `False` then these options are not needed.

resource_pool: The name of the resource pool used for this profile.

storage_repo: The name of the storage repository for the target system.

ipv4_cidr: If template is Windows, and running guest tools then a static ip address can be set.

ipv4_gw: If template is Windows, and running guest tools then a gateway can be set.

Set up an initial profile at `/etc/salt/cloud.profiles` or in the `/etc/salt/cloud.profiles.d/` directory:

```
# file: /etc/salt/cloud.profiles.d/xenprofiles.conf
sles:
  provider: myxen
  deploy: False
  image: sles12sp2-template

suse:
  user: root
  password: p@ssw0rd
  provider: myxen
  image: opensuseleap42_2-template
  storage_repo: 'Local storage'
  clone: False
  minion:
    master: 10.0.0.20

w2k12:
  provider: myxen
  image: w2k12svr-template
  clone: True
  userdata_file: /srv/salt/win/files/windows-firewall.ps1
  win_installer: /srv/salt/win/files/Salt-Minion-2016.11.3-AMD64-Setup.exe
  win_username: Administrator
  win_password: p@ssw0rd
```

```
  use_winrm: False
  ipv4_cidr: 10.0.0.215/24
  ipv4_gw: 10.0.0.1
  minion:
    master: 10.0.0.21
```

The first example will create a clone of the sles12sp2-template in the same storage repository without deploying the Salt minion.

The second example will make a copy of the image and deploy a new suse VM with the Salt minion installed.

The third example will create a clone of the Windows 2012 template and deploy the Salt minion.

The profile can be used with a salt command:

```
salt-cloud -p suse  xenvm02
```

This will create an salt minion instance named `xenvm02` in Xen. If the command was executed on the salt-master, its Salt key will automatically be signed on the master.

Once the instance has been created with a salt-minion installed, connectivity to it can be verified with Salt:

```
salt xenvm02 test.ping
```

Listing Sizes

Sizes can be obtained using the `--list-sizes` option for the `salt-cloud` command:

```
# salt-cloud --list-sizes myxen
```

Note: Since size information is build in a template this command is not implemented.

Listing Images

Images can be obtained using the `--list-images` option for the `salt-cloud` command:

```
# salt-cloud --list-images myxen
```

This command will return a list of templates with details.

Listing Locations

Locations can be obtained using the `--list-locations` option for the `salt-cloud` command:

```
# salt-cloud --list-locations myxen
```

Returns a list of resource pools.

13.8 Miscellaneous Options

13.8.1 Miscellaneous Salt Cloud Options

This page describes various miscellaneous options available in Salt Cloud

Deploy Script Arguments

Custom deploy scripts are unlikely to need custom arguments to be passed to them, but salt-bootstrap has been extended quite a bit, and this may be necessary. script_args can be specified in either the profile or the map file, to pass arguments to the deploy script:

```
ec2-amazon:
  provider: my-ec2-config
  image: ami-1624987f
  size: t1.micro
  ssh_username: ec2-user
  script: bootstrap-salt
  script_args: -c /tmp/
```

This has also been tested to work with pipes, if needed:

```
script_args: '| head'
```

Selecting the File Transport

By default, Salt Cloud uses SFTP to transfer files to Linux hosts. However, if SFTP is not available, or specific SCP functionality is needed, Salt Cloud can be configured to use SCP instead.

```
file_transport: sftp
file_transport: scp
```

Sync After Install

Salt allows users to create custom modules, grains, and states which can be synchronised to minions to extend Salt with further functionality.

This option will inform Salt Cloud to synchronise your custom modules, grains, states or all these to the minion just after it has been created. For this to happen, the following line needs to be added to the main cloud configuration file:

```
sync_after_install: all
```

The available options for this setting are:

```
modules
grains
states
all
```

Setting Up New Salt Masters

It has become increasingly common for users to set up multi-hierarchal infrastructures using Salt Cloud. This sometimes involves setting up an instance to be a master in addition to a minion. With that in mind, you can now lay down master configuration on a machine by specifying master options in the profile or map file.

```
make_master: True
```

This will cause Salt Cloud to generate master keys for the instance, and tell salt-bootstrap to install the salt-master package, in addition to the salt-minion package.

The default master configuration is usually appropriate for most users, and will not be changed unless specific master configuration has been added to the profile or map:

```
master:
  user: root
  interface: 0.0.0.0
```

Setting Up a Salt Syndic with Salt Cloud

In addition to *setting up new Salt Masters*, *syndics* can also be provisioned using Salt Cloud. In order to set up a Salt Syndic via Salt Cloud, a Salt Master needs to be installed on the new machine and a master configuration file needs to be set up using the `make_master` setting. This setting can be defined either in a profile config file or in a map file:

```
make_master: True
```

To install the Salt Syndic, the only other specification that needs to be configured is the `syndic_master` key to specify the location of the master that the syndic will be reporting to. This modification needs to be placed in the `master` setting, which can be configured either in the profile, provider, or `/etc/salt/cloud` config file:

```
master:
  syndic_master: 123.456.789  # may be either an IP address or a hostname
```

Many other Salt Syndic configuration settings and specifications can be passed through to the new syndic machine via the `master` configuration setting. See the *Salt Syndic* documentation for more information.

SSH Port

By default ssh port is set to port 22. If you want to use a custom port in provider, profile, or map blocks use ssh_port option.

New in version 2015.5.0.

```
ssh_port: 2222
```

SSH Port

By default ssh port is set to port 22. If you want to use a custom port in provider, profile, or map blocks use ssh_port option.

```
ssh_port: 2222
```

Delete SSH Keys

When Salt Cloud deploys an instance, the SSH pub key for the instance is added to the known_hosts file for the user that ran the salt-cloud command. When an instance is deployed, a cloud host generally recycles the IP address for the instance. When Salt Cloud attempts to deploy an instance using a recycled IP address that has previously been accessed from the same machine, the old key in the known_hosts file will cause a conflict.

In order to mitigate this issue, Salt Cloud can be configured to remove old keys from the known_hosts file when destroying the node. In order to do this, the following line needs to be added to the main cloud configuration file:

```
delete_sshkeys: True
```

Keeping /tmp/ Files

When Salt Cloud deploys an instance, it uploads temporary files to /tmp/ for salt-bootstrap to put in place. After the script has run, they are deleted. To keep these files around (mostly for debugging purposes), the --keep-tmp option can be added:

```
salt-cloud -p myprofile mymachine --keep-tmp
```

For those wondering why /tmp/ was used instead of /root/, this had to be done for images which require the use of sudo, and therefore do not allow remote root logins, even for file transfers (which makes /root/ unavailable).

Hide Output From Minion Install

By default Salt Cloud will stream the output from the minion deploy script directly to STDOUT. Although this can been very useful, in certain cases you may wish to switch this off. The following config option is there to enable or disable this output:

```
display_ssh_output: False
```

Connection Timeout

There are several stages when deploying Salt where Salt Cloud needs to wait for something to happen. The VM getting it's IP address, the VM's SSH port is available, etc.

If you find that the Salt Cloud defaults are not enough and your deployment fails because Salt Cloud did not wait log enough, there are some settings you can tweak.

Note

All settings should be provided in lowercase All values should be provided in seconds

You can tweak these settings globally, per cloud provider, or event per profile definition.

wait_for_ip_timeout

The amount of time Salt Cloud should wait for a VM to start and get an IP back from the cloud host. Default: varies by cloud provider (between 5 and 25 minutes)

wait_for_ip_interval

The amount of time Salt Cloud should sleep while querying for the VM's IP. Default: varies by cloud provider (between .5 and 10 seconds)

ssh_connect_timeout

The amount of time Salt Cloud should wait for a successful SSH connection to the VM. Default: varies by cloud provider (between 5 and 15 minutes)

wait_for_passwd_timeout

The amount of time until an ssh connection can be established via password or ssh key. Default: varies by cloud provider (mostly 15 seconds)

wait_for_passwd_maxtries

The number of attempts to connect to the VM until we abandon. Default: 15 attempts

wait_for_fun_timeout

Some cloud drivers check for an available IP or a successful SSH connection using a function, namely, SoftLayer, and SoftLayer-HW. So, the amount of time Salt Cloud should retry such functions before failing. Default: 15 minutes.

wait_for_spot_timeout

The amount of time Salt Cloud should wait before an EC2 Spot instance is available. This setting is only available for the EC2 cloud driver. Default: 10 minutes

Salt Cloud Cache

Salt Cloud can maintain a cache of node data, for supported providers. The following options manage this functionality.

update_cachedir

On supported cloud providers, whether or not to maintain a cache of nodes returned from a --full-query. The data will be stored in `msgpack` format under `<SALT_CACHEDIR>/cloud/active/<DRIVER>/<PROVIDER>/<NODE_NAME>.p`. This setting can be True or False.

diff_cache_events

When the cloud cachedir is being managed, if differences are encountered between the data that is returned live from the cloud host and the data in the cache, fire events which describe the changes. This setting can be True or False.

Some of these events will contain data which describe a node. Because some of the fields returned may contain sensitive data, the `cache_event_strip_fields` configuration option exists to strip those fields from the event return.

```
cache_event_strip_fields:
  - password
  - priv_key
```

The following are events that can be fired based on this data.

salt/cloud/minionid/cache_node_new

A new node was found on the cloud host which was not listed in the cloud cachedir. A dict describing the new node will be contained in the event.

salt/cloud/minionid/cache_node_missing

A node that was previously listed in the cloud cachedir is no longer available on the cloud host.

salt/cloud/minionid/cache_node_diff

One or more pieces of data in the cloud cachedir has changed on the cloud host. A dict containing both the old and the new data will be contained in the event.

SSH Known Hosts

Normally when bootstrapping a VM, salt-cloud will ignore the SSH host key. This is because it does not know what the host key is before starting (because it doesn't exist yet). If strict host key checking is turned on without the key in the `known_hosts` file, then the host will never be available, and cannot be bootstrapped.

If a provider is able to determine the host key before trying to bootstrap it, that provider's driver can add it to the `known_hosts` file, and then turn on strict host key checking. This can be set up in the main cloud configuration file (normally `/etc/salt/cloud`) or in the provider-specific configuration file:

```
known_hosts_file: /path/to/.ssh/known_hosts
```

If this is not set, it will default to `/dev/null`, and strict host key checking will be turned off.

It is highly recommended that this option is *not* set, unless the user has verified that the provider supports this functionality, and that the image being used is capable of providing the necessary information. At this time, only the EC2 driver supports this functionality.

SSH Agent

New in version 2015.5.0.

If the ssh key is not stored on the server salt-cloud is being run on, set ssh_agent, and salt-cloud will use the forwarded ssh-agent to authenticate.

```
ssh_agent: True
```

File Map Upload

New in version 2014.7.0.

The `file_map` option allows an arbitrary group of files to be uploaded to the target system before running the deploy script. This functionality requires a provider uses salt.utils.cloud.bootstrap(), which is currently limited to the ec2, gce, openstack and nova drivers.

The `file_map` can be configured globally in `/etc/salt/cloud`, or in any cloud provider or profile file. For example, to upload an extra package or a custom deploy script, a cloud profile using `file_map` might look like:

```
ubuntu14:
  provider: ec2-config
  image: ami-98aa1cf0
  size: t1.micro
  ssh_username: root
  securitygroup: default
  file_map:
    /local/path/to/custom/script: /remote/path/to/use/custom/script
    /local/path/to/package: /remote/path/to/store/package
```

Running Pre-Flight Commands

New in version 2018.3.0.

To execute specified preflight shell commands on a VM before the deploy script is run, use the `preflight_cmds` option. These must be defined as a list in a cloud configuration file. For example:

```
my-cloud-profile:
  provider: linode-config
  image: Ubuntu 16.04 LTS
  size: Linode 2048
  preflight_cmds:
    - whoami
    - echo 'hello world!'
```

These commands will run in sequence **before** the bootstrap script is executed.

Force Minion Config

New in version 2018.3.0.

The `force_minion_config` option requests the bootstrap process to overwrite an existing minion configuration file and public/private key files. Default: False

This might be important for drivers (such as `saltify`) which are expected to take over a connection from a former salt master.

```
my_saltify_provider:
  driver: saltify
  force_minion_config: true
```

13.9 Troubleshooting Steps

13.9.1 Troubleshooting Salt Cloud

This page describes various steps for troubleshooting problems that may arise while using Salt Cloud.

Virtual Machines Are Created, But Do Not Respond

Are TCP ports 4505 and 4506 open on the master? This is easy to overlook on new masters. Information on how to open firewall ports on various platforms can be found *here*.

Generic Troubleshooting Steps

This section describes a set of instructions that are useful to a large number of situations, and are likely to solve most issues that arise.

Debug Mode

Frequently, running Salt Cloud in debug mode will reveal information about a deployment which would otherwise not be obvious:

```
salt-cloud -p myprofile myinstance -l debug
```

Keep in mind that a number of messages will appear that look at first like errors, but are in fact intended to give developers factual information to assist in debugging. A number of messages that appear will be for cloud providers that you do not have configured; in these cases, the message usually is intended to confirm that they are not configured.

Salt Bootstrap

By default, Salt Cloud uses the Salt Bootstrap script to provision instances:

This script is packaged with Salt Cloud, but may be updated without updating the Salt package:

```
salt-cloud -u
```

The Bootstrap Log

If the default deploy script was used, there should be a file in the `/tmp/` directory called `bootstrap-salt.log`. This file contains the full output from the deployment, including any errors that may have occurred.

Keeping Temp Files

Salt Cloud uploads minion-specific files to instances once they are available via SSH, and then executes a deploy script to put them into the correct place and install Salt. The --keep-tmp option will instruct Salt Cloud not to remove those files when finished with them, so that the user may inspect them for problems:

```
salt-cloud -p myprofile myinstance --keep-tmp
```

By default, Salt Cloud will create a directory on the target instance called /tmp/.saltcloud/. This directory should be owned by the user that is to execute the deploy script, and should have permissions of 0700.

Most cloud hosts are configured to use root as the default initial user for deployment, and as such, this directory and all files in it should be owned by the root user.

The /tmp/.saltcloud/ directory should the following files:

- A deploy.sh script. This script should have permissions of 0755.
- A .pem and .pub key named after the minion. The .pem file should have permissions of 0600. Ensure that the .pem and .pub files have been properly copied to the /etc/salt/pki/minion/ directory.
- A file called minion. This file should have been copied to the /etc/salt/ directory.
- Optionally, a file called grains. This file, if present, should have been copied to the /etc/salt/ directory.

Unprivileged Primary Users

Some cloud hosts, most notably EC2, are configured with a different primary user. Some common examples are ec2-user, ubuntu, fedora, and bitnami. In these cases, the /tmp/.saltcloud/ directory and all files in it should be owned by this user.

Some cloud hosts, such as EC2, are configured to not require these users to provide a password when using the sudo command. Because it is more secure to require sudo users to provide a password, other hosts are configured that way.

If this instance is required to provide a password, it needs to be configured in Salt Cloud. A password for sudo to use may be added to either the provider configuration or the profile configuration:

```
sudo_password: mypassword
```

/tmp/ is Mounted as noexec

It is more secure to mount the /tmp/ directory with a noexec option. This is uncommon on most cloud hosts, but very common in private environments. To see if the /tmp/ directory is mounted this way, run the following command:

```
mount | grep tmp
```

The if the output of this command includes a line that looks like this, then the /tmp/ directory is mounted as noexec:

```
tmpfs on /tmp type tmpfs (rw,noexec)
```

If this is the case, then the deploy_command will need to be changed in order to run the deploy script through the sh command, rather than trying to execute it directly. This may be specified in either the provider or the profile config:

```
deploy_command: sh /tmp/.saltcloud/deploy.sh
```

Please note that by default, Salt Cloud will place its files in a directory called `/tmp/.saltcloud/`. This may be also be changed in the provider or profile configuration:

```
tmp_dir: /tmp/.saltcloud/
```

If this directory is changed, then the `deploy_command` need to be changed in order to reflect the `tmp_dir` configuration.

Executing the Deploy Script Manually

If all of the files needed for deployment were successfully uploaded to the correct locations, and contain the correct permissions and ownerships, the deploy script may be executed manually in order to check for other issues:

```
cd /tmp/.saltcloud/
./deploy.sh
```

13.10 Extending Salt Cloud

13.10.1 Writing Cloud Driver Modules

Salt Cloud runs on a module system similar to the main Salt project. The modules inside saltcloud exist in the `salt/cloud/clouds` directory of the salt source.

There are two basic types of cloud modules. If a cloud host is supported by libcloud, then using it is the fastest route to getting a module written. The Apache Libcloud project is located at:

http://libcloud.apache.org/

Not every cloud host is supported by libcloud. Additionally, not every feature in a supported cloud host is necessarily supported by libcloud. In either of these cases, a module can be created which does not rely on libcloud.

All Driver Modules

The following functions are required by all driver modules, whether or not they are based on libcloud.

The __virtual__() Function

This function determines whether or not to make this cloud module available upon execution. Most often, it uses `get_configured_provider()` to determine if the necessary configuration has been set up. It may also check for necessary imports, to decide whether to load the module. In most cases, it will return a `True` or `False` value. If the name of the driver used does not match the filename, then that name should be returned instead of `True`. An example of this may be seen in the Azure module:

https://github.com/saltstack/salt/tree/develop/salt/cloud/clouds/msazure.py

The get_configured_provider() Function

This function uses `config.is_provider_configured()` to determine wither all required information for this driver has been configured. The last value in the list of required settings should be followed by a comma.

Libcloud Based Modules

Writing a cloud module based on libcloud has two major advantages. First of all, much of the work has already been done by the libcloud project. Second, most of the functions necessary to Salt have already been added to the Salt Cloud project.

The create() Function

The most important function that does need to be manually written is the `create()` function. This is what is used to request a virtual machine to be created by the cloud host, wait for it to become available, and then (optionally) log in and install Salt on it.

A good example to follow for writing a cloud driver module based on libcloud is the module provided for Linode:

https://github.com/saltstack/salt/tree/develop/salt/cloud/clouds/linode.py

The basic flow of a `create()` function is as follows:

- Send a request to the cloud host to create a virtual machine.
- Wait for the virtual machine to become available.
- Generate kwargs to be used to deploy Salt.
- Log into the virtual machine and deploy Salt.
- Return a data structure that describes the newly-created virtual machine.

At various points throughout this function, events may be fired on the Salt event bus. Four of these events, which are described below, are required. Other events may be added by the user, where appropriate.

When the `create()` function is called, it is passed a data structure called `vm_`. This dict contains a composite of information describing the virtual machine to be created. A dict called `__opts__` is also provided by Salt, which contains the options used to run Salt Cloud, as well as a set of configuration and environment variables.

The first thing the `create()` function must do is fire an event stating that it has started the create process. This event is tagged `salt/cloud/<vm name>/creating`. The payload contains the names of the VM, profile, and provider.

A set of kwargs is then usually created, to describe the parameters required by the cloud host to request the virtual machine.

An event is then fired to state that a virtual machine is about to be requested. It is tagged as `salt/cloud/<vm name>/requesting`. The payload contains most or all of the parameters that will be sent to the cloud host. Any private information (such as passwords) should not be sent in the event.

After a request is made, a set of deploy kwargs will be generated. These will be used to install Salt on the target machine. Windows options are supported at this point, and should be generated, even if the cloud host does not currently support Windows. This will save time in the future if the host does eventually decide to support Windows.

An event is then fired to state that the deploy process is about to begin. This event is tagged `salt/cloud/<vm name>/deploying`. The payload for the event will contain a set of deploy kwargs, useful for debugging purposed. Any private data, including passwords and keys (including public keys) should be stripped from the deploy kwargs before the event is fired.

If any Windows options have been passed in, the `salt.utils.cloud.deploy_windows()` function will be called. Otherwise, it will be assumed that the target is a Linux or Unix machine, and the `salt.utils.cloud.deploy_script()` will be called.

Both of these functions will wait for the target machine to become available, then the necessary port to log in, then a successful login that can be used to install Salt. Minion configuration and keys will then be uploaded to a temporary directory on the target by the appropriate function. On a Windows target, the Windows Minion Installer will be run in silent mode. On a Linux/Unix target, a deploy script (`bootstrap-salt.sh`, by default) will be run, which will auto-detect the operating system, and install Salt using its native package manager. These do not need to be handled by the developer in the cloud module.

The `salt.utils.cloud.validate_windows_cred()` function has been extended to take the number of retries and retry_delay parameters in case a specific cloud host has a delay between providing the Windows credentials and the credentials being available for use. In their `create()` function, or as a sub-function called during the creation process, developers should use the `win_deploy_auth_retries` and `win_deploy_auth_retry_delay` parameters from the provider configuration to allow the end-user the ability to customize the number of tries and delay between tries for their particular host.

After the appropriate deploy function completes, a final event is fired which describes the virtual machine that has just been created. This event is tagged `salt/cloud/<vm name>/created`. The payload contains the names of the VM, profile, and provider.

Finally, a dict (queried from the provider) which describes the new virtual machine is returned to the user. Because this data is not fired on the event bus it can, and should, return any passwords that were returned by the cloud host. In some cases (for example, Rackspace), this is the only time that the password can be queried by the user; post-creation queries may not contain password information (depending upon the host).

The libcloudfuncs Functions

A number of other functions are required for all cloud hosts. However, with libcloud-based modules, these are all provided for free by the libcloudfuncs library. The following two lines set up the imports:

```
from salt.cloud.libcloudfuncs import *   # pylint: disable=W0614,W0401
import salt.utils.functools
```

And then a series of declarations will make the necessary functions available within the cloud module.

```
get_size = salt.utils.functools.namespaced_function(get_size, globals())
get_image = salt.utils.functools.namespaced_function(get_image, globals())
avail_locations = salt.utils.functools.namespaced_function(avail_locations, globals())
avail_images = salt.utils.functools.namespaced_function(avail_images, globals())
avail_sizes = salt.utils.functools.namespaced_function(avail_sizes, globals())
script = salt.utils.functools.namespaced_function(script, globals())
destroy = salt.utils.functools.namespaced_function(destroy, globals())
list_nodes = salt.utils.functools.namespaced_function(list_nodes, globals())
list_nodes_full = salt.utils.functools.namespaced_function(list_nodes_full, globals())
list_nodes_select = salt.utils.functools.namespaced_function(list_nodes_select,
↪globals())
show_instance = salt.utils.functools.namespaced_function(show_instance, globals())
```

If necessary, these functions may be replaced by removing the appropriate declaration line, and then adding the function as normal.

These functions are required for all cloud modules, and are described in detail in the next section.

Non-Libcloud Based Modules

In some cases, using libcloud is not an option. This may be because libcloud has not yet included the necessary driver itself, or it may be that the driver that is included with libcloud does not contain all of the necessary features required by the developer. When this is the case, some or all of the functions in `libcloudfuncs` may be replaced. If they are all replaced, the libcloud imports should be absent from the Salt Cloud module.

A good example of a non-libcloud driver is the DigitalOcean driver:

https://github.com/saltstack/salt/tree/develop/salt/cloud/clouds/digitalocean.py

The `create()` Function

The `create()` function must be created as described in the libcloud-based module documentation.

The get_size() Function

This function is only necessary for libcloud-based modules, and does not need to exist otherwise.

The get_image() Function

This function is only necessary for libcloud-based modules, and does not need to exist otherwise.

The avail_locations() Function

This function returns a list of locations available, if the cloud host uses multiple data centers. It is not necessary if the cloud host uses only one data center. It is normally called using the `--list-locations` option.

```
salt-cloud --list-locations my-cloud-provider
```

The avail_images() Function

This function returns a list of images available for this cloud provider. There are not currently any known cloud providers that do not provide this functionality, though they may refer to images by a different name (for example, ``templates''). It is normally called using the `--list-images` option.

```
salt-cloud --list-images my-cloud-provider
```

The avail_sizes() Function

This function returns a list of sizes available for this cloud provider. Generally, this refers to a combination of RAM, CPU, and/or disk space. This functionality may not be present on some cloud providers. For example, the Parallels module breaks down RAM, CPU, and disk space into separate options, whereas in other providers, these options are baked into the image. It is normally called using the `--list-sizes` option.

```
salt-cloud --list-sizes my-cloud-provider
```

The script() Function

This function builds the deploy script to be used on the remote machine. It is likely to be moved into the `salt.utils.cloud` library in the near future, as it is very generic and can usually be copied wholesale from another module. An excellent example is in the Azure driver.

The destroy() Function

This function irreversibly destroys a virtual machine on the cloud provider. Before doing so, it should fire an event on the Salt event bus. The tag for this event is `salt/cloud/<vm name>/destroying`. Once the virtual machine has been destroyed, another event is fired. The tag for that event is `salt/cloud/<vm name>/destroyed`.

This function is normally called with the `-d` options:

```
salt-cloud -d myinstance
```

The list_nodes() Function

This function returns a list of nodes available on this cloud provider, using the following fields:

- id (str)
- image (str)
- size (str)
- state (str)
- private_ips (list)
- public_ips (list)

No other fields should be returned in this function, and all of these fields should be returned, even if empty. The private_ips and public_ips fields should always be of a list type, even if empty, and the other fields should always be of a str type. This function is normally called with the `-Q` option:

```
salt-cloud -Q
```

The list_nodes_full() Function

All information available about all nodes should be returned in this function. The fields in the list_nodes() function should also be returned, even if they would not normally be provided by the cloud provider. This is because some functions both within Salt and 3rd party will break if an expected field is not present. This function is normally called with the `-F` option:

```
salt-cloud -F
```

The list_nodes_select() Function

This function returns only the fields specified in the `query.selection` option in `/etc/salt/cloud`. Because this function is so generic, all of the heavy lifting has been moved into the `salt.utils.cloud` library.

A function to call `list_nodes_select()` still needs to be present. In general, the following code can be used as-is:

```
def list_nodes_select(call=None):
    '''
    Return a list of the VMs that are on the provider, with select fields
    '''
    return salt.utils.cloud.list_nodes_select(
        list_nodes_full('function'), __opts__['query.selection'], call,
    )
```

However, depending on the cloud provider, additional variables may be required. For instance, some modules use a `conn` object, or may need to pass other options into `list_nodes_full()`. In this case, be sure to update the function appropriately:

```
def list_nodes_select(conn=None, call=None):
    '''
    Return a list of the VMs that are on the provider, with select fields
    '''
    if not conn:
        conn = get_conn()   # pylint: disable=E0602

    return salt.utils.cloud.list_nodes_select(
        list_nodes_full(conn, 'function'),
        __opts__['query.selection'],
        call,
    )
```

This function is normally called with the `-S` option:

```
salt-cloud -S
```

The show_instance() Function

This function is used to display all of the information about a single node that is available from the cloud provider. The simplest way to provide this is usually to call `list_nodes_full()`, and return just the data for the requested node. It is normally called as an action:

```
salt-cloud -a show_instance myinstance
```

Actions and Functions

Extra functionality may be added to a cloud provider in the form of an `--action` or a `--function`. Actions are performed against a cloud instance/virtual machine, and functions are performed against a cloud provider.

Actions

Actions are calls that are performed against a specific instance or virtual machine. The `show_instance` action should be available in all cloud modules. Actions are normally called with the `-a` option:

```
salt-cloud -a show_instance myinstance
```

Actions must accept a `name` as a first argument, may optionally support any number of kwargs as appropriate, and must accept an argument of `call`, with a default of `None`.

Before performing any other work, an action should normally verify that it has been called correctly. It may then perform the desired feature, and return useful information to the user. A basic action looks like:

```
def show_instance(name, call=None):
'''
Show the details from EC2 concerning an AMI
'''
if call != 'action':
    raise SaltCloudSystemExit(
        'The show_instance action must be called with -a or --action.'
    )

return _get_node(name)
```

Please note that generic kwargs, if used, are passed through to actions as `kwargs` and not `**kwargs`. An example of this is seen in the Functions section.

Functions

Functions are called that are performed against a specific cloud provider. An optional function that is often useful is `show_image`, which describes an image in detail. Functions are normally called with the `-f` option:

```
salt-cloud -f show_image my-cloud-provider image='Ubuntu 13.10 64-bit'
```

A function may accept any number of kwargs as appropriate, and must accept an argument of `call` with a default of `None`.

Before performing any other work, a function should normally verify that it has been called correctly. It may then perform the desired feature, and return useful information to the user. A basic function looks like:

```
def show_image(kwargs, call=None):
    '''
    Show the details from EC2 concerning an AMI
    '''
    if call != 'function':
        raise SaltCloudSystemExit(
            'The show_image action must be called with -f or --function.'
        )

    params = {'ImageId.1': kwargs['image'],
              'Action': 'DescribeImages'}
    result = query(params)
    log.info(result)

    return result
```

Take note that generic kwargs are passed through to functions as `kwargs` and not `**kwargs`.

13.10.2 Cloud deployment scripts

Salt Cloud works primarily by executing a script on the virtual machines as soon as they become available. The script that is executed is referenced in the cloud profile as the `script`. In older versions, this was the `os` argument. This was changed in 0.8.2.

A number of legacy scripts exist in the deploy directory in the saltcloud source tree. The preferred method is currently to use the salt-bootstrap script. A stable version is included with each release tarball starting with 0.8.4.

The most updated version can be found at:

https://github.com/saltstack/salt-bootstrap

Note that, somewhat counter-intuitively, this script is referenced as `bootstrap-salt` in the configuration.

You can specify a deploy script in the cloud configuration file (`/etc/salt/cloud` by default):

```
script: bootstrap-salt
```

Or in a provider:

```
my-provider:
  # snip...
  script: bootstrap-salt
```

Or in a profile:

```
my-profile:
  provider: my-provider
  # snip...
  script: bootstrap-salt
```

If you do not specify a script argument in your cloud configuration file, provider configuration or profile configuration, the ``bootstrap-salt'' script will be used by default.

Other Generic Deploy Scripts

If you want to be assured of always using the latest Salt Bootstrap script, there are a few generic templates available in the deploy directory of your saltcloud source tree:

```
curl-bootstrap
curl-bootstrap-git
python-bootstrap
wget-bootstrap
wget-bootstrap-git
```

These are example scripts which were designed to be customized, adapted, and refit to meet your needs. One important use of them is to pass options to the salt-bootstrap script, such as updating to specific git tags.

Custom Deploy Scripts

If the Salt Bootstrap script does not meet your needs, you may write your own. The script should be written in shell and is a Jinja template. Deploy scripts need to execute a number of functions to do a complete salt setup. These functions include:

1. Install the salt minion. If this can be done via system packages this method is HIGHLY preferred.
2. Add the salt minion keys before the minion is started for the first time. The minion keys are available as strings that can be copied into place in the Jinja template under the dict named ``vm''.
3. Start the salt-minion daemon and enable it at startup time.
4. Set up the minion configuration file from the ``minion'' data available in the Jinja template.

A good, well commented example of this process is the Fedora deployment script:

https://github.com/saltstack/salt/blob/develop/salt/cloud/deploy/Fedora.sh

A number of legacy deploy scripts are included with the release tarball. None of them are as functional or complete as Salt Bootstrap, and are still included for academic purposes.

Custom deploy scripts are picked up from `/etc/salt/cloud.deploy.d` by default, but you can change the location of deploy scripts with the cloud configuration `deploy_scripts_search_path`. Additionally, if your deploy script has the extension `.sh`, you can leave out the extension in your configuration.

For example, if your custom deploy script is located in `/etc/salt/cloud.deploy.d/my_deploy.sh`, you could specify it in a cloud profile like this:

```
my-profile:
  provider: my-provider
  # snip...
  script: my_deploy
```

You're also free to use the full path to the script if you like. Using full paths, your script doesn't have to live inside `/etc/salt/cloud.deploy.d` or whatever you've configured with `deploy_scripts_search_path`.

Post-Deploy Commands

Once a minion has been deployed, it has the option to run a salt command. Normally, this would be the `state.apply`, which would finish provisioning the VM. Another common option (for testing) is to use *test.ping*. This is configured in the main cloud config file:

```
start_action: state.apply
```

This is currently considered to be experimental functionality, and may not work well with all cloud hosts. If you experience problems with Salt Cloud hanging after Salt is deployed, consider using Startup States instead:

http://docs.saltstack.com/ref/states/startup.html

Skipping the Deploy Script

For whatever reason, you may want to skip the deploy script altogether. This results in a VM being spun up much faster, with absolutely no configuration. This can be set from the command line:

```
salt-cloud --no-deploy -p micro_aws my_instance
```

Or it can be set from the main cloud config file:

```
deploy: False
```

Or it can be set from the provider's configuration:

```
RACKSPACE.user: example_user
RACKSPACE.apikey: 123984bjjas87034
RACKSPACE.deploy: False
```

Or even on the VM's profile settings:

```
ubuntu_aws:
  provider: my-ec2-config
  image: ami-7e2da54e
  size: t1.micro
  deploy: False
```

The default for deploy is True.

In the profile, you may also set the script option to `None`:

```
script: None
```

This is the slowest option, since it still uploads the None deploy script and executes it.

Updating Salt Bootstrap

Salt Bootstrap can be updated automatically with `salt-cloud`:

```
salt-cloud -u
salt-cloud --update-bootstrap
```

Bear in mind that this updates to the latest **stable** version from:

https://bootstrap.saltstack.com/stable/bootstrap-salt.sh

To update Salt Bootstrap script to the **develop** version, run the following command on the Salt minion host with `salt-cloud` installed:

```
salt-call config.gather_bootstrap_script 'https://bootstrap.saltstack.com/develop/
↪bootstrap-salt.sh'
```

Or just download the file manually:

```
curl -L 'https://bootstrap.saltstack.com/develop' > /etc/salt/cloud.deploy.d/bootstrap-
↪salt.sh
```

Keeping /tmp/ Files

When Salt Cloud deploys an instance, it uploads temporary files to /tmp/ for salt-bootstrap to put in place. After the script has run, they are deleted. To keep these files around (mostly for debugging purposes), the --keep-tmp option can be added:

```
salt-cloud -p myprofile mymachine --keep-tmp
```

For those wondering why /tmp/ was used instead of /root/, this had to be done for images which require the use of sudo, and therefore do not allow remote root logins, even for file transfers (which makes /root/ unavailable).

Deploy Script Arguments

Custom deploy scripts are unlikely to need custom arguments to be passed to them, but salt-bootstrap has been extended quite a bit, and this may be necessary. script_args can be specified in either the profile or the map file, to pass arguments to the deploy script:

```
aws-amazon:
  provider: my-ec2-config
  image: ami-1624987f
  size: t1.micro
  ssh_username: ec2-user
  script: bootstrap-salt
  script_args: -c /tmp/
```

This has also been tested to work with pipes, if needed:

```
script_args: '| head'
```

13.11 Using Salt Cloud from Salt

13.11.1 Using the Salt Modules for Cloud

In addition to the `salt-cloud` command, Salt Cloud can be called from Salt, in a variety of different ways. Most users will be interested in either the execution module or the state module, but it is also possible to call Salt Cloud as a runner.

Because the actual work will be performed on a remote minion, the normal Salt Cloud configuration must exist on any target minion that needs to execute a Salt Cloud command. Because Salt Cloud now supports breaking out configuration into individual files, the configuration is easily managed using Salt's own `file.managed` state function. For example, the following directories allow this configuration to be managed easily:

```
/etc/salt/cloud.providers.d/
/etc/salt/cloud.profiles.d/
```

Minion Keys

Keep in mind that when creating minions, Salt Cloud will create public and private minion keys, upload them to the minion, and place the public key on the machine that created the minion. It will *not* attempt to place any public minion keys on the master, unless the minion which was used to create the instance is also the Salt Master. This is because granting arbitrary minions access to modify keys on the master is a serious security risk, and must be avoided.

Execution Module

The `cloud` module is available to use from the command line. At the moment, almost every standard Salt Cloud feature is available to use. The following commands are available:

list_images

This command is designed to show images that are available to be used to create an instance using Salt Cloud. In general they are used in the creation of profiles, but may also be used to create an instance directly (see below). Listing images requires a provider to be configured, and specified:

```
salt myminion cloud.list_images my-cloud-provider
```

list_sizes

This command is designed to show sizes that are available to be used to create an instance using Salt Cloud. In general they are used in the creation of profiles, but may also be used to create an instance directly (see below). This command is not available for all cloud providers; see the provider-specific documentation for details. Listing sizes requires a provider to be configured, and specified:

```
salt myminion cloud.list_sizes my-cloud-provider
```

list_locations

This command is designed to show locations that are available to be used to create an instance using Salt Cloud. In general they are used in the creation of profiles, but may also be used to create an instance directly (see below). This command is not available for all cloud providers; see the provider-specific documentation for details. Listing locations requires a provider to be configured, and specified:

```
salt myminion cloud.list_locations my-cloud-provider
```

query

This command is used to query all configured cloud providers, and display all instances associated with those accounts. By default, it will run a standard query, returning the following fields:

id The name or ID of the instance, as used by the cloud provider.

image The disk image that was used to create this instance.

private_ips Any public IP addresses currently assigned to this instance.

public_ips Any private IP addresses currently assigned to this instance.

size The size of the instance; can refer to RAM, CPU(s), disk space, etc., depending on the cloud provider.

state The running state of the instance; for example, `running`, `stopped`, `pending`, etc. This state is dependent upon the provider.

This command may also be used to perform a full query or a select query, as described below. The following usages are available:

```
salt myminion cloud.query
salt myminion cloud.query list_nodes
salt myminion cloud.query list_nodes_full
```

full_query

This command behaves like the `query` command, but lists all information concerning each instance as provided by the cloud provider, in addition to the fields returned by the `query` command.

```
salt myminion cloud.full_query
```

select_query

This command behaves like the `query` command, but only returned select fields as defined in the `/etc/salt/cloud` configuration file. A sample configuration for this section of the file might look like:

```
query.selection:
  - id
  - key_name
```

This configuration would only return the `id` and `key_name` fields, for those cloud providers that support those two fields. This would be called using the following command:

```
salt myminion cloud.select_query
```

profile

This command is used to create an instance using a profile that is configured on the target minion. Please note that the profile must be configured before this command can be used with it.

```
salt myminion cloud.profile ec2-centos64-x64 my-new-instance
```

Please note that the execution module does *not* run in parallel mode. Using multiple minions to create instances can effectively perform parallel instance creation.

create

This command is similar to the `profile` command, in that it is used to create a new instance. However, it does not require a profile to be pre-configured. Instead, all of the options that are normally configured in a profile are passed directly to Salt Cloud to create the instance:

```
salt myminion cloud.create my-ec2-config my-new-instance \
    image=ami-1624987f size='t1.micro' ssh_username=ec2-user \
    securitygroup=default delvol_on_destroy=True
```

Please note that the execution module does *not* run in parallel mode. Using multiple minions to create instances can effectively perform parallel instance creation.

destroy

This command is used to destroy an instance or instances. This command will search all configured providers and remove any instance(s) which matches the name(s) passed in here. The results of this command are *non-reversable* and should be used with caution.

```
salt myminion cloud.destroy myinstance
salt myminion cloud.destroy myinstance1,myinstance2
```

action

This command implements both the `action` and the `function` commands used in the standard `salt-cloud` command. If one of the standard `action` commands is used, an instance name must be provided. If one of the standard `function` commands is used, a provider configuration must be named.

```
salt myminion cloud.action start instance=myinstance
salt myminion cloud.action show_image provider=my-ec2-config \
    image=ami-1624987f
```

The actions available are largely dependent upon the module for the specific cloud provider. The following actions are available for all cloud providers:

`list_nodes` This is a direct call to the `query` function as described above, but is only performed against a single cloud provider. A provider configuration must be included.

list_nodes_select This is a direct call to the full_query function as described above, but is only performed against a single cloud provider. A provider configuration must be included.

list_nodes_select This is a direct call to the select_query function as described above, but is only performed against a single cloud provider. A provider configuration must be included.

show_instance This is a thin wrapper around list_nodes, which returns the full information about a single instance. An instance name must be provided.

State Module

A subset of the execution module is available through the cloud state module. Not all functions are currently included, because there is currently insufficient code for them to perform statefully. For example, a command to create an instance may be issued with a series of options, but those options cannot currently be statefully managed. Additional states to manage these options will be released at a later time.

cloud.present

This state will ensure that an instance is present inside a particular cloud provider. Any option that is normally specified in the cloud.create execution module and function may be declared here, but only the actual presence of the instance will be managed statefully.

```
my-instance-name:
  cloud.present:
    - provider: my-ec2-config
    - image: ami-1624987f
    - size: 't1.micro'
    - ssh_username: ec2-user
    - securitygroup: default
    - delvol_on_destroy: True
```

cloud.profile

This state will ensure that an instance is present inside a particular cloud provider. This function calls the cloud.profile execution module and function, but as with cloud.present, only the actual presence of the instance will be managed statefully.

```
my-instance-name:
  cloud.profile:
    - profile: ec2-centos64-x64
```

cloud.absent

This state will ensure that an instance (identified by name) does not exist in any of the cloud providers configured on the target minion. Please note that this state is *non-reversable* and may be considered especially destructive when issued as a cloud state.

```
my-instance-name:
  cloud.absent
```

Runner Module

The `cloud` runner module is executed on the master, and performs actions using the configuration and Salt modules on the master itself. This means that any public minion keys will also be properly accepted by the master.

Using the functions in the runner module is no different than using those in the execution module, outside of the behavior described in the above paragraph. The following functions are available inside the runner:

- list_images
- list_sizes
- list_locations
- query
- full_query
- select_query
- profile
- destroy
- action

Outside of the standard usage of `salt-run` itself, commands are executed as usual:

```
salt-run cloud.profile ec2-centos64-x86_64 my-instance-name
```

CloudClient

The execution, state, and runner modules ultimately all use the CloudClient library that ships with Salt. To use the CloudClient library locally (either on the master or a minion), create a client object and issue a command against it:

```
import salt.cloud
import pprint
client = salt.cloud.CloudClient('/etc/salt/cloud')
nodes = client.query()
pprint.pprint(nodes)
```

Reactor

Examples of using the reactor with Salt Cloud are available in the ec2-autoscale-reactor and salt-cloud-reactor formulas.

13.12 Feature Comparison

13.12.1 Feature Matrix

A number of features are available in most cloud hosts, but not all are available everywhere. This may be because the feature isn't supported by the cloud host itself, or it may only be that the feature has not yet been added to Salt Cloud. In a handful of cases, it is because the feature does not make sense for a particular cloud provider (Saltify, for instance).

This matrix shows which features are available in which cloud hosts, as far as Salt Cloud is concerned. This is not a comprehensive list of all features available in all cloud hosts, and should not be used to make business decisions concerning choosing a cloud host. In most cases, adding support for a feature to Salt Cloud requires only a little effort.

Legacy Drivers

Both AWS and Rackspace are listed as ``Legacy''. This is because those drivers have been replaced by other drivers, which are generally the preferred method for working with those hosts.

The EC2 driver should be used instead of the AWS driver, when possible. The OpenStack driver should be used instead of the Rackspace driver, unless the user is dealing with instances in ``the old cloud'' in Rackspace.

Note for Developers

When adding new features to a particular cloud host, please make sure to add the feature to this table. Additionally, if you notice a feature that is not properly listed here, pull requests to fix them is appreciated.

Standard Features

These are features that are available for almost every cloud host.

	AWS (Legacy)	CloudStack	Digital Ocean	EC2	GoGrid	JoyEnt	Linode	OpenStack	Parallels	Rackspace (Legacy)	Saltify	Vagrant	Softlayer	Softlayer Hardware	Aliyun
Query	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[1]	[1]	Yes	Yes	Yes
Full Query	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[1]	[1]	Yes	Yes	Yes
Selective Query	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[1]	[1]	Yes	Yes	Yes
List Sizes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[2]	[2]	Yes	Yes	Yes
List Images	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
List Locations	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[2]	[2]	Yes	Yes	Yes
create	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[1]	Yes	Yes	Yes
destroy	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	[1]	[1]	Yes	Yes	Yes

[1] Yes, if salt-api is enabled.

[2] Always returns *{}*.

Actions

These are features that are performed on a specific instance, and require an instance name to be passed in. For example:

```
# salt-cloud -a attach_volume ami.example.com
```

Actions	AWS (Legacy)	CloudStack	Digital Ocean	EC2	GoGrid	JoyEnt	Linode	OpenStack	Parallels	Rackspace (Legacy)	

Saltify& Vagrant

SoftlayerSoftlayer HardwareAliyun

attach_volume Yes

create_attach_volumes Yes Yes

del_tags Yes Yes

delvol_on_destroy Yes

detach_volume Yes

disable_term_protect Yes Yes

enable_term_protect Yes Yes

get_tags Yes Yes

keepvol_on_destroy Yes

list_keypairs Yes

rename Yes Yes

set_tags Yes Yes

show_delvol_on_destroy Yes

show_instance Yes Yes Yes Yes Yes Yes Yes

show_term_protect Yes

start Yes Yes Yes Yes Yes Yes

stop Yes Yes Yes Yes Yes Yes

take_action Yes

Functions

These are features that are performed against a specific cloud provider, and require the name of the provider to be passed in. For example: