Salt Experiences

and best practice taken from writing a salt module for ceph for SUSE and working as a contractor with salt.

Owen Synge

owen.synge@jaysnest.de

Who made Salt? Thomas Hatch (CTO)

Why are we here?

Why is automation important?

- Data centers without automation:
  - Are monotonous!
    - Setting up nodes is boring.
  - Are Unreliable!
    - Human errors creep in.
  - Do not scale!
    - Upgrading 20 nodes takes all day?
    - Installing 100 disks is tedious.
  - Have no recovery strategy!
    - Redeploying a server from bare metal often cures issues!

Who is the audience?

- Raise your hand if!
  - You know everything on the slides so far?
  - You are a system admin?
  - You are a dev-ops person?
  - You are a data center manager?
  - You are developer?
  - You already use?
    - Salt,puppet, chef, ansible?

About me

- Ceph is my 3rd distributed storage product.
  - Previously, EDG SE, dCache, (Also DPM, Castor and others)
- Been working on mass deployment for years
  - Packaging and automation with over 15 years experience.
    - In culture of people working on this for 30+ years!
- I am a software maker
  - I like admins and like to hear them complain about real stuff.
    - I also like making admins life easier.

Configuration Management Systems are similar.

Salt, Puppet, Chef, Ansible

- Configuration management tools are now common.
  - Not a new idea
    - Tomas Finnern gave a HEPIX talk about his new CMS at first HEPIX.
      - Replacing DESY’s old CMS over 25 years ago.
- CMS mostly do the same thing.
  - Manage state transitions on many computers.
  - Take booted bare OS to a production service
    - Non-interactively.

CMS: Usual structure to user

- Made up of a library of reusable modules.
- Have a DSL to call the libraries
  - Express dependency
  - Include other DSL files.
  - Express branching.
- Have meta-data about nodes.
  - Can query this meta-data in the DSL.

90% of what you do with a CMS system.

Giving nodes a role or set of roles.

- Use “top.sls” file on salt master.

```
base:
  'artifacts*':
    - jenkins-artifacts
  'jenkins*swarm*':
    - hydnstrasse
    - salt.roles
    - jenkins-swarm
  'osceph*':
    - sesceph
    - ceph_deploy
    - osceph
  '*':
    - hydnstrasse
    - salt.roles
```

Placing files on node.

- Use “sls files” on salt master.
  - You can also template files, and edit them.
    - But lets get on with the talk!

```
/etc/ceph/ceph.conf:
  file:
    - managed
    - source:
        # Where to get the source file will have
        # to be customized to your environment
        - salt://osceph/ceph.conf
    - user: root
    - group: root
    - mode: 644
    - makedirs: True
```

Adding packages to a node.

- Use “sls files” on salt master.
  - You can also have conditionals, and the like

```
ceph_packages_mon:
  pkg:
    - installed
    - names:
      - ceph-mon
      - python-ceph-cfg
```

Starting a service

- Usage example for an “sls file”.
- Nice way to start a service on any platform.
  - Salt works out the init system
    - Save you from caring if its
      - systemd or
      - sysVinit or
      - even BSD init.

```
openvpn:
  service.running:
    - enable: True
```

The other 10% of CMS's work

- Dependencies.
- Conditionals.
- Special modules.
  - Examples:
    - Cron, Apache, virtualenv, ceph, etc
  - Not everything you have to do is packaged by SUSE.
- Making your own modules!
  - Where this talk will start to focus on more.

Comparing Configuration management systems.

Puppet Comparing to Salt.

- Puppet has biggest deployment base.
- Polls master server for config to apply.
  - Minimized dependency on master service.
  - Salt was first a remote execution service.
    - Similar to mcollective.
      - Puppet added mcollective much later.
  - Salt added state management later.
- Puppet is ruby based while Salt is python based.

Chef comparing to Salt

- Chef has the biggest deployment base in Germany.
  - Quiet mature but I find docs confusing.
  - Newer than puppet.
- Chef relies on polling.
  - Salt allows you to push configuration to client.
- Chef uses json for config
  - Salt uses yaml.
- Chef is ruby based / Salt is python based.

I don't know chef as well as I know puppet and salt

Ansible comparing to Salt

- Ansible uses ssh rather than agents.
  - Pushes commands to clients.
  - Low startup costs.
  - Fast growing community (Red hat now owns Ansible).
- Python based just like salt.
- Newer than puppet and chef
- Great test suite.

I don't know ansible as well as I know puppet and salt

Salt compared to other CMS.

- Youngest major player.
- Steep learning curve.
  - Documentation is improving, but many components
- Event based model.
  - More moving parts (beacons, mines, pillars, reactors)
- Based on Event bus.
  - Events sent between

Salt : Programming your data center

- Basic usage similar to Puppet / Chef / Ansible
  - Thin DSL in YAML calling modules.
- Advanced usage:
  - Database integration
    - Pillar (as a data source) Mine (For read write)
  - Monitoring events.
    - Beacons (can dynamically be started on minions)
  - Event chaining.
    - Reactors, Orchestration engine.

Salt overview

- Message Queue at its core (zmq).
  - Master/Slave (Minion) model.
- Agent based, Event based.
- Think of it as a framework for distributed computing.
  - Extendable modules (master and minion).
  - Database modules (master and minion).
    - Backend can be simple jaml to full RDBMS (called pillars or mines)
  - Extendable attributes (called grains).
  - Events can be fired by any module.

Push Vs Pull in distributed computing.

Puppet,Chef, CFengine are pull based.

- Minion requests from master declarative config.
  - So can cache desired configuration.
    - Makes master off line issues trivial.
    - Makes intermittent connectivity failure irrelevant.
    - Makes overload of master simpler.
    - Maker error recovery simpler.
  - Not the beginners way to use a computer!
    - Minion nodes will converge with desired state.
    - This is a major objection for people proposing push models.

Salt like Ansible is Push based.

- Push based systems require ‘master’ to be running.
- Push based systems require ‘minion’ to be listening.
- Makes scaling difficult.
  - Some ‘minion’ will always be disconnected/down.
- Make reliability difficult.
  - Restarting the master will require minions to reconnect.

Why Salt and Ansible at SUSE and Redhat?

- Puppet should be installed with puppet.
  - Against the packaging philosophy of SUSE and Redhat
- Puppet uses Java technology.
- Chef has too steep learning curve.
- Large Customers already doing their own thing.
- Core customer bases need small scale automation
  - How can I install a cluster of 5 nodes?

Why Salt and Ansible at SUSE and Redhat?

- Both are Python based companies
  - Ruby modules terrible for long term support!
- Both companies don’t work at HEPIX scale
  - Both companies do not yet understand push limitations.
    - I have little doubt experience with containers will change this.
- Redhat and SUSE are not admin companies.
  - Overly optimistic about how things break.

Salt components

Salt components : master

- Salt master
  - Hosts event bus
  - Controls the cluster
  - Manages cluster authentication.
  - Has many sub components
    - We talk about this later
  - Provides simple remote execution options.

Salt Formula

- Custom DSL for salt called "salt formula"
  - Calls State / execution modules.
  - Follows YAML syntax
- With jinja2 template engine
  - Allows conditionals and looping
  - Works on DSL and for delivered content.
  - Gets variables from pillars and grains.
  - Use jinja2 sparingly!
    - When you need to do complex variable substitution use python

Salt Variables Pillars

- Yaml syntax
- Simple include syntax
- Simple to extend in python.
  - But do understand that this can be blocking.
    - So a blocking request can stop the entire salt system.

Salt components : minion

- Salt minion
  - Connects to the salt master
    - Marked up with grains (eg ipv4 address, Operating system)
  - Accepts instructions from salt master
  - Can execute python scripts (custom or premade)
    - As state / execution modules.
    - For Grains
      - So you can add your own, eg public x509 key
    - For Mines
      - So you can write data about node

Lets get on with Salt modules!

What is a Salt state/execution module

- CMS mostly do the same thing.
  - Manage state transitions on many computers.
  - Take booted bare OS to a production service
  - Non-interactively.
- Have a DSL to call the libraries
  - Express dependency
  - Include other DSL files.
  - Express branching.

Upgrading packages as an example.

- Two ways to illistrate using sls calling modules:
  - State module
  - Execution module:

```
# Upgrade with a state module
upgrade:
  pkg:
    - uptodate
```

```
# Upgrade with a execution module
upgrade:
  module.run:
    - name: pkg.upgrade
```

Why Execution modules?

- Run on the minion (remote node)
- Simple python methods exported to salt.
  - Least abstract interface.
    - Don't have to be idempotent
      - But it helps.
  - Very simple to develop.
- Simple to deploy
  - Place file in “/srv/salt/_modules”
    - Deploy to all nodes with “salt '*' saltutils.sync_all”

Execution module 'namespace'

- Giving your execution module a name.
  - So your module can be called in salt DSL.
  - So your module can be conditionally available.
    - Eg. Only runs on one platform
      - Zypper, yum, apt-get dependent on platform.

```
__virtualname__ = 'ceph'

def __virtual__():
    if HAS_CEPH_CFG is False:
        return False,
            'The %s execution module cannot be loaded: ceph_cfg unavailable.'
            % (__virtualname__)
    return __virtualname__
```

What methods are/can be exported.

- Any top level function.
  - Unless starts with a '_'
    - Eg. 'def _elephant()'
- Any method on an object
  - But only when no constructor parameters in object.
    - So only syntactical groupings as object created.
- Online help when you add docstrings:

```
def ceph_version():
    """
    Get the version of ceph installed
    """
    return ceph_cfg.ceph_version()
```

Execution modules and Errors.

- No rules on output structure!
  - Will be rendered as YAML to end user.
- Only way to fail is to raise exception.
  - This exception is reported to end user.
  - Note argument errors are swallowed by salt.
    - Can make debugging a little tricky.

Logging your modules.

- Salt uses standard python logging.
  - Your modules should also.
- Execution and State modules log locally.
  - So you can look at the local logs.
    - Default loglevel at warning.
    - Can be changed on command line or in config.

Why State modules?

- More user friendly than execution modules.
  - Report what was changed.
    - As a series of stages each with a
  - Standardized return value.
    - Allows branching on Success / Failure
    - Allows branching on No change.
  - Have a test function.
    - Only tells user what will be changed.

How are State modules different?

- Each function must match salt structure.
  - Calling syntax includes context.
    - Useful to allow introspection of calling.
      - Never used this.
  - Richer return syntax.
    - {'name' : name, 'result' : True, 'comment' : msg, 'changes' : {}}
    - Allowing triggers on success.
    - Allows admin to see changes
      - Also see if no changes.

State models are like execution modules

- Same name space idea.
  - Virtual function to enable namespace.
- Python docstrings to give end user help.
- Simple to deploy
  - Place file in “/srv/salt/_states”
    - Deploy to all nodes with “salt '*' saltutil.sync_all”

State modules must be idempotent

- Idempotent
  - When called again the output is the same.
  - Makes the system predicable.
    - So things wont break on second call.
  - Makes the users life easier.
    - Should not worry about recalling configuration.
  - Its what admins expect of configuration management.
    - So they can manage configuration drift over upgrade.
  - Executions modules should be idempotent
    - This is my opinion but is regarded as optional by salt upstream.

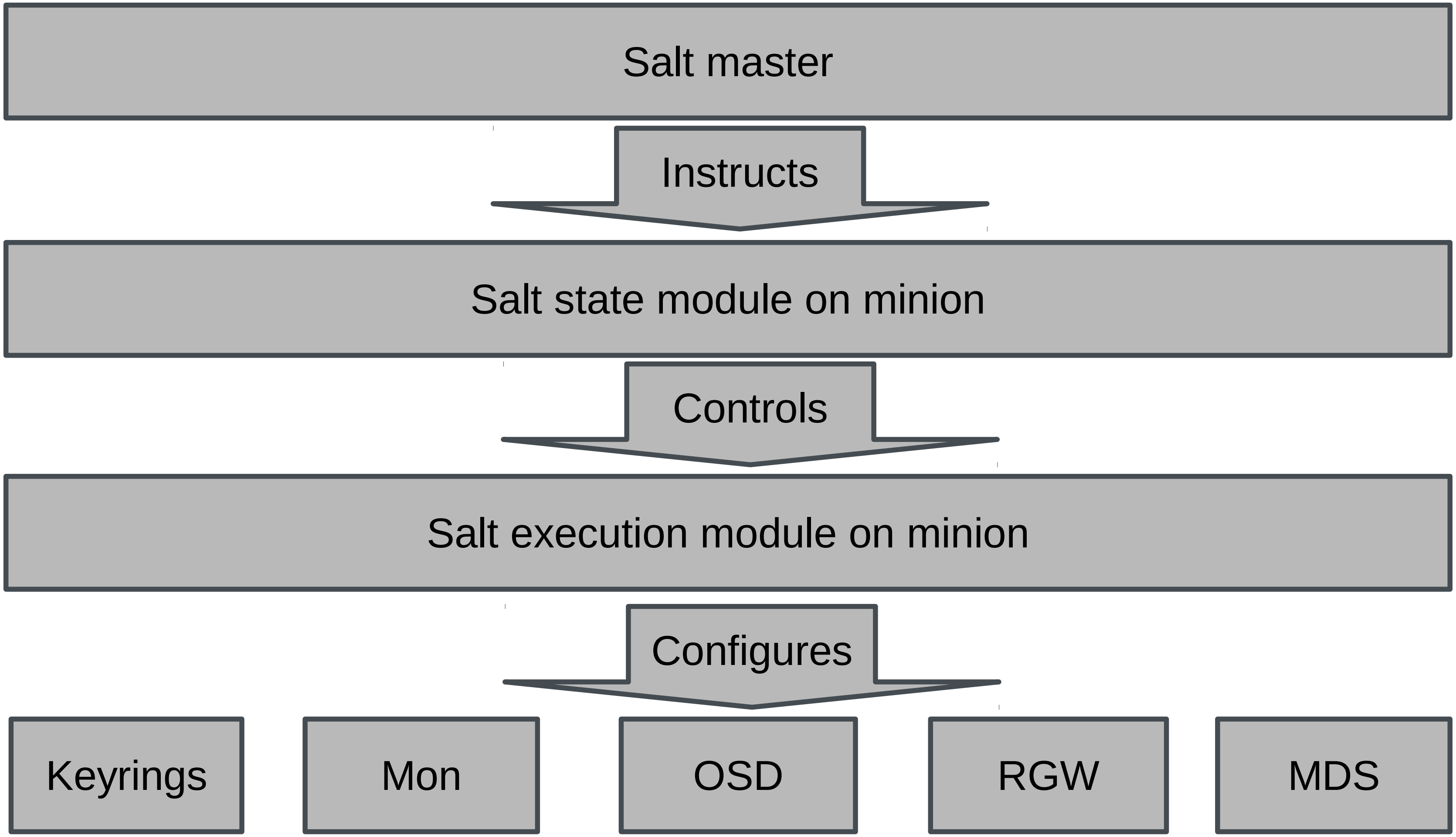
State modules reuse execution modules.

- State modules may call execution modules.
  - This is a nice to have.
    - but not essential see later
  - Know the execution modules contain needed methods.
- Possible optimization (Sometimes a good idea)
  - If the execution module has lots of state gathering.
    - Make pure python library.
    - Make execution module call library.
    - Call library directly from state module.

Example : Ceph Components

- Ceph has a nice dependency hierarchy
  - Keyrings (have a hierarchy of dependencies)
  - MON service (depend on keys)
  - OSD service (depend on mon + keys)
  - RGW service (depend on osd + mon + keys)
  - MDS Service (depend on osd + mon + keys)
  - RBD Service (depend on osd + mon + keys)
  - iSCSI Service (depend on rbd + osd + mon + keys)

Basic salt module implementation.



Salt master

Instructs

Salt state module on minion

Controls

Salt execution module on minion

Configures

Keyrings

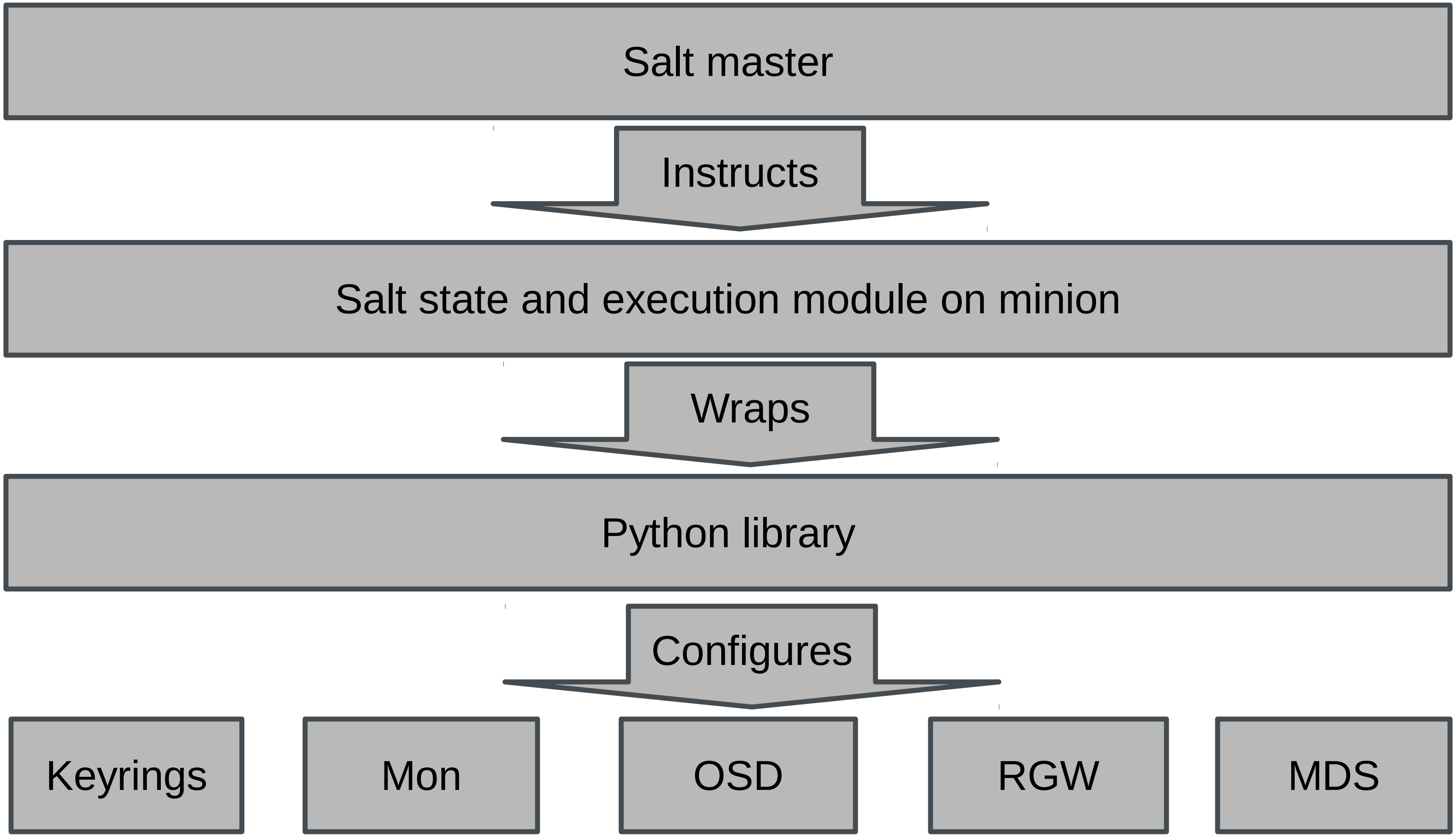
Mon

OSD

RGW

MDS

Reusable Ceph module implementation.



Salt master

Instructs

Salt state and execution module on minion

Wraps

Python library

Configures

Keyrings

Mon

OSD

RGW

MDS

Testing your modules.

- Config management on cluster is hard.
  - Functional tests are needed.
  - Unit test only go so far in this area.
  - Good to have test clusters.
- Salt has a testing framework built in.
  - Have not used it much as ..
- Alternatively if your code is a thin library wrapper.
  - You have all the standard python unit test options.
    - py.test, nosetest, tox, python-coverage etc.

Why have execution and state modules?

- Think of execution methods as primitives.
  - Best called from command line.
  - Don't have to be idempotent (So simpler to make)
    - But I recommend it.
  - Useful for debugging.
- Think of state modules as higher level functions.
  - Encapsulating logic of transformation.
  - Have to be idempotent.

A few words about your API

Function arguments

- Salt supports:
  - Explicit Arguments
    - Name
  - Defaulted Arguments
    - Name='default value'
  - Positional Arguments
    - *Args
  - Keyword arguments
    - **kwargs

My Advice about API's

- We all might have opinions here.
  - I like to use **kwargs when I am unsure
    - Because parameters change over life time of API.
    - Can catch unset parameters in code rather than API.
      - And returns error clearly.
    - Allows same method with many alternative parameter sets.
  - I don't like defaulted arguments.
    - They force order of parameters.
    - Defaulting also is effected with ordering.

Python Scope

- When an sls file is processed.
  - Modules are loaded.
  - Module method is then executed.
    - Maybe start processing another module.
    - Then next method is executed.
  - Scope is destroyed.
- This has caused issues for me with memoization.
  - Specifically storing paths of executable in library globals.
- May cause performance issues.
  - When state gathering is expensive.

Talk Summary

CMS:Take home summary.

- Configuration management is worth it.
  - 90% of your work is very very easy.
  - Benefits are imminence
- Most of what you want from a CMS
  - Install packages on specific nodes.
  - Configure files.
  - Start services.

Extending Salt:Take home summary.

- Mostly you don’t need to do this!
- Salt execution and state modules
  - Just python, and its easy.
  - You can even wrap standard python libraries.
    - You should then package them.
- All functions should be Idempotent.
  - Say this again its so important!

Questions?