An Introduction to SaltStack Terminology and Concepts

Posted Oct 5, 2015

Top of Form

Bottom of Form

11

 13.5kviews [Configuration Management](https://www.digitalocean.com/community/tags/configuration-management?type=tutorials)

Tutorial Series

This tutorial is part 1 of 6 in the series: [Managing Development Environments with SaltStack](https://www.digitalocean.com/community/tutorials/an-introduction-to-saltstack-terminology-and-concepts#tutorial_series_53)

**Introduction**

Salt, or SaltStack, is a remote execution tool and configuration management system. The remote execution capabilities allow administrators to run commands on various machines in parallel with a flexible targeting system. The configuration management functionality establishes a client-server model to quickly, easily, and securely bring infrastructure components in line with a given policy.

In this guide, we will discuss some of the basic concepts and terminology needed to begin effectively learning to use Salt.

Salt Machine Roles

Salt's control structure is fairly simple as configuration management systems go. In a typical setup, there are only two distinct classes of machines.

**Master**

The Salt **master** is the machine that controls the infrastructure and dictates policies for the servers it manages. It operates both as a repository for configuration data and as the control center that initiates remote commands and ensures the state of your other machines. A daemon called salt-master is installed on the master to provide this functionality.

While it is possible to control infrastructure using a masterless configuration, most setups benefit from the advanced features available in the Salt master. In fact, for larger infrastructure management, Salt has the ability to delegate certain components and tasks typically associated with the master to dedicated servers. It can also operate in a tiered master configuration where commands can be relayed through lower master machines.

**Minions**

The servers that Salt manages are called **minions**. A daemon called salt-minion is installed on each of the managed machines and configured to communicate with the master. The minion is responsible for executing the instructions sent by the master, reporting on the success of jobs, and providing data about the underlying host.

How Salt Components Communicate

Salt masters and minions, by default, communicate using the ZeroMQ messaging library. This provides extremely high performance network communication between parties, allowing Salt to send messages and data at rapid speeds. Because ZeroMQ is a library and not an independent service, this functionality is available in the salt-master and salt-minion daemons natively.

When using ZeroMQ, Salt maintains a public key system for authenticating masters and minions. Upon first boot, a minion generates a key pair and sends its credentials to the master server it is configured to contact. The master can then accept this key after verifying the identity of the minion. The two parties can then communicate quickly and securely using ZeroMQ encrypted with the keys.

If for some reason it is impossible to install the salt-minion daemon on a node, Salt can also issue commands over SSH. This transport option is provided for convenience, but it degrades performance quite considerably and can lead to complications with other Salt commands in some instances. It is highly recommended that you use the salt-minion daemon where possible for performance, security, and simplicity.

Salt Terminology

Before diving into Salt, it is a good idea to familiarize yourself with some of the terminology that will be used. Salt has many powerful features, but it can be difficult to match names with their functionality at first. Let's take a look at some of the more general terms you are likely to see.

**Remote Execution: Execution Modules and Functions**

Salt attempts to provide a distinction between its remote execution and configuration management functions. The remote execution capabilities are provided by **execution modules**. Execution modules are sets of related **functions** that perform work on minions.

While Salt includes functions that allow you to run arbitrary shell commands on minions, the idea behind execution modules is to provide a concise mechanism for executing commands without having to "shell out" and provide detailed instructions about how to complete the process. The use of modules allows Salt to abstract the underlying differences between systems. You can get similar information from minions running Linux or BSD, even though the actual mechanisms to gather that data would be different.

Salt comes with a decent selection of [builtin execution modules](https://docs.saltstack.com/en/stage/ref/modules/all/index.html) to provide out-of-the-box functionality. Administrators can also write their own modules or include [community-written modules](https://github.com/saltstack/salt-contrib/tree/master/modules) to extend the library of commands that can be executed on minion machines.

**Configuration Management: States, Formulas, and Templates**

The configuration management functionality of Salt can be accessed by creating repositories of configuration files. The files contained within these repositories can be of a few different types.

**States and Formulas**

The configuration management portion of Salt is primarily implemented using the **state** system.

The state system uses **state modules**, which are distinct from the execution modules described above. Fortunately, state and execution modules tend to mirror each other quite closely. The state system is aptly named, as it allows an administrator to describe the state that a system should be placed in. As with execution modules, most state modules represent functionality shortcuts and provide easy syntax for many common actions. This helps maintain readability and removes the necessity of including complex logic in the configuration management files themselves.

Salt **formulas** are sets of state module calls, arranged with the aim of producing a certain result. These are the configuration management files that describe how a system should look once the formula has been applied. By default, these are written in the YAML data serialization format, which provides a very good middle ground between high-readability and machine-friendliness.

The Salt administrator can apply formulas by mapping minions to specific sets of formulas. Formulas can also be applied in an ad hoc manner as needed. Minions will execute the state modules found within to bring its system in line with the provided policy.

A good collection of Salt formulas created by the SaltStack organization and community can by found in[this GitHub account](https://github.com/saltstack-formulas).

**Templates**

**Templating** allows Salt formulas and other files to be written in a more flexible manner. Templates can use the information available about the minions to construct customized versions of formula or configuration files. By default, Salt uses the Jinja template format, which provides substitution functionality and simple logical constructs for decision making.

**Renderers** are the components that runs the template to produce valid state or configuration files. Renderers are defined by the templating format that constitutes the input and the data serialization format that will be produced as an output. Considering the defaults described above, the default renderer processes Jinja templates in order to produce YAML files.

**Querying about and Assigning Information to Minions**

In order to manage vast numbers of systems, Salt requires some information about each of the host systems. The templates described above can use data associated with each system to tailor the behavior of each minion. There are a few different systems in place to query about or assign this information to hosts.

**Grains**

Salt **grains** are pieces of information, gathered by and maintained by a minion, primarily concerning its underlying host system. These are typically collected by the salt-minion daemon and passed back to the master upon request. This functionality can be leveraged for many different purposes.

For instance, grain data can be used for targeting a specific subset of nodes from the pool of minions for remote execution or configuration management. If you want to see the uptime of your Ubuntu servers, grains allow you to target only these machines.

Grains can also be used as arguments for configuration changes or commands. For example, you can use grains to get the IPv4 address associated with the eth0 interface for a change to a configuration file or as an argument to a command.

Administrators can also assign grains to minions. For instance, it is fairly common to use grains to assign a "role" to a server. This can then be used to target a subset of nodes similar to the operating system example above.

**Pillars**

While it is possible to assign grains to minions, the vast majority of configuration variables will be assigned through the **pillars** system. In Salt, a pillar represents a key-value store that a minion can use to retrieve arbitrary assigned data. This functions as a dictionary data structure which can be nested or tiered for organizational purposes.

Pillars offer a few important advantages over the grain system for assigning values. Most importantly, pillar data is only available to the minions assigned to it. Other minions will not have access to the values stored within. This makes it ideal for storing sensitive data specific to a node or a subset of nodes. For instance, secret keys or database connection strings are often provided in a pillar configuration.

Pillar data is often leveraged in the configuration management context as a way to inject variable data into a configuration template. Salt offers a selection of template formats for replacing the variable portions of a configuration file with the items specific to the node that will be applying it. Grains are also often used in this way for referencing host data.

**Mine**

Salt **mine** is an area on the master server where the results from regularly executed commands on minions can be stored. The purpose of this system is to collect the results of arbitrary commands run on minion machines. This global store can then be queried by other components and minions throughout your infrastructure.

The Salt mine only stores the most recent result for each command run, meaning that it will not help you if you need access to historic data. The main purpose of the mine is to provide up-to-date information from minion machines as a flexible supplement to the grain data that is already available. Minions can query data about their counterparts using the mine system. The interval in which the minion refreshes the data in the mine can be configured on a per-minion basis.

**Additional Functionality**

Salt provides a few other systems that do not fit nicely into the above categories.

**Reactors**

The Salt **reactor** system provides a mechanism for triggering actions in response to generated events. In Salt, changes occurring throughout your infrastructure will cause the salt-minion or salt-masterdaemons to generate events on a ZeroMQ message bus. The reactor system watches this bus and compares events against its configured reactors in order to respond appropriately.

The main goal of the reactor system is to provide a flexible system for creating automated situational responses. For instance, if you have developed an auto-scaling strategy, your system will automatically create nodes to dynamically meet your resource demands. Each new node would trigger an event. A reactor could be set up to listen to these events and configure the new node, incorporating it into the existing infrastructure.

**Runners**

Salt **runners** are modules that execute on the master server instead of minions. Some runners are general purpose utilities used to check the status of various parts of the system or to do maintenance. Some are powerful applications that provide the ability to orchestrate your infrastructure on a broader scale.

**Returners**

Salt **returners** are used to specify alternative locations where the results of an action run on a minion will be sent. By default, minions return their data to the master. A returner allows the administrator to re-route the return data to a different destination. Typically, this means that the results are returned to the destination specified by the returner *and* to the process that initiated the minion command.

Most often, returners will pass the results off to a database system or a metrics or logging service. This provides a flexible method for getting arbitrary data into these systems. Returners can also be used to collect Salt-specific data like job caches and event data.

Salt Commands

Salt provides a number of commands to take advantage of the components outlined above. There is some significant crossover in terms of functionality between these tools, but we've attempted to highlight their primary functions below.

* **salt-master**: This is the master daemon process. You can start the master service with this command directly, or more typically, through an init script or service file.
* **salt-minion**: Likewise, this is minion daemon process, used to communicate with the master and execute commands. Most users will also start this from init scripts or service files.
* **salt-key**: This tool is used to manage minion public keys. This tool is used to view current keys and to make decisions about public keys sent by prospective minions. It can also generate keys to place on minions out-of-band.
* **salt**: This command is used to target minions in order to run ad-hoc execution modules. This is the main tool used for remote execution.
* **salt-ssh**: This command allows you to use SSH as an alternative to ZeroMQ for the transport mechanism.
* **salt-run**: This command is used to run runner modules on the master server.
* **salt-call**: This command is used to run execution modules directly on a minion you are logged into. This is often used to debug problematic commands by bypassing the master.
* **salt-cloud**: This command is used to control and provision cloud resources from many different providers. New minions can easily be spun up and bootstrapped.

There are some other commands as well, like salt-api, salt-cp, and salt-syndic, which aren't used quite so often.

Conclusion

Now that you are familiar with the basic SaltStack terminology and have a high level understanding of what the tools you'll encounter are responsible for, you can begin setting up Salt to control your infrastructure. In the [next guide](https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-salt-master-and-minion-servers-on-ubuntu-14-04), we will cover how to install and configure a Salt master server on an Ubuntu 14.04 server. We will also demonstrate how to configure new minion servers to bring them under your master's management.

SaltStack Infrastructure: Installing the Salt Master

Posted Oct 5, 2015

Top of Form

Bottom of Form

10

 9.9kviews [Configuration Management](https://www.digitalocean.com/community/tags/configuration-management?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

Tutorial Series

This tutorial is part 2 of 6 in the series: [Managing Development Environments with SaltStack](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-installing-the-salt-master#tutorial_series_53)

**Introduction**

SaltStack, or Salt, is a powerful remote execution and configuration management system that can be used to easily manage infrastructure in a structured, repeatable way. In this series, we will be demonstrating one method of managing your development, staging, and production environments from a Salt deployment. We will use the Salt state system to write and apply repeatable actions. This will allow us to destroy any of our environments, safe in the knowledge that we can easily bring them back online in an identical state at a later time.

In this article, we'll introduce the various environments we'll be building out and we will install the Salt master daemon onto our primary server in order to get started. This is the core system that will house our configuration instructions, control our infrastructure nodes, and manage our requirements.

**Prerequisites**

To complete this guide, you will need a clean Ubuntu 14.04 server. This server will need to have private networking enabled.

On this server, you should have a non-root account configured with sudo privileges. You can learn how to set up an account of this type in our [Ubuntu 14.04 initial server setup guide](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-14-04).

A later article in this series will use the salt-cloud command to spin up infrastructure servers using the DigitalOcean cloud. You can use other cloud providers if you would like, or even virtual instances using software like vagrant. However, this is outside of the scope of this guide and you will have to do your own research in these areas.

Overview of the Infrastructure We Will Build

The servers we spin up with salt-cloud will represent our three environments we want to maintain for application development. Because our Salt master server will maintain the configuration for each of the servers we need, we will be able to spin these servers down when we're not using them. For instance, if your development team has halted work for the holidays, you can spin down your non-production environments. When you return from your break, you can easily rebuild them and redeploy your application on top of this fresh infrastructure.

We will break down our example infrastructure into development, staging, and production.

Our development infrastructure will be our most modest. It will simply contain a single web server and an associated database server. Decoupling the database and web server will be enough to ensure that our application is built with remote data in mind.

The staging environment will be more robust. Most advice that you'll find recommends configuring your staging environment to be as similar to your intended production environment as possible. With this in mind, our staging environment will consist of two web servers in order to spread out the traffic load. We will distribute traffic between these two servers using a load balancer. On the database side, we will spin up two database servers. We will set up master-master replication between these two servers so that both can accept write requests. Master-master replication has some disadvantages, but it allows us to be a bit lazy in our application design (any database server can receive writes) and it allows us to demonstrate a fairly complex configuration management scenario.

As we stated before, the production environment will be very similar to the staging environment. The only difference in our design will be an additional load balancer up front to provide high availability and fail over. If you wish, you could also use higher capacity servers for your production load. This is often needed to adequately handle traffic requirements, even though it makes it more difficult to assess load in the staging environment.

Keep in mind that while we are creating configurations for the environments listed above, they do not all have to be running at the same time. This is especially true during testing. Throughout this series, you will likely only have a few servers active at any one time. This is desirable from a cost perspective, but destroying and bringing up our environments as needed also ensures that our environment bootstrapping is robust and repeatable.

Now that you know the general layout of the environments we will be configuring, we can get our Salt master up and running.

Installing the Salt Master

Start by logging into the server you intend to set up as the Salt master as a non-root user with sudoprivileges.

There are quite a [few different ways](https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-salt-master-and-minion-servers-on-ubuntu-14-04) to install the Salt master daemon on a server. There are PPAs available for Ubuntu, but these can often be out-of-date. The best approach for planning and managing configuration management software is to target a specific version. This will allow you to update your systems (after thorough testing) in a planned and structured way instead of relying on whatever is available from a repository at the time of installation.

For this guide, we will be targeting Salt version [v2015.8.0](https://github.com/saltstack/salt/releases/tag/v2015.8.0), the latest stable version at the time of this writing. If you choose a different version or installation method, be aware that the processes in this guide may not work as written.

The easiest way to install a specific version is with SaltStack's bootstrap script. Download the latest bootstrap script to your home directory by typing:

* cd ~
* curl -L https://bootstrap.saltstack.com -o install\_salt.sh

Feel free to take a look at the contents of the downloaded script until you are comfortable with the operations it will perform.

When you are ready to install the Salt master, you can run the script with the sh shell. We will pass it the -P flag to indicate that we are okay with allowing dependency installations with pip, the Python package manager. We will also need to include the -M flag to indicate that we want to install the master daemon. Finish off the command by including git v2015.8.0, which tells the script to fetch the specified release tag from the [SaltStack GitHub repo](https://github.com/saltstack/salt) and use that for the installation:

* sudo sh install\_salt.sh -P -M git v2015.8.0

The script will install all of the necessary dependencies, pull the version specified from the git repo, and install the Salt master and minion daemons, as well as some related Salt utilities.

The installation should be rather straight forward. Next, we can begin to configure our Salt master.

Configure the Salt Master

The first thing we need to is to edit the main Salt master configuration file. Open it now with sudoprivileges:

* sudo nano /etc/salt/master

The configuration file is fairly long and well-commented. You only need to uncomment and set options when you wish to deviate from the default values. We only need to make a few edits to start off.

First, find the file\_recv option in the file. Enabling this allows Salt minions to send files to the Salt master. This is extremely helpful when creating states to get configuration files you wish to modify, however, it does come with some risk. We will enable it for the duration of this guide. You can disable it afterwards if you'd like:

/etc/salt/master

file\_recv: True

Next, we will need to set the file\_roots dictionary. The Salt master includes a file server that it uses to store and serve files for the entire infrastructure. This includes the configuration management State files themselves, as well as any minion files that are managed by our system. This YAML dictionary defines the root of the file server, which will be located at /srv/salt. We need to specify that this is located under the "base" environment, the mandatory default environment for all Salt deployments:

/etc/salt/master

file\_roots:

base:

- /srv/salt

Note

It is important to replicate the formats given exactly. Salt uses YAML-style configuration files. YAML requires strict attention to spacing and indentation in order to ensure correct interpretation. Typically, each level of indentation will be two spaces.

The last item that we need for now is the pillar\_roots dictionary. The pillar system is used to store configuration data that can be restricted to certain nodes. This allows us to customize behavior and to prevent sensitive data from being seen by infrastructure components not associated with the data. This format mirrors the file\_roots exactly. The location of our pillar data will be at /srv/pillar:

/etc/salt/master

pillar\_roots:

base:

- /srv/pillar

Save and close the file when you are finished.

We can go ahead and create the directories we referenced in the configuration file by typing:

* sudo mkdir -p /srv/{salt,pillar}

Configure the Minion Daemon on the Salt Master

We want to also configure our Salt master server to accept Salt commands. We can do this by configuring the minion daemon on our server. Open the file to get started:

* sudo nano /etc/salt/minion

The only item we need to change here is the location of the master server. Since both daemons are operating on the same host, we can set the address to the local loopback interface:

/etc/salt/minion

master: 127.0.0.1

Save and close the file when you are finished.

Restart the Services and Accept the Salt Keys

Now that we have the Salt master and minion configuration in place, restart the services to pick up our changes:

* sudo restart salt-master
* sudo restart salt-minion

Before the Salt master can communicate securely with a minion (even on the same server), it must accept the minion's key. This is a security feature. You can see all accepted and pending keys by typing:

* sudo salt-key --list all

If your daemons were configured correctly and were restarted, you should see a key for your Salt master server in the "Unaccepted Keys" section. In our case, our Salt master is being hosted on a machine called "sm":

Output

Accepted Keys:

Denied Keys:

Unaccepted Keys:

sm

Rejected Keys:

You can accept this key by passing the server's minion ID (sm in this case) to thesalt-keycommand with the-a` flag:

* sudo salt-key -a sm

If you check again, your key will have moved to the "Accepted Keys" section:

* sudo salt-key --list all

Output

Accepted Keys:

sm

Denied Keys:

Unaccepted Keys:

Rejected Keys:

You can verify that your Salt master server now responds to Salt commands by typing:

* sudo salt '\*' test.ping

You should get a response back that looks something like this:

Output

sm:

True

Your Salt master server is now up and running.

Conclusion

In this guide, we got started managing our infrastructure by running through the initial configuration of our Salt master server. This is the central server within our management design which will be used both as a control center and as a repository of configuration data.

In the [next guide](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-configuring-salt-cloud-to-spin-up-digitalocean-resources) in this series, we will configure our Salt master server with our DigitalOcean API credentials. We will create a provider configuration that allows us to connect to our DigitalOcean account using the salt-cloud command and create and manage cloud resources. We will create the profiles for our infrastructure machines so that we can define the properties for each of our servers.

SaltStack Infrastructure: Creating Salt States for Nginx Web Servers

Posted Oct 5, 2015

Top of Form

Bottom of Form

2

 7.9kviews [Configuration Management](https://www.digitalocean.com/community/tags/configuration-management?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

Tutorial Series

This tutorial is part 4 of 6 in the series: [Managing Development Environments with SaltStack](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-creating-salt-states-for-nginx-web-servers#tutorial_series_53)

**Introduction**

SaltStack, or Salt, is a powerful remote execution and configuration management system that can be used to easily manage infrastructure in a structured, repeatable way. In this series, we will be demonstrating one method of managing your development, staging, and production environments from a Salt deployment. We will use the Salt state system to write and apply repeatable actions. This will allow us to destroy any of our environments, safe in the knowledge that we can easily bring them back online in an identical state at a later time.

In our [previous guide](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-configuring-salt-cloud-to-spin-up-digitalocean-resources), we expanded our Salt master server's capabilities by setting up the DigitalOcean provider for salt-cloud. We created the files needed to allow us to spin up fresh servers for each of our environments. In this article, we will begin to dive into configuration management by creating Salt state files for Nginx. Nginx will be used on our web server nodes in all three environments in order to handle web requests.

Create the Main Nginx State File

Salt handles configuration management through its state system. In the simplest case, these are controlled by files located within Salt's file server root (which we configured as /srv/salt). To start off our Nginx configuration, we'll create a directory in this location specific to the software we are configuring:

* sudo mkdir /srv/salt/nginx

State files have an .sls suffix. An init.sls file within a directory functions as the main configuration file for that specific Salt state or formula. We refer to the parent directory name to execute the functionality contained within the associated init.sls file.

With that in mind, create and open an init.sls file within this directory to get started:

* sudo nano /srv/salt/nginx/init.sls

**The Nginx Package and Service States**

We will start by creating a state using the nginx identifier. This will serve as the unique name for this particular state within the Salt state system. Since we won't be including the "name" attribute for our state modules, it will also serve as the target to be installed (for the pkg.installed function) and the service to be running (for the service.running function).

We want Nginx to automatically reload under certain conditions: when the package is updated, when the main configuration file has been changed, or when the default server block file is modified. We can tell Salt to restart the Nginx service when these conditions occur by using watch:

/etc/salt/nginx/init.sls

nginx:

pkg:

- installed

service.running:

- watch:

- pkg: nginx

- file: /etc/nginx/nginx.conf

- file: /etc/nginx/sites-available/default

The pkg: and file: keys beneath the watch: key represent the state modules associated with the resources to watch. The pkg resource is taken care of within the first part of this same definition. We will have to create the states to match the file resources next.

**The Nginx Configuration File States**

We can start with the /etc/nginx/nginx.conf file. We would like to make this a managed file. In Salt terminology, this just means that we will define the contents of the file on the master server and upload it to each minion who needs it. We will set rather typical permissions and ownership on the file. The source references a location within the Salt file server (our current file is within this structure as well). We will be creating this path and file momentarily:

/etc/salt/nginx/init.sls

nginx:

pkg:

- installed

service.running:

- watch:

- pkg: nginx

- file: /etc/nginx/nginx.conf

- file: /etc/nginx/sites-available/default

/etc/nginx/nginx.conf:

file.managed:

- source: salt://nginx/files/etc/nginx/nginx.conf

- user: root

- group: root

- mode: 640

We also want to control the contents of the /etc/nginx/sites-available/default file. This defines the server block that controls how our content will be served. The state block is fairly similar to the last one. The major difference is that this file will be a Jinja template.

Jinja templates allow Salt to customize some of the contents of the file with details specific to each of the minions where it will be placed. This means that we can pull information from each host and construct an appropriate, customized version of the file for each of our web servers. We indicate that this file will use Jinja with the template option. We also will use the .jinja suffix on the source file so that we can tell at a glance that the file is a template:

/etc/salt/nginx/init.sls

. . .

/etc/nginx/nginx.conf:

file.managed:

- source: salt://nginx/files/etc/nginx/nginx.conf

- user: root

- group: root

- mode: 640

/etc/nginx/sites-available/default:

file.managed:

- source: salt://nginx/files/etc/nginx/sites-available/default.jinja

- template: jinja

- user: root

- group: root

- mode: 640

We have our default server block file slated to be placed in the sites-available directory on the minion hosts. However, we still need to link the file to the sites-enabled directory to activate it. We can do that with the file.symlink function. We just need to provide the original file location as the target. We also need to "require" that file so that this state is only executed after the previous state has completed successfully:

/etc/salt/nginx/init.sls

. . .

/etc/nginx/sites-available/default:

file.managed:

- source: salt://nginx/files/etc/nginx/sites-available/default.jinja

- template: jinja

- user: root

- group: root

- mode: 640

/etc/nginx/sites-enabled/default:

file.symlink:

- target: /etc/nginx/sites-available/default

- require:

- file: /etc/nginx/sites-available/default

**The State for Our Default Site Content**

We have our Nginx installation and configuration states written. Now, we just need to create a state for ourindex.html file which will be the actual content for our site.

This state uses the exact same format as our previous template state. The only differences are the identifier, the source, and the permissions mode on this file:

/etc/salt/nginx/init.sls

. . .

/etc/nginx/sites-enabled/default:

file.symlink:

- target: /etc/nginx/sites-available/default

- require:

- file: /etc/nginx/sites-available/default

/usr/share/nginx/html/index.html:

file.managed:

- source: salt://nginx/files/usr/share/nginx/html/index.html.jinja

- template: jinja

- user: root

- group: root

- mode: 644

When you are finished, save and close this file. We are done with the actual Nginx state information for the moment.

Install Nginx and Transfer Original Files to the Salt Master

We have our main Nginx Salt state file created. However, some of the states we created references files on the Salt master's file server that do not exist yet.

Since our files will be largely the same as the default files installed by Ubuntu's Nginx package, the easiest way for us to start is with the files from that package. The web servers from one of our environments offer a perfect place to install Nginx so that we can grab the necessary files.

If you do not already have one of your environments spun up, select one of your environment map files to deploy. We will use the "stage" environment in this series because it is the smallest environment that has all of the server types we'll need.

* sudo salt-cloud -P -m /etc/salt/cloud.maps.d/stage-environment.map

Once your servers are up and running, choose one of your web servers to install Nginx onto. We are just going to use the pkg execution module at this time, since our states are not fully functional yet:

* sudo salt stage-www1 pkg.install nginx

When we set up our Salt master configuration, we enabled the file\_recv option. This allows us to request minions to push certain files back up to the master. We can use this to grab the default versions of the files we'll be managing:

* sudo salt stage-www1 cp.push /etc/nginx/nginx.conf
* sudo salt stage-www1 cp.push /etc/nginx/sites-available/default
* sudo salt stage-www1 cp.push /usr/share/nginx/html/index.html

These files should now be available on the master. The path to these files is recreated within the/var/cache/salt/master/minions/minion\_id/files directory. In our case, the minion ID would bestage-www1. We can copy the directories beneath this location, which represents the file paths on the minion, to our Salt state directory by typing:

* sudo cp -r /var/cache/salt/master/minions/stage-www1/files /srv/salt/nginx

If you look at the contents of your state directory, you will see a new directory called "files". Beneath this directory, the relevant directories within the minion's filesystem and the three files we copied are available:

* find /srv/salt/nginx -printf "%P\n"

Output

files

files/usr

files/usr/share

files/usr/share/nginx

files/usr/share/nginx/html

files/usr/share/nginx/html/index.html

files/etc

files/etc/nginx

files/etc/nginx/sites-available

files/etc/nginx/sites-available/default

files/etc/nginx/nginx.conf

init.sls

This is where all of our managed files will be maintained. This aligns with the "source" location we set in our Nginx state file.

Since we now have all of the files we need pulled from the minion where Nginx was installed, we can destroy the minion and rebuild it. This will ensure that later on, our state files can be tested on a clean server. Destroy the Nginx minion:

* sudo salt-cloud -d stage-www1

After waiting for the event to process, we can rebuild the minion.

We typically would use the map file for this, but since we are only rebuilding a single server, it's actually preferable to use the stage-web profile directly. We can then use the cloud.profile Salt execution function instead of salt-cloud, which allows us to add the --async flag. Basically, this lets us rebuild ourstage-www1 server in the background as we continue to work. We will have to target our Salt master in this command since this is the server with the cloud profiles we need:

* sudo salt --async sm cloud.profile stage-web stage-www1

While our stage-www1 node is rebuilding in the background, we can continue.

Configure the /etc/nginx/nginx.conf File

Let's look at the main Nginx configuration file first, which will be placed at /etc/nginx/nginx.conf on our minions. We can find this path under the files directory with out Nginx state directory:

* cd /srv/salt/nginx/files/etc/nginx

We're actually not going to modify this file at the moment, but we can do ourselves a favor and back up the original right now:

* sudo cp nginx.conf nginx.conf.orig

This will give us a good point of reference for customizations we might make in the future. We can quickly see any changes we've made by typing:

* diff nginx.conf nginx.conf.orig

In the future, if we find we need to customize Nginx's configuration in our various environments (for instance, we might want to match the worker\_processes with the number of CPUs on our production servers later on), we might want to transition to using a template file. We don't need this at the moment so, as a non-template file, our changes will be hard-coded.

As we stated earlier though, we don't need any modifications at this time. Let's move on.

Configure the /etc/nginx/sites-available/default Template

Next, let's take a look at our default server block template. We can find the original in this directory:

* cd /srv/salt/nginx/files/etc/nginx/sites-available

Again, we should copy the original to a backup location in case we need it later:

* sudo cp default default.orig

We can then rename the file so that it has a .jinja extension. This will visually remind us that this file is a template and not a usable file by itself:

* sudo mv default default.jinja

Now, we can open the template file to make some changes:

* sudo nano default.jinja

At the very top of the file, we need to start utilizing Jinja's templating features. Our default server block needs to render different files depending on whether the web server will be behind a load balancer.

When connections are being received through a load balancer, we want our web server to restrict its traffic to the private interface. When we're in the development environment however, we do not have a load balancer, so we'll want to serve over the public interface. We can create this distinction with Jinja.

We will create a variable called interface which should contain the interface we want the address of. We'll test if the environment of the minion is set to "dev", in which case we'll use the eth0 interface. Otherwise, we'll set it to eth1, the server's private interface. We'll then use the grains.get execution module function to grab the address associated with the selected interface and use that as the value for the addr variable. We will add this to the very top of the file:

/srv/salt/nginx/files/etc/nginx/sites-available/default.jinja

{%- **set** interface = 'eth0' if salt['grains.get']('env') == 'dev' else 'eth1' -%}

{%- **set** addr = salt['network.interface\_ip'](interface) -%}

# You may add here your

# server {

# ...

# }

. . .

Next, we can edit the server block further down in the file. We can use the addr variable we set at the top in the listen and server\_name directives. We've removed the IPv6 and default server portions to restrict what this block serves:

/srv/salt/nginx/files/etc/nginx/sites-available/default.jinja

{%- **set** interface = 'eth0' if salt['grains.get']('env') == 'dev' else 'eth1' -%}

{%- **set** addr = salt['network.interface\_ip'](interface) -%}

. . .

server {

**listen** {{ **addr** }}:80;

**root** /usr/share/nginx/html;

**index** index.html index.htm;

**server\_name** {{ **addr** }};

**location** / {

**try\_files** $uri $uri/ =404;

}

}

Save and close the file when you are finished.

Configure the /usr/share/nginx/html/index.html Template

We can now move on to the index.html file. Move over to the directory on the Salt master that contains the file:

* cd /srv/salt/nginx/files/usr/share/nginx/html

Inside, we'll need to start with the same procedure that we used last time. We should store a copy of the original file for auditing and backup purposes. We should then rename the file to indicate that this will be a template:

* sudo cp index.html index.html.orig
* sudo mv index.html index.html.jinja

Open up the template file so we can make the modifications we need:

* sudo nano index.html.jinja

At the top, we'll set another variable using Jinja. We'll use the grains.get execution module function to grab the minion's hostname. We'll store this in the host variable:

{% set host = salt['grains.get']('host') -%}

<!DOCTYPE html>

<**html**>

. . .

We'll then use this value throughout the file so that we can easily tell which web server is serving our requests. Change the <title> value first:

{% set host = salt['grains.get']('host') -%}

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Welcome from {{ host }}</**title**>

. . .

Let's change the body text to this:

. . .

<**body**>

<**h1**>Welcome to nginx!</**h1**>

<**p**>Hello! This is being served from:</**p**>

<**h2**>{{ host }}</**h2**>

</**body**>

</**html**>

Save and close the file when you are finished.

Testing the Nginx State File

We've now completed our Nginx configuration. We can test certain aspects of the state in order to ensure that it works properly.

First, we can use the state.show\_sls execution module function to view how Salt will interpret our Nginx state file. We can use our stage-www1 server as the target. Nothing will execute on the server at this point though:

* sudo salt stage-www1 state.show\_sls nginx

You should get back output that looks something like this:

Output

stage-www1:

----------

/etc/nginx/nginx.conf:

----------

\_\_env\_\_:

base

\_\_sls\_\_:

nginx

file:

|\_

----------

source:

salt://nginx/files/etc/nginx/nginx.conf

|\_

----------

user:

root

|\_

----------

group:

root

|\_

----------

mode:

640

- managed

|\_

----------

order:

10002

. . .

It mainly renders the information from our /srv/salt/nginx/init.sls file with some interesting additions. Check that there are no interpretation errors where Salt did not know how to read commands. The "order" of each piece is another good item to check. This determines when each of the individual states in the file will run. The first state will have the order number "10000". Every additional state will count up from there. Note that the \_\_env\_\_ is different than the env we set using grains. We are not using Salt's conception of environments in this guide.

Next, we can do a dry-run of applying our state file. We can do this with the state.apply function with thetest=True option. The command looks like this:

* sudo salt stage-www1 state.apply nginx test=True

This will show you the changes that will be made if the test=True option is removed. Take a look to make sure that the changes make sense and that Salt is able to interpret all of your files correctly. The "Comment" field is particularly important as it can reveal issues even in cases where Salt did not mark the state as failed.

If the dry-run did not reveal any problems, you can try to apply the state to all of your available web servers by typing:

* sudo salt -G 'role:webserver' state.apply nginx

If you applied the Nginx state to your staging or production web servers, you'll want to get their internal IP addresses. The pages will not be available over the public interface:

* sudo salt-call mine.get 'role:webserver' internal\_ip expr\_form=grain

Output

local:

----------

stage-www1:

ip\_address

stage-www2:

ip\_address

If, on the other hand, you spun up your development web server and applied the Nginx state, you'll want to grab the external address since:

* sudo salt-call mine.get 'role:webserver' external\_ip expr\_form=grain

You can test your servers using curl:

* curl ip\_address

You should see the index.html page we modified:

Output

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Welcome from stage-www1</**title**>

<**style**>

**body** {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</**style**>

</**head**>

<**body**>

<**h1**>Welcome to nginx!</**h1**>

<**p**>Hello! This is being served from:</**p**>

<**h2**>stage-www1</**h2**>

<**p**><**em**>Thank you for using nginx.</**em**></**p**>

</**body**>

</**html**>

As you can see, the minion's host name was placed in the file when the Jinja was rendered. Our Nginx state is now complete.

Conclusion

You should now have a fully functional Nginx state. This will allow you to turn any Salt-controlled machine into a web server with your specifications quickly and easily. We will use this as a part of our larger infrastructure management strategy to easily construct the web servers in our environments.

In the [next guide](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-creating-salt-states-for-haproxy-load-balancers), we will move ahead and construct the state for the load balancers that will direct traffic in front of our web servers. We will use some of the same techniques that we used in this guide to make our load balancers flexible.

# SaltStack Infrastructure: Creating Salt States for HAProxy Load Balancers

Posted Oct 5, 2015

Top of Form

Bottom of Form

1

 4.3kviews [Configuration Management](https://www.digitalocean.com/community/tags/configuration-management?type=tutorials) [Ubuntu](https://www.digitalocean.com/community/tags/ubuntu?type=tutorials)

## Tutorial Series

This tutorial is part 5 of 6 in the series: [Managing Development Environments with SaltStack](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-creating-salt-states-for-haproxy-load-balancers#tutorial_series_53)

### Introduction

SaltStack, or Salt, is a powerful remote execution and configuration management system that can be used to easily manage infrastructure in a structured, repeatable way. In this series, we will be demonstrating one method of managing your development, staging, and production environments from a Salt deployment. We will use the Salt state system to write and apply repeatable actions. This will allow us to destroy any of our environments, safe in the knowledge that we can easily bring them back online in an identical state at a later time.

In our [previous guide](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-creating-salt-states-for-nginx-web-servers), we created a Salt state for our web servers which installed and configured Nginx. In this guide, we will configure states for the load balancer that will sit in front of our web servers in our staging and production environments. Our load balancers need to be configured with the web server addresses in order to correctly pass traffic.

Let's get started.

## Create the Main HAProxy State File

Our load balancers will use HAProxy to spread the traffic for our application between all of the available web servers in the environment. As with the Nginx state file, we will create a directory for this state in the/srv/salt directory:

* sudo mkdir /srv/salt/haproxy

We will use the name init.sls for our main state file within this directory so that we can refer to the state by the directory name:

* sudo nano /srv/salt/haproxy/init.sls

Inside, we can use the same pattern we used for Nginx in order to install the haproxy package and ensure that it is running. We will make sure the service is reloaded when there are changes to the package or changes to the /etc/default/haproxy file file or the /etc/haproxy/haproxy.cfg file. Again, be very careful with spacing to avoid YAML errors:

/srv/salt/haproxy/init.sls

haproxy:

pkg:

- installed

service.running:

- watch:

- pkg: haproxy

- file: /etc/haproxy/haproxy.cfg

- file: /etc/default/haproxy

We need to manage both of the files that the haproxy service is watching. We can create states for each.

The /etc/haproxy/haproxy.cfg file will be a template. This file will need to pull information about the environment in order to populate its list of web servers to pass traffic to. Our web servers will not have the same IPs each time that they're created. We will need to dynamically create the list each time this state is applied.

The /etc/default/haproxy file is just a regular file. We are managing it because we want to ensure that HAProxy is started at boot. This isn't dynamic information though, so we do not need to make this a template:

/srv/salt/haproxy/init.sls

haproxy:

pkg:

- installed

service.running:

- watch:

- pkg: haproxy

- file: /etc/haproxy/haproxy.cfg

- file: /etc/default/haproxy

/etc/haproxy/haproxy.cfg:

file.managed:

- source: salt://haproxy/files/etc/haproxy/haproxy.cfg.jinja

- template: jinja

- user: root

- group: root

- mode: 644

/etc/default/haproxy:

file.managed:

- source: salt://haproxy/files/etc/default/haproxy

- user: root

- group: root

- mode: 644

This is actually all we need for the state file itself. Save and close the file when you are done.

## Install HAProxy and Transfer Package Files to the Salt Master

We will use the same technique that we used with Nginx in order to get the basic HAProxy files we need. We will install the package on a minion and then tell the server to push the files back up to the master.

Let's use the stage-lb server since that will be the final target for this package anyways. If you don't already have your staging machines up and running, type:

* sudo salt-cloud -P -m /etc/salt/cloud.maps.d/stage-environment.map

Once your servers are available, you can install the haproxy package on the stage-lb server by typing:

* sudo salt stage-lb pkg.install haproxy

Once the installation is complete, we can tell the minion to push the two files we need up to the master server:

* sudo salt stage-lb cp.push /etc/default/haproxy
* sudo salt stage-lb cp.push /etc/haproxy/haproxy.cfg

The relevant portions of the minion filesystem will be recreated in the/var/cache/salt/master/minions/minion\_id/files directory. In this case, the minion ID is stage-lb. Copy the entire minion file structure to our HAProxy state directory:

* sudo cp -r /var/cache/salt/master/minions/stage-lb/files /srv/salt/haproxy

We can see the file structure by typing:

* find /srv/salt/haproxy -printf "%P\n"

Output

files

files/etc

files/etc/default

files/etc/default/haproxy

files/etc/haproxy

files/etc/haproxy/haproxy.cfg

init.sls

Now that we have the files from the minion, we can destroy the load balancing server:

* sudo salt-cloud -d stage-lb

We can then recreate the server in the background so that we have a clean slate later to do our final testing and confirmation. Target your Salt master server with this command, since it has access to the relevant cloud files:

* sudo salt --async sm cloud.profile stage-lb stage-lb

While the server is rebuilding, we can move on and make the necessary modifications to the HAProxy files we are managing.

## Configure the /etc/default/haproxy File

We can start with the /etc/default/haproxy file. In our HAProxy state directory on the Salt master, move to the directory that houses the default file:

* cd /srv/salt/haproxy/files/etc/default

Copy the file to haproxy.orig so that we can preserve the file as it was originally packaged:

* sudo cp haproxy haproxy.orig

Now, open the file for editing:

* sudo nano haproxy

Change ENABLED to "1". This will tell Ubuntu's init system, Upstart, to start the HAProxy service when the server boots:

/srv/salt/haproxy/files/etc/default/haproxy

# Set ENABLED to 1 if you want the init script to start haproxy.

ENABLED=1

# Add extra flags here.

#EXTRAOPTS="-de -m 16"

This is the only change we need to make. Save and close the file.

## Configure the /etc/haproxy/haproxy.cfg Template File

Next, let's work on the main HAProxy configuration file. Move into the appropriate directory on the Salt master server:

* cd /srv/salt/haproxy/files/etc/haproxy

Again, let's copy the configuration to save it's original state:

* sudo cp haproxy.cfg haproxy.cfg.orig

Then, rename the file to reflect that this is a Jinja template file:

* sudo mv haproxy.cfg haproxy.cfg.jinja

Open the template file in your text editor:

* sudo nano haproxy.cfg.jinja

At the top of the file, we can start by setting a Jinja variable. We need to grab the environment that the load balancer is operating in using the network.interface\_ip execution function. We will use this later to populate the server list with the web servers from the same environment:

/srv/salt/haproxy/files/etc/haproxy/haproxy.cfg.jinja

{%- set env = salt['grains.get']('env') -%}

global

log /dev/log local0

log /dev/log local1 notice

chroot /var/lib/haproxy

. . .

Skip down to the "defaults" section of the file. We need to change mode to "tcp" and the first option to "tcplog":

/srv/salt/haproxy/files/etc/haproxy/haproxy.cfg.jinja

. . .

defaults

. . .

mode tcp

option tcplog

. . .

At the bottom of the file, we need to create our actual configuration. We need to create a "frontend" section, which will describe how HAProxy will accept connections. We will label this section "www".

We want to bind this to the server's public IP address. We can grab this using the network.interface\_ipexecution module function with the eth0 argument. Web requests will come in at port 80. We can specify the default backend to pass to with the default\_backend option. We will call our backend nginx\_pool:

/srv/salt/haproxy/files/etc/haproxy/haproxy.cfg.jinja

. . .

frontend www

bind {{ salt['network.interface\_ip']('eth0') }}:80

default\_backend nginx\_pool

Next, we need to add the nginx\_pool backend. We will use the conventional round robin balancing model and set the mode to "tcp" again.

After that, we need to populate the list of backend web servers from our environment. We can do this using a "for" loop in Jinja. We can use the mine.get execution module function to get the value of theinternal\_ip mine function. We will match the web server role and the environment. The ~ env will catenate the value of env variable we set earlier to the match string that precedes it.

The results of this lookup will be stored in the server and addr variables for each iteration of the loop. Within the loop, we will add the server's details using these loop variables. The final result looks like this:

/srv/salt/haproxy/files/etc/haproxy/haproxy.cfg.jinja

. . .

frontend www

bind {{ salt['network.interface\_ip']('eth0') }}:80

default\_backend nginx\_pool

backend nginx\_pool

balance roundrobin

mode tcp

{% for server, addr in salt['mine.get']('G@role:webserver and G@env:' ~ env, 'internal\_ip', expr\_form='compound').items() -%}

server {{ server }} {{ addr }}:80 check

{% endfor -%}

Save and close the file when you are finished.

## Testing the HAProxy State File

Our load balancing state is fairly basic, but complete. We can now move on to testing it.

First, let's use state.show\_sls to display the file ordering:

* sudo salt stage-lb state.show\_sls haproxy

We can tell by the sequence in the various "order" values in the output that the package will be installed, the service will be started, and then the two files will be applied. This is what we expected. The file changes will trigger a service reload due to the "watch" setting we configured.

Next, we can do a dry run of the state application. This will catch some (but not all) errors that would cause the state to fail when run:

* sudo salt stage-lb state.apply haproxy test=True

Check that all of the states would have passed. Regardless of the failure count at the bottom or the output, scroll up and look at the "Comment" line for each of the states. Sometimes, this will include extra information about potential issues, even though the test was marked as successful.

After fixing any issues that have surfaced during the test commands, you can apply your state to your load balancer servers. Make sure that you have the backend Nginx web servers running and configured prior to applying the state:

* sudo salt-cloud -P -m /etc/salt/cloud.maps.d/stage-environment.map
* sudo salt -G 'role:webserver' state.apply nginx

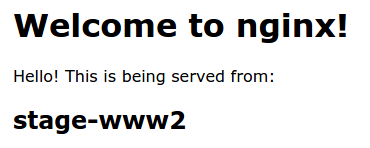
When your web servers are running, apply the haproxy state:

* sudo salt -G 'role:lbserver' state.apply haproxy

You should now be able to get to one of your two backend web servers through your load balancer's public IP address. You can display your load balancer's public IP address with this command:

* sudo salt -G 'role:lbserver' network.interface\_ip eth0

If you use the browser, it will look something like this:



It is easier to see the load balancer pass traffic between the backend servers with curl:

* curl load\_balancer\_public\_IP

Output

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Welcome from stage-www2</**title**>

<**style**>

**body** {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</**style**>

</**head**>

<**body**>

<**h1**>Welcome to nginx!</**h1**>

<**p**>Hello! This is being served from:</**p**>

<**h2**>stage-www2</**h2**>

</**body**>

</**html**>

If you type the command again a few times, it should swap between your two servers:

* curl load\_balancer\_public\_IP

Output

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Welcome from stage-www1</**title**>

<**style**>

**body** {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</**style**>

</**head**>

<**body**>

<**h1**>Welcome to nginx!</**h1**>

<**p**>Hello! This is being served from:</**p**>

<**h2**>stage-www1</**h2**>

</**body**>

</**html**>

As you can see, the server serving the request has changed, meaning that our load balancer is functioning correctly.

## Conclusion

At this point, we have a functioning HAProxy state that can be applied to our load balancer machines. This can be used to split the incoming traffic for our application among all of the backend Nginx servers. We can easily destroy our load balancers and then rebuild them based on the web servers available.

In the [next guide](https://www.digitalocean.com/community/tutorials/saltstack-infrastructure-creating-salt-states-for-mysql-database-servers), we will focus on getting MySQL up and running as our backend database system. This will be used to store application data in our various environments.