Salt system architecture

Overview

Most users find it helpful to understand what Salt is and how it works before they begin the installation process. This page provides a high-level overview of the Salt system architecture and its different components.

What is Salt?

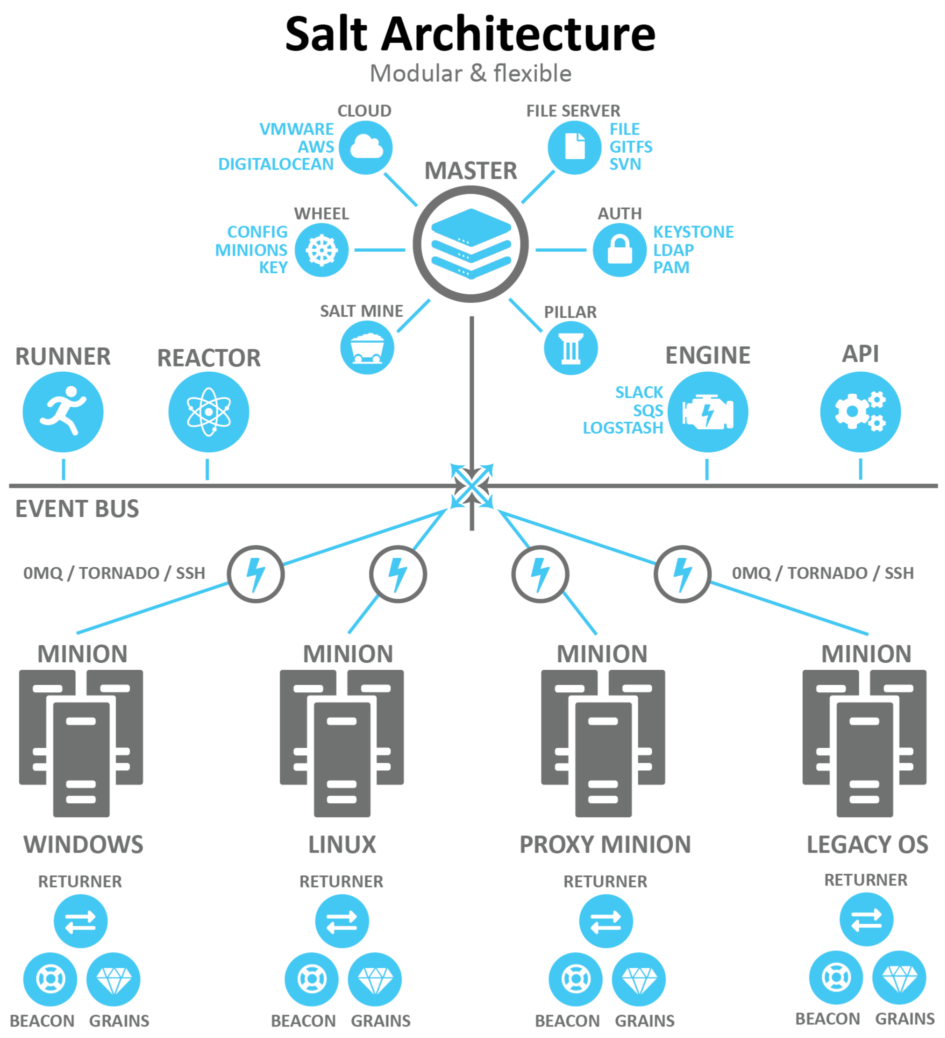
SaltStack Enterprise is powered by Salt, a Python-based open-source remote execution framework used for:

- Configuration management
- Automation
- Provisioning
- Orchestration

Salt is the technology that underlies the core functionality of SaltStack Enterprise. SaltStack Enterprise enhances and extends Salt, providing additional functionality and features that improve ease of use. For a summary of the SaltStack Enterprise infrastructure, see **SaltStack Enterprise system architecture** **[sse-system-architecture.html#sse-system-architecture]**.

The Salt system architecture

The following diagram shows the primary components of the basic Salt architecture:



[../_images/salt-architecture.png]

The following sections describe some of the core components of the Salt architecture that are relevant to SaltStack Enterprise installation.

SALT MASTERS AND SALT MINIONS

Salt uses the master-client model in which a master issues commands to a client and the client executes the command. In the Salt ecosystem, the Salt Master is a server that is running the `salt-master` service. It issues commands to one or more Salt Minions, which are servers that are running the `salt-minion` service and that are registered with that particular Salt Master.

Another way to describe Salt is as a publisher-subscriber model. The master publishes jobs that need to be executed and Salt Minions subscribe to those jobs. When a specific job applies to that minion, it will execute the job.

When a minion finishes executing a job, it sends job return data back to the master. Salt has two ports used by default for the minions to communicate with their master(s). These ports work in concert to receive and deliver data to the Message Bus. Salt's message bus is ZeroMQ, which creates an asynchronous network topology to provide the fastest communication possible.

TARGETS AND GRAINS

The master indicates which minions should execute the job by defining a ***target***. A target is the group of minions, across one or many masters, that a job's Salt command applies to.

Note

A master can also be managed like a minion and can be a target if it is running the `salt-minion` service.

The following is an example of one of the many kinds of commands that a master might issue to a minion. This command indicates that all minions should install the Vim application:

```
salt -v '*' pkg.install vim
```

In this case the glob `'*'` is the target, which indicates that all minions should execute this command. Many other targeting options are available, including targeting a specific minion by its ID or targeting minions by their shared traits or characteristics (called ***grains*** in Salt).

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. You can also create your own custom grain data.

Grain data is relatively static. However, grain data is refreshed when system information changes (such as network settings) or when a new value is assigned to a custom grain.

OPEN EVENT SYSTEM (EVENT BUS)

The event system is used for inter-process communication between the Salt Master and Salt Minions. In the event system:

- Events are seen by both the master and minions.
- Events can be monitored and evaluated by both.

The event bus lays the groundwork for orchestration and real-time monitoring.

All minions see jobs and results by subscribing to events published on the event system. Salt uses a pluggable event system with two layers:

- **ZeroMQ (0MQ)** - The current default socket-level library providing a flexible transport layer.
- **Tornado** - Full TCP-based transport layer event system.

One of the greatest strengths of Salt is the speed of execution. The event system's communication bus is more efficient than running a higher-level web service (http). The remote execution system is the component that all components are built upon, allowing for decentralized remote execution to spread load across resources.

SALT STATES

In addition to remote execution, Salt provides another method for configuring minions by declaring which ***state*** a minion should be in, otherwise referred to as ***Salt states***. Salt states make configuration management possible. You can use Salt states to deploy and manage infrastructure with simple YAML files. Using states, you can automate recursive and predictable tasks by queueing jobs for Salt to implement without needing user input. You can also add more complex conditional logic to state files with Jinja.

To illustrate the subtle differences between remote execution and configuration management, take the command referenced in the previous section about **Targets and grains [#targets-and-grains]** in which Salt installed the application Vim on all minions:

Methodology	Implementation	Result
Remote execution	• Run `salt -v '*' pkg.install vim` from the terminal	• Remotely installs Vim on the targeted minions
Configuration management	• Write a YAML state file that checks whether Vim is installed • This state file is then applied to the targeted minions	• Ensures that Vim is always installed on the targeted minions • Salt analyzes the state file and determines what actions need to be taken to ensure the minion complies with the state declarations • If Vim is not installed, it automates the processes to install Vim on the targeted minions

The state file that verifies Vim is installed might look like the following example:

```
# File:/srv/salt/vim_install.sls

install_vim_now:
  pkg.installed:
    - pkgs:
      - vim
```

To apply this state to a minion, you would use the `state.apply` module, such as in the following example:

```
salt '*' state.apply vim_install
```

This command applies the `vim_install` state to all minions.

Formulas are collections of states that work in harmony to configure a minion or application. For example, one state might trigger another state.

THE TOP FILE

It is not practical to manually run each state individually targeting specific minions each time. Some environments have hundreds of state files targeting thousands of minions.

Salt offers two features to help with this scaling problem:

- **The top.sls file** - Maps Salt states to their applicable minions.
- **Highstate execution** - Runs all Salt states outlined in `top.sls` in a single execution.

The top file maps which states should be applied to different minions in certain environments. The following is an example of a simple top file:

```
# File: /srv/salt/top.sls

base:
  '*':
    - all_server_setup
  '01webserver':
    - web_server_setup
```

In this example, `base` refers to the Salt environment, which is the default. You can specify more than one environment as needed, such as prod, dev, QA, etc.

Groups of minions are specified under the environment, and states are listed for each set of minions. This top file indicates that a state called `all_server_setup` should be applied to all minions `'*'` and the state called `web_server_setup` should be applied to the `01webserver` minion.

To run the Salt command, you would use the state.highstate function:

```
salt \* state.highstate
```

This command applies the top file to the targeted minions.

SALT PILLAR

Salt's pillar feature takes data defined on the Salt Master and distributes it to minions as needed. Pillar is primarily used to store secrets or other highly sensitive data, such as account credentials, cryptographic keys, or passwords. Pillar is also useful for storing non-secret data that you don't want to place directly in your state files, such as configuration data.

Salt pillar brings data into the cluster from the opposite direction as grains. While grains are data generated from the minion, the pillar is data generated from the master.

Pillars are organized similarly to states in a Pillar state tree, where `top.sls` acts to coordinate pillar data to environments and minions privy to the data. Information transferred using pillar has a dictionary generated for the targeted minion and encrypted with that minion's key for secure data transfer. Pillar data is encrypted on a per-minion basis, which makes it useful for storing sensitive data specific to a particular minion.

BEACONS AND REACTORS

The beacon system is a monitoring tool that can listen for a variety of system processes on Salt Minions. Beacons can trigger reactors which can then help implement a change or troubleshoot an issue. For example, if a service's response times out, the reactor system can restart the service.

Beacons are used for a variety of purposes, including:

- Automated reporting
- Error log delivery
- Microservice monitoring
- User shell activity
- Resource monitoring

When coupled with reactors, beacons can create automated pre-written responses to infrastructure and application issues. Reactors expand Salt with automated responses using pre-written remediation states.

Reactors can be applied in a variety of scenarios:

- Infrastructure scaling
- Notifying administrators
- Restarting failed applications
- Automatic rollback

When both beacons and reactors are used together , you can create unique states customized to your specific needs.

SALT RUNNERS AND ORCHESTRATION

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 instead of the Salt Minions. A Salt runner can be a simple client call or a complex application.

Salt provides the ability to orchestrate system administrative tasks throughout the enterprise. Orchestration makes it possible to coordinate the activities of multiple machines from a central place. It has the added advantage of being able to control the sequence of when certain configuration events occur. Orchestration states execute on the master using the state runner module.

When you run a multi-node installation, you are actually running an orchestration to install SaltStack Enterprise. In the multi-node installation scenario, you run an orchestration highstate designed by SaltStack. The highstate runs on your Salt Master and sets up the multi-node environment. It installs the core SaltStack Enterprise architecture on the three other nodes that will host PostgreSQL, Redis, and the Enterprise API (RaaS).