MIRANTIS

# Configuration Management with Salt

training.mirantis.com

# Objectives

- Define Salt and its terminologies
- Demonstrate knowledge on managing minions with execution modules
- Demonstrate knowledge on configuration management using state modules

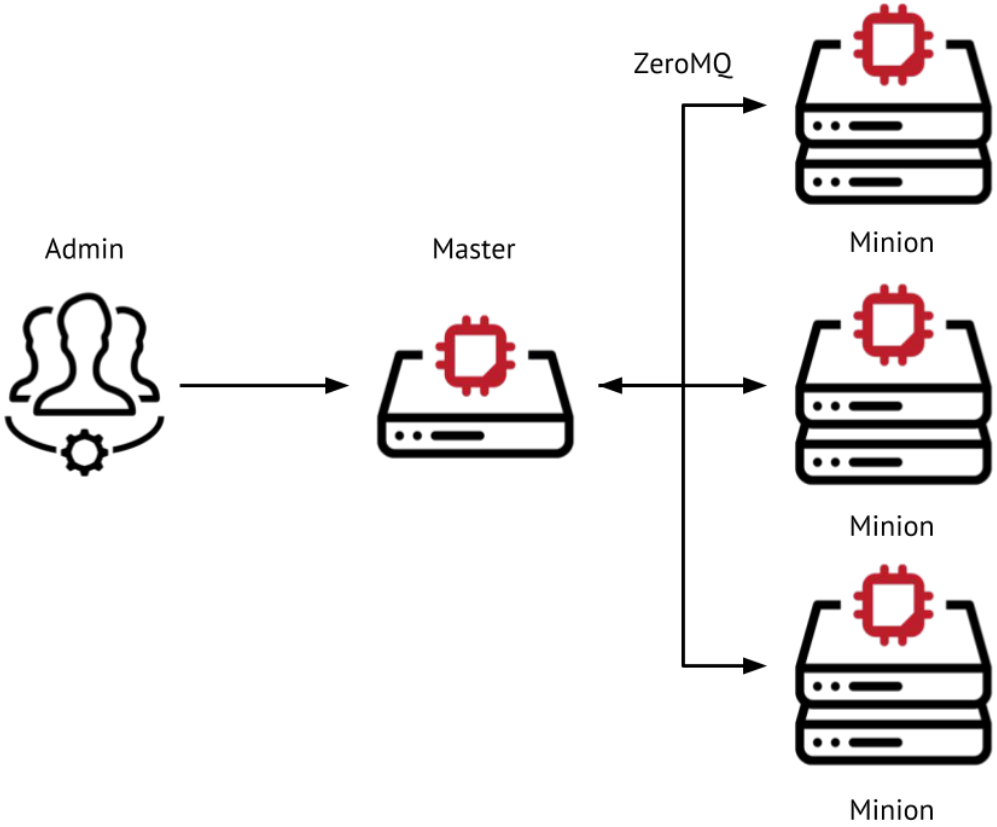
# Introduction to Salt

Section Subtitle

# Why Salt?

- **Remote execution tool**
- **Configuration management tool**
- **Orchestration tool**
- Written in Python
- Apache 2.0 license with diverse contributors
- Secure
- Communication over scalable high performance message bus (ZeroMQ) or SSH

# Salt Architecture: master-minion



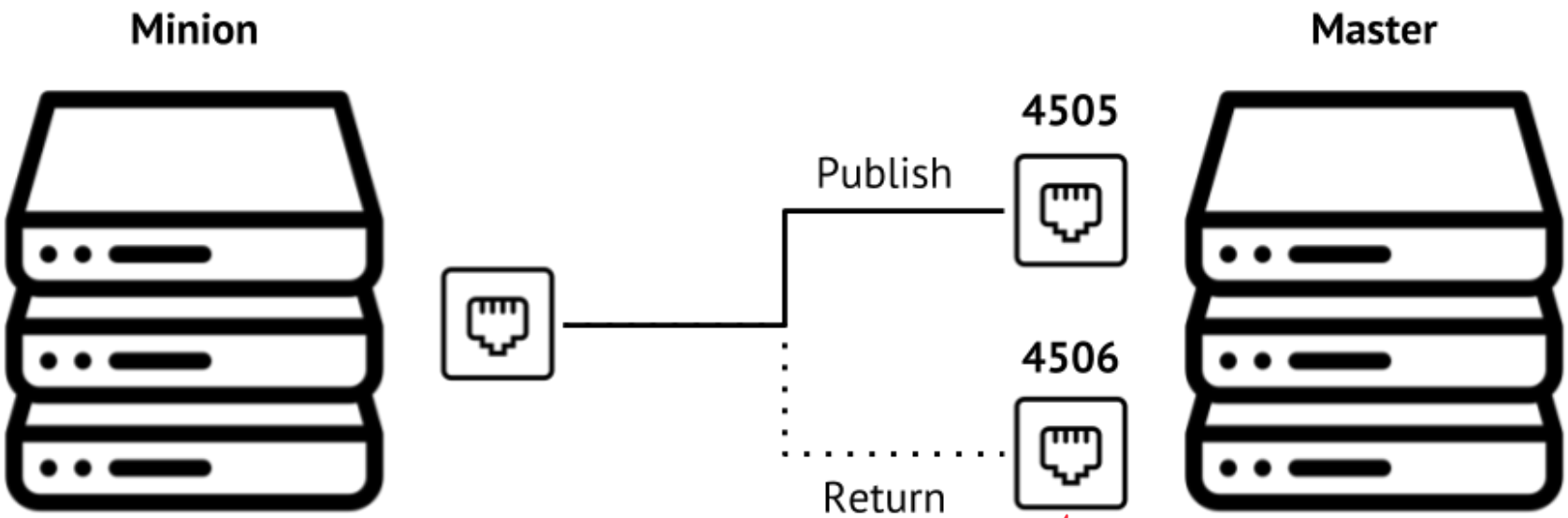

Minion => computer control box

# ZeroMQ

- Fundamental underlying component of Salt
- Socket based programming **library** written in C++
- Library bindings in many languages - incl Python
- Multicore support
- Provides high-speed asynchronous communication between connected systems

ZeroMQ not AMQP

# Salt Architecture: TCP ports




will connect to master Node

Response result

# Salt Terminologies

- Salt master
- Salt minion
- Execution modules
- States
- State modules
- Formulas
- Grains
- Pillar
- Syndic

# Salt master

- Server running ‘salt-master’ service (daemon)
- Authenticates minions
- Sends commands to minions

# Salt minion

- Server running ‘salt-minion’ service (daemon)
- Initiates handshake and key authentication process with master
- Receives and executes commands
- Reports status to master

# Execution modules

- Python modules with functions executed on minions
- Platform independent
- Target minion, module, function, arguments required in command line

# Execution module: Example

```
salt ‘minion1’ network.interface eth0
salt ‘minion1’ cmd.run “ip a show eth0”

salt ‘minion1’ user.add roger
salt ‘minion1’ cmd.run “useradd roger”
```

# Salt states

- Set of configuration management tasks
- Describes end-goal of systems state
  - Declarative
  - Idempotent As many time as you want
  - Platform independent
- Defined in **S**a**L**t **S**tate File (SLS)
  - Infrastructure as Code

# State modules

- Python modules with functions to enforce state on minions
- They call functions from execution modules internally
- Module, function, arguments required in state file
- Target minion and state file required in command line

# Salt states: Example

```
# /srv/salt/users/roger.sls

user_roger:
  user.present:
    - name: roger
    - uid: 1022
    - home: /home/roger
```

platform independant

# Salt states: Example

```
salt ‘minion1’ state.sls users.roger
VS.
salt ‘minion1’ user.add roger
salt ‘minion1’ cmd.run “useradd roger”
```

System specific

# Salt Formula

- Formulas are pre-written Salt States files
- Published on github and delivered as Ubuntu binary packages
- Can be dynamically parameterized with Salt grains and pillars
- Deliver core MCP functionality

# Salt grains

- Static information about minions
  - IP address, OS type, memory, etc.
- Pre-defined ‘core’ grains
- User-defined ‘custom’ grains

# Salt grains: Example

```
# minion /etc/salt/grains
roles:
  - webserver
  - memcache
deployment: datacenter4
cabinet: 13
shelf: 14
```

# Salt pillars

- Global value that can be assigned to minion(s)
- Defined on the master
- Data encrypted on per-minion basis
- Data only accessible to targeted minions only
- Ideal for sensitive data

# Salt pillars: Example

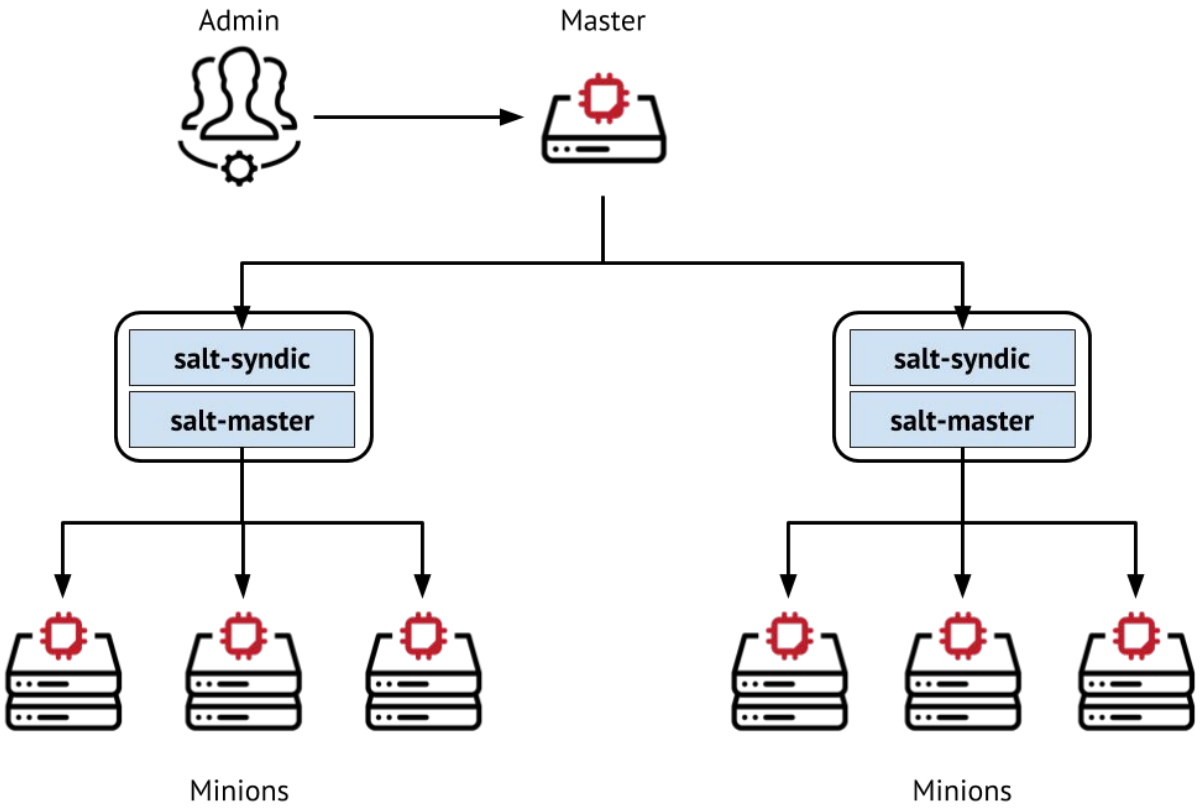
```
# /srv/pillar/users/uid/init.sls

user:
  admin: 1011
  roger: 1022
  frank: 1034
  alice: 1045
```

# Syndic

- Service that acts as proxy in multi-master setup
- Handles requests from upper-level master
- Runs alongside salt-master service

# Syndic topology example

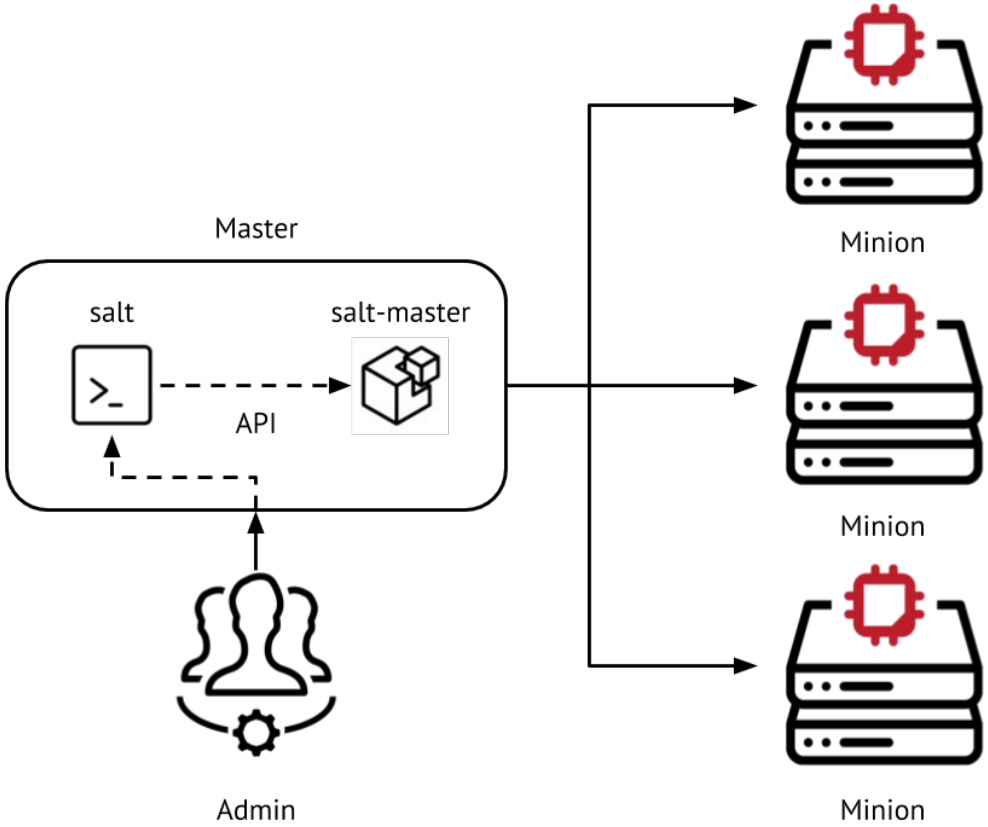


# System Management with execution modules

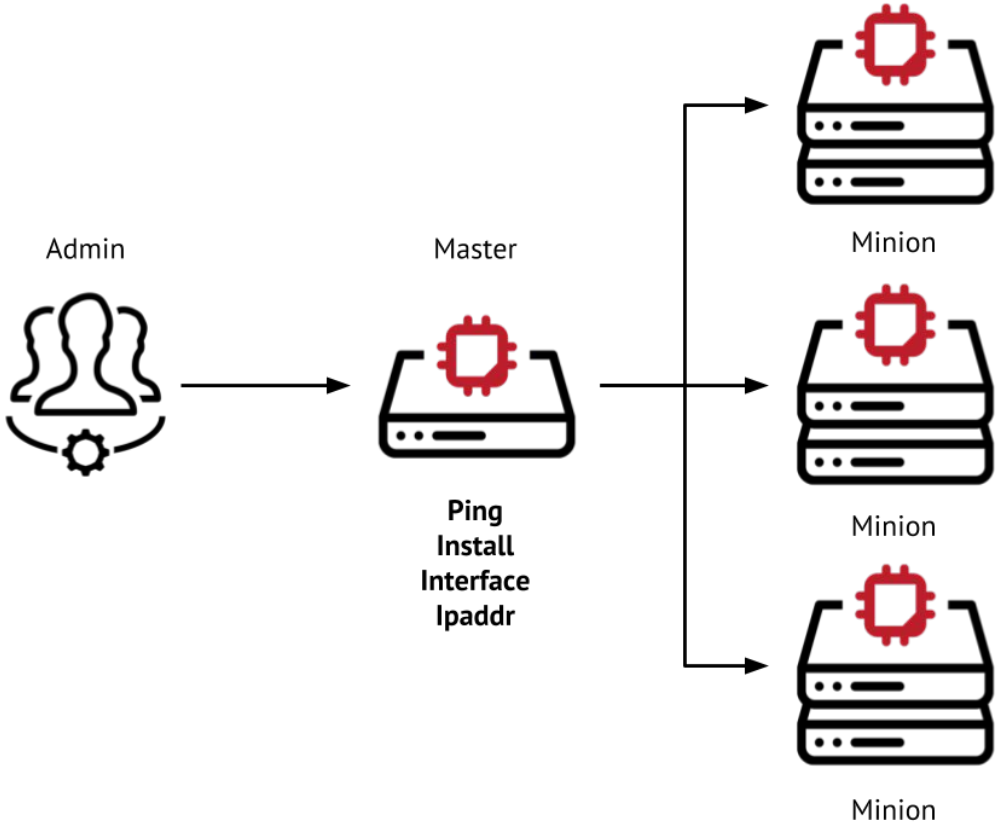
# Salt command line client

- Resides on the master node
- Used to send remote execution commands to minions
- Used to collect information about minions

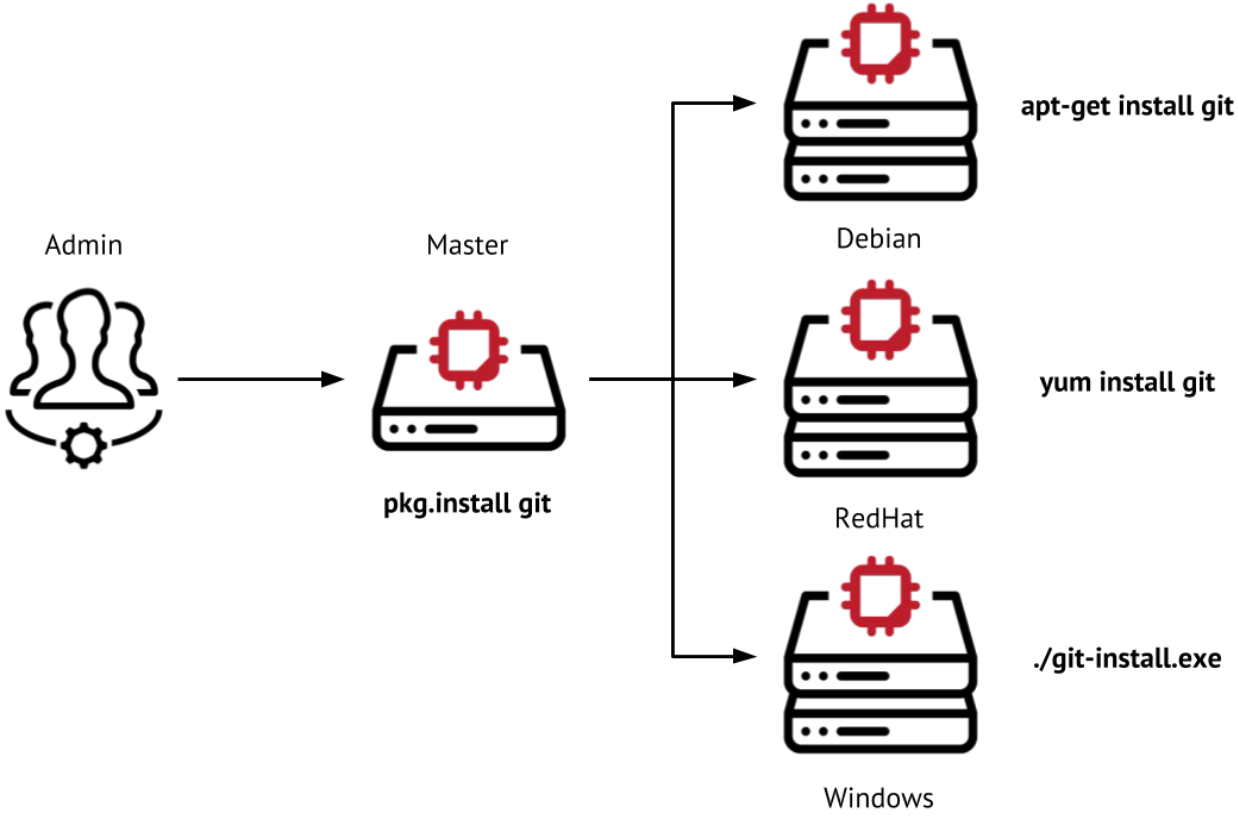
# Salt command line client



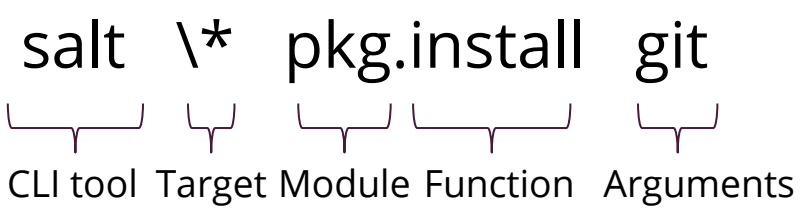
# Execute a module



# Platform independent



# Salt Command Line




- CLI tool - Used to interact with the salt-master
- Target - Specify group of minions that should execute the command
- Module - The Python module that contains functions
- Function - The actual command from the module to run
- Arguments - Argument to pass-in to function

# More on target

- Target options
  - Globs
  - Grains
  - Regex
  - List
  - Pillars
  - Compound
- Publication of data still occurs on all minions

# Targeting examples

```
Glob targeting
# salt ‘*db*’ test.ping

Regex (Regular expression) targeting
# salt ‘example-db-[0-9]+’ disk.usage

Grain targeting
# salt -G ‘os:Ubuntu’ status.uptime
```

# Execution module: Grains

```
Set grains key:value on matched minions
# salt ‘*db*’ grains.setval location west

Ping all minions in the “west” location
# salt -G ‘location:west’ test.ping

Run a shell command
# salt -G ‘os:Ubuntu’ cmd.run ‘ls -l /etc’
```

# Execution module: Pillars

```
Refresh pillar data on minions
# salt ‘*’ saltutil.refresh_pillar

Sync the rest of the data
# salt ‘*’ saltutil.sync_all
```

# Execution module: Compound

```
Target database nodes labeled with ‘west’ using compound statement
# salt -C ‘G@location:west and *db* or E@west-db.*’ test.ping
```

# Configuration Management with state modules

# Outline

- Salt state concepts
- State in-depth
  - Salt state file
  - Top file

# Salt state concepts

- Declarative expression of desired state
  - Salt State Files (SLS)
- Automate and enforce deployment of systems
  - “Configuration management”
  - Idempotent
- State modules
  - Leverages execution modules

# SLS file example

/srv/salt/redis.sls

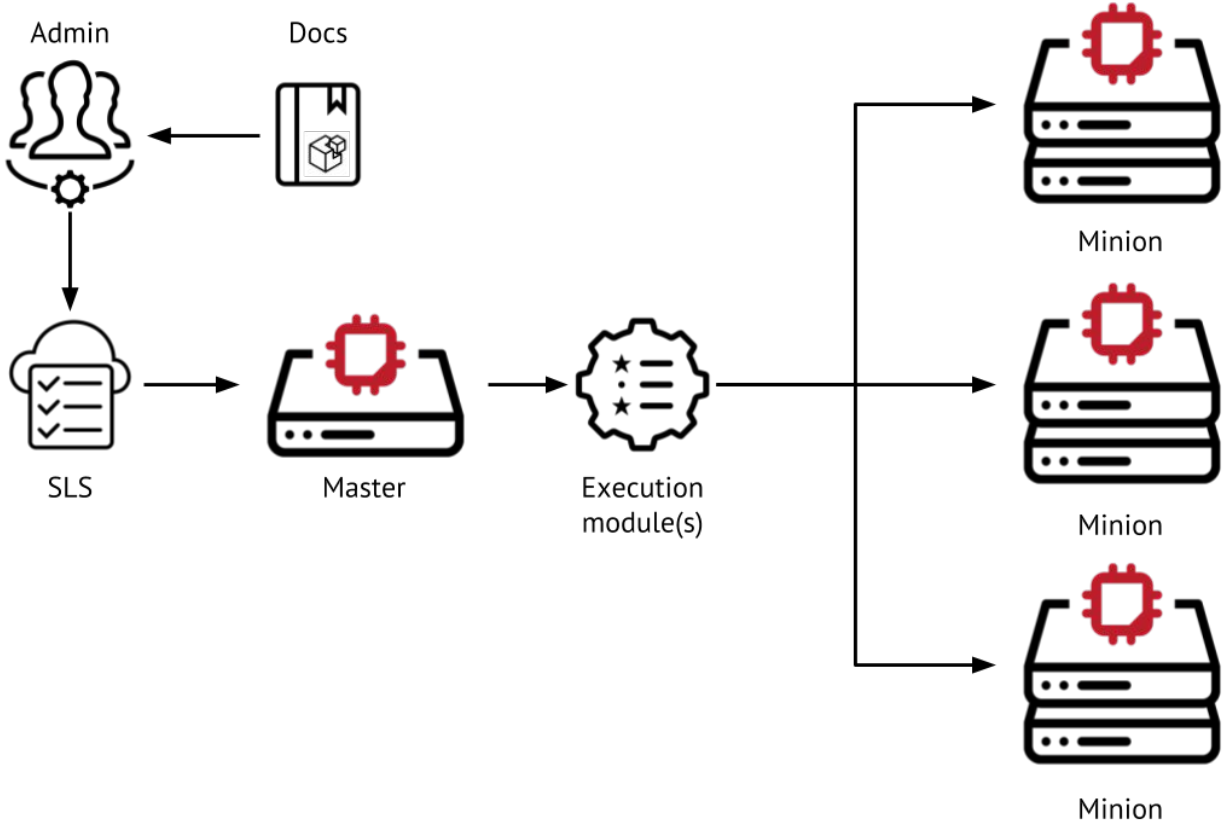
```
install_redis:                          #  ID declaration
  pkg.installed:                        #  State module.function name
    - name: redis-server                #  Function arguments
    - version: 2.8

start_redis:
  service.running:
    - name: redis
```

# SLS file example

```
root@master:~# salt \* state.apply redis
minion:
-------
          ID: install_redis
    Function: pkg.installed
      Result: True
...
```

# Salt state module processing



# Why not execution module script?

```
salt \* pkg.install redis-server

salt \* service.start redis-server
```

# State tree

- Representation of infrastructure deployment
- Hierarchy of directories containing state files

# State tree example

```
/srv/salt/
  |__base
     |__network.sls
     |__sshd.sls
  |__dev
     |__vim.sls
  |__prod
     |__httpd.sls
```

# State tree example

```
# /etc/salt/master
file_roots:
  base:
    - /srv/salt/base
  dev:
    - /srv/salt/dev
  prod:
    - /srv/salt/prod
```

# High state and Top file

- High state defines the complete configuration state which should be applied to a minion in a form of a
- List of state files
- Defined by top.sls file in top directory of state tree

# Top file example

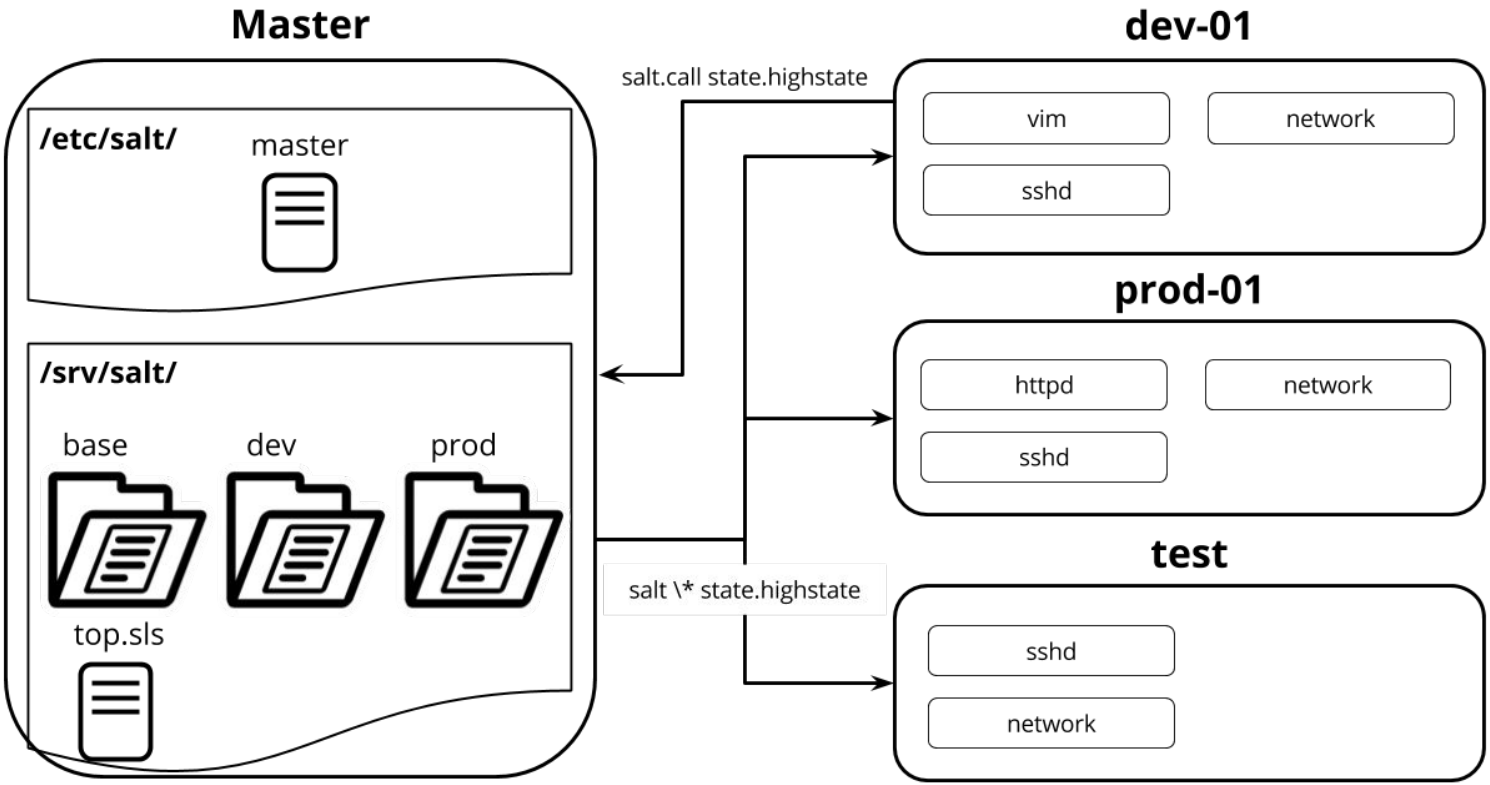
```
# /srv/salt/top.sls
  base:
    '*':
      - network
      - sshd
  dev:
    '*dev*':
      - vim
  prod:
    '*prod*':
      - httpd
```

# Top file example

```
# Run the top file states on all minions
salt \* state.highstate
# Alternatively
salt ‘*’ state.apply

# Poll the master for highstate from a minion
salt-call state.highstate
```

# State tree and top file summary



# Salt installation

# Salt Installation

- Bootstrap
- Packaged
- Source

# Salt Installation

## Bootstrap example

```
# Salt master node installation
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh install_salt.sh -P -M

# Salt minion node installation
curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com
sudo sh install_salt.sh -P
```

# Post-install configuration

```
# Configure specific IP interface for master
# /etc/salt/master
interface: 10.0.0.1

# Configure the minion to find master by IP address
# /etc/salt/minion
master: 10.0.0.1
```

# Post-install configuration

```
# Start salt processes as daemons
salt-master -d
salt-minion -d
# View the minion key fingerprints
salt-call --local key.finger
# List the minion keys known to master and compare with above
root@master:~# salt-key -L
# Accept all minion keys
root@master:~# salt-key -A
```

# FAQ

# FAQ

- Salt vs Fuel?
- Salt vs Ansible, Chef, Puppet?
- Is there a GUI for Salt?