Salt Cookbook

Salt has become one of the major players in automation and configuration management solutions. This book starts with the basics of the tool, the procedures to get up and running with Salt and then moves on to configuring very simple but important details to receive optimal performance from the tool. It also walks you through Salt configurations for different infrastructure components and the details of the Salt modules for each of the components. The book also provides some common problem scenarios and how to troubleshoot them.

With detailed configuration, their explanation and command line outputs of the module execution, Salt Cookbook will help you to get up and running with Salt for all your infrastructural needs.

What this book will do for you...

- Understand the basic Salt architecture and components and learn how to configure them
- Understand and configure advanced Salt components and manipulate configurations
- Configure Salt modules, the orchestration procedure and set up multi-master configurations
- Configure operating system components such as files, users, volumes, authentication, and so on
- Leverage the cloud capabilities of Salt
- Understand and configure the Salt event and reactor system

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Apply solutions to other real-world situations

Salt Cookbook

Over 80 hands-on recipes to efficiently configure and manage your infrastructure with Salt

Anirban Saha

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Salt Architecture and Components'
- A synopsis of the book’s content
- More information on Salt Cookbook
Anirban Saha has been a system engineer for several years, mostly working with Linux and cloud-based infrastructures. He has worked extensively on large system deployments in data center environments, as well as public and private cloud infrastructures for several organizations in various industries. He is also an open source and technology enthusiast and a speaker at conferences and meetups. Anirban likes and supports anything about technology.
Preface

Salt is one of the many but unique tools available today for configuration management and orchestration. Salt not only enables us to seamlessly configure our infrastructure, but also to perform all sorts of tasks based on a variety of conditions and properties of the Salt minions. Along with super-fast module execution with the help of the ZeroMQ messaging library, Salt takes orchestration to a whole new level, which can be used for numerous day-to-day tasks starting from simple data collection to complex deployments.

This book provides a hands-on approach to the world of Salt. It provides a basic understanding of the concepts and architecture on which Salt is based, and then moves on to details about configuring it and also writing complex recipes for different tasks and scenarios.

The content of the book definitely helps professionals associated with infrastructure management to consider Salt as a potential tool for implementing their infrastructure and it also provides lot of recipes which existing Salt users can refer to. With the help of the content of the book, users can also integrate Salt with other tools and services to get the optimal results required.

The book starts off with the concepts and architecture of Salt, followed by some basic and advanced configurations and module usages. The next topics are about configuring application and database servers, cloud-based services, and event systems. It finishes off with recipes for troubleshooting Salt.

What this book covers

Chapter 1, Salt Architecture and Components, gives you a basic understanding about the concept and architecture of Salt. It explains a lot of Salt terminology, such as the Salt master, minions, states, pillars, environments, and grains, and demonstrates how to configure them.
Preface

Chapter 2, Writing Advanced Salt Configurations, goes a bit deeper into Salt configuration and gives you a detailed explanation of the various data manipulation you can perform in Salt, such as conditionals, iterations, and Python functions. It also focuses on the very important topic of targeting minions using different techniques.

Chapter 3, Modules, Orchestration, and Scaling Salt, gives you an even deeper understanding of the advanced Salt concepts of execution and state modules, using templates, requisites, and Salt runners, performing orchestration and demonstrates multi-master setups for redundancy.

Chapter 4, General Administration Tasks, provides hands-on recipes on configuring general system entities, such as users, groups, SSH configurations, scheduling with cron, volume, and disk management with network configuration. It also explains how to add custom grains to a minion.

Chapter 5, Advanced Administration Tasks, provides explanations and demonstrations about advanced system topics such as package repository configuration and package management, file and service management, managing code repositories and handling alternatives.

Chapter 6, Managing Application Servers, gives an understanding of how to configure web and application servers by handling each configuration step-by-step in the recipes. It demonstrates package and service configuration, manages the server configuration file, and deployments for application servers and security for web servers.

Chapter 7, Managing Databases, provides detailed explanations and demonstrations of database configuration for MySQL and PostgreSQL. It includes database package and service management, managing users and groups, databases, grants and queries.

Chapter 8, Configuring Salt Cloud, provides extensive knowledge about how to manage cloud infrastructures using Salt Cloud. It demonstrates the usage of Salt Cloud, providers and profiles, cloud maps, post-install scripts, and management of cloud resources such as instances and volumes using Salt Cloud.

Chapter 9, Managing Amazon Web Services, focuses specifically on how to manage the EC2 Cloud hosted by Amazon Web Services. It includes recipes to manage security groups, load balancers, DNS with Route53, Simple Queue Service, ElastiCache clusters and CloudWatch alarms.

Chapter 10, Salt Event and Reactor System, explains and demonstrates the event system of Salt and how to request and receive events using the Salt message bus. It also explains how to use the event system to configure the reactor system in Salt to perform required tasks.

Chapter 11, Troubleshooting, demonstrates several scenarios where things can go wrong with the Salt system and how you can troubleshoot and fix such problems.
In this chapter, you will cover:

- Installing and configuring the Salt master
- Configuring the Salt environment and pillar paths
- Understanding and configuring Salt pillars
- Understanding and writing Salt states
- Understanding and writing the top file
- Installing and configuring the Salt minion
- Configuring environment and grains on the minion
- Applying Salt states to minions

Introduction

So you have heard of this new tool called Salt, you’ve probably looked it up on the Internet, and now it has drawn your attention. What is Salt?

Salt is a systems management software created and maintained by SaltStack. It is used to manage infrastructure, virtualization, applications, databases, private and public clouds, software, or code. With Salt, data center automation, server provisioning, configuration management, and orchestration are a few tasks that can be done in a simple, elegant, and fast manner. Salt was first released in 2011 and has been gaining tremendous momentum ever since.
Salt Architecture and Components

The primary advantages of Salt are as follows:

- **Simple to set up and manage**: No matter what the size of the infrastructure is, Salt is extremely easy to set up and maintain. It has a simple server/client model, which works seamlessly from a small deployment to massive data center deployments.

- **Ability for parallel execution**: Salt uses hostname, system properties called grains and custom configured matchers to perform remote module, state, and command execution on target nodes. These executions happen in parallel for rapid deployments.

- **Speed and security**: Salt incorporates a message broker in the Salt daemon itself by using the ZeroMQ networking library. It uses public keys for authentication between the master and the client nodes with AES encryption for faster payload communication. Salt enables light and fast network traffic using msgpack.

- **Scalability and flexibility**: Salt provides flexibility for systems management by providing various ways for the master and client nodes to communicate and synchronize using both the push and pull methods. The Salt master can also be scaled with multiple master configurations to support large and growing infrastructures.

- **Uses Python interface**: Salt routines can be written as Python modules, and Salt itself can be called from the command line or Python API. The Salt states even allow the use of Python style conditionals, iterators, and functions to carry out automated systems management tasks.

Apart from the few aforementioned advantages, Salt has many more features that we will explore throughout this book. Salt has a large and vibrant community, which has come up with a massive number of built-in state and execution modules to simplify the management and deployment process as much as possible.

It is to be noted that almost all of the examples mentioned in this book have been targeted to work on the CentOS operating system, unless specified otherwise. The methods for other operating systems should be similar with different package, file, and service names.

Already excited by the possibilities of Salt? Let's dive in and explore it hands on.

**Installing and configuring the Salt master**

In this section, we are going to configure the most important component of the Salt architecture—the Salt master. We'll install the Salt master package and configure the most important parameters needed for our infrastructure.
How to do it...

Let's see how we can install the Salt master on various types of OS.

Installing the Salt master on RedHat/CentOS/Fedora

1. Salt packages are available in the EPEL repository. First, the repository needs to be added to the system. As the system being used is CentOS 6.5 (64-bit), we are using the `epel-release` package at `http://dl.fedoraproject.org/pub/epel/6/x86_64/`. This needs to be changed as per the version and architecture of the operating system being used:

   [root@salt-master ~]# rpm -ihv \  
   http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm

2. After the EPEL release package has been installed, we will install the `salt-master` package with the following command, and the dependencies should automatically be fetched from the repository:

   [root@salt-master ~]# yum -y install salt-master

Installing the Salt master on Ubuntu

While installing the Salt master on Ubuntu, the SaltStack PPA repository needs to be added to the system. It is to be noted that the following commands need to be executed as a privileged user, that is, either the `root` user can be used, or the `sudo` command needs to be added before the mentioned commands:

1. The following command adds the `add-apt-repository` binary to the system:

   [root@salt-master ~]# apt-get -y install python-software-properties

2. Now, we will add the repository with the command given here:

   [root@salt-master ~]# add-apt-repository ppa:saltstack/salt

3. The Salt master package then needs to be installed with the following command:

   [root@salt-master ~]# apt-get -y install salt-master

Configuring the Salt master

The primary configuration file for the Salt master is `/etc/salt/master`. It is also a good practice to create additional configuration files in `/etc/salt/master.d/` with the `.conf` extension, and they will get read along with all the other files when the Salt master daemon starts.
Most of the Salt configuration parameters are set by default and need not be set explicitly. However, let's look at some of the important parameters that can be altered to suit one's needs:

- To determine which network interface the service binds to:
  ```
  interface: 0.0.0.0
  ```

- The port on which to listen to for client node (minion) communications:
  ```
  publish_port: 4505
  ```

- The path that gets prepended to other files such as `log_file`, `pki_dir`, and `cache_dir` if set. It is also to be noted that this path gets prepended to all other defined configuration parameters in the master configuration files, where each of them is also explained in detail:
  ```
  root_dir: /
  ```

- The directory to hold the master and minion keys that have already been authenticated or rejected:
  ```
  pki_dir: /etc/salt/pki/master
  ```

- The file containing log entries for the master daemon:
  ```
  log_file: /var/log/salt/master
  ```

- The file that allows the keys of the host's that match the listed patterns to be accepted automatically (it is always a good practice to define this file). We will uncomment this line and set the filename as follows:
  ```
  autosign_file: /etc/salt/autosign.conf
  ```

- Edit the file `/etc/salt/autosign.conf` and set the content to be a wild card entry as follows (this is being done to facilitate easier demonstrations in the rest of the book, it is to be noted that this is a security risk otherwise):
  ```
  *
  ```

- The Salt service daemon then needs to be started and configured to start at boot time.
  On RedHat/CentOS/Fedora:
  ```
  [root@salt-master ~]# service salt-master start
  [root@salt-master ~]# chkconfig salt-master on
  ```

  On Ubuntu, the installation process automatically starts the daemon, hence the daemon needs to be restarted:
  ```
  [root@salt-master ~]# service salt-master restart
  [root@salt-master ~]# update-rc.d salt-master defaults
  ```
The firewall needs to be configured to allow communication on ports 4505 and 4506 from the minions:

```bash
[root@salt-master ~]# iptables -A INPUT -m state --state new -m tcp -p tcp --dport 4505 -j ACCEPT
[root@salt-master ~]# iptables -A INPUT -m state --state new -m tcp -p tcp --dport 4506 -j ACCEPT
```

- Save the firewall rules:
  - On RedHat/CentOS/Fedora:
    ```bash
    [root@salt-master ~]# service iptables save
    ```
  - On Ubuntu:
    ```bash
    [root@salt-master ~]# iptables-save
    ```

In the scenario that a virtualized environment is being used, such as a cloud provider, the aforementioned ports should be opened in the respective security group of the master node.

### How it works...

The **salt-master** is the package for the Salt master service and it also requires a few other dependencies, such as the ZeroMQ library, msgpack, jinja, yaml, and so on, which is automatically pulled along with the package from the configured repositories.

Most of the Salt configuration parameters are set by default and need not be explicitly mentioned in the file. The options can be found commented in the file and act as the defaults. However, if they need to be changed, then they can be uncommented and necessary changes can be made.

We have explicitly uncommented the **autosign_file** parameter and set the value as `/etc/salt/autosign.conf`:

```bash
autosign_file: /etc/salt/autosign.conf
```

We then populated the file with a wildcard entry, that is, `*`, to allow all minions' certificate requests to be automatically signed and accepted by the master.

Finally, the service daemons for salt master are started/restarted, configured to start automatically at boot time, and firewalls are configured to open the ports 4505 and 4506 for communication with the minions using the system-specific commands.
See also

- The Salt multi-master setup (active-active mode) and Salt multi-master setup (active-passive mode) recipes in Chapter 3, Modules, Orchestration, and Scaling Salt, to learn more about highly available and redundant Salt master setups
- The Configuring the Salt environment and pillar paths recipe, for advanced configuration of the Salt master

Configuring the Salt environment and pillar paths

In this recipe, we are going to configure the environment and pillar paths for Salt to use when we mention environments while performing Salt operations. Ideally, any infrastructure should have environments for development, testing, QA, production, and so on, for isolation of respective code and a seamless workflow. Salt enables us to implement this in a few simple steps by providing options to define environments for states and pillars.

How to do it...

In the primary configuration /etc/salt/master file, we will configure the section called file_roots. Under this parameter option, we will name each environment we want to configure and the path of the directory that will contain the configuration files for that environment.

1. Let's create the directories we want to use as the environment paths and then configure four environments named base, production, staging, and development as follows:
   
   ```bash
   [root@salt-master ~]# mkdir -p /opt/salt-cookbook/{base,production,staging,development}
   ```

2. Edit /etc/salt/master to have following content:

   ```yaml
   file_roots:
   base:
     - /opt/salt-cookbook/base
   production:
     - /opt/salt-cookbook/production
   staging:
     - /opt/salt-cookbook/staging
   development:
     - /opt/salt-cookbook/development
   ```
3. Next, we will create the directories we want to use as pillar paths and configure them following the same environment conventions as follows:

   [root@salt-master ~]# mkdir -p \\
   /opt/salt-cookbook/pillar/{base,production,staging,development}

4. Edit /etc/salt/master to have following content:

   pillar_roots:
   base:
   - /opt/salt-cookbook/pillar/base
   production:
   - /opt/salt-cookbook/pillar/production
   staging:
   - /opt/salt-cookbook/pillar/staging
   development:
   - /opt/salt-cookbook/pillar/development

5. We now restart the salt-master daemon for these changes to take effect:

   [root@salt-master ~]# service salt-master restart

---

**How it works...**

Environments in Salt are designed to keep the configuration files for each environment isolated into respective containers for different stages of deployment such as development, QA, staging, and production. Ideally, the configurations can be written in either of the environments such as staging, and after a proper test phases, they can be migrated to the production environment.

Here, we opted for the /opt/salt-cookbook path to be the base directory of all our configurations. Next, we have chosen four environments for our infrastructure, namely, base, staging, production, and development. The base environment is the default in Salt and the path /srv/salt is taken as the base directory for all configurations if not configured explicitly as we have done earlier.

We then created the directory paths we want to use as the environment paths and mentioned them under the file_roots parameter, specifying the environment name and the path that points the configuration files for that environment:

   file_roots:
   <environment_name>:
   - <environment_directory_path>

**Pillars** are one of the best features of Salt, which enable us to isolate data that needs to be kept secure such as keys and passwords. Next, we have chosen the /opt/salt-cookbook/pillar path to be our base directory for all pillar data and created directories for each environment that will contain pillar data, which is usable in that environment only.
We then edited the Salt master configuration file to include the environment names and their paths under the `pillar_roots` section for Salt to identify them as the pillar configuration paths:

```
pillar_roots:
  <environment_name>:
    - <pillar_directory_path>
```

Finally, we restarted the Salt master daemon for the changes to take effect.

It is also to be noted that when executing the `salt` command, the environment has to be mentioned for the minion(s). This is generally done with the `saltenv` option on the command line. However, this parameter can also be specified in the minion configuration `/etc/salt/minion` file, as follows:

```
environment: production
```

If this parameter is not set, the base environment is taken as the default environment.

See also

- The Using pillar data in states and Writing and retrieving pillar data recipes in Chapter 2, Writing Advanced Salt Configurations, to learn more about how to use pillars in states
- The Understanding and configuring Salt pillars recipe, for detailed examples of pillar configurations

### Understanding and configuring Salt pillars

In this recipe, we will learn about pillars, how they fit into the Salt architecture, and how to configure them. We will create pillar data for a user state, which we will configure later using this pillar data.

#### How to do it...

We will create pillar data for the development environment.

1. First, we will create a directory for the name of the pillar data (usually it is named the same as the state we are configuring this pillar for, that is, if we are configuring this pillar for the user state, we will name this pillar directory user:

   ```
   [root@salt-master ~]# mkdir \
   /opt/salt-cookbook/pillar/development/user
   ```
2. Edit /opt/salt-cookbook/salt/pillar/development/user/init.sls and add the following content:

```bash
dev_user:
    name: thomas
    password: "$1$PGlinys0$kB2I83KzEVzVs9G7xLHjAl"
    uid: 1001
    comment: "Thomas"
```

And run the following command:

```bash
[root@salt-master ~]# touch \
/opt/salt-cookbook/pillar/development/top.sls
```

3. Edit /opt/salt-cookbook/pillar/development/top.sls and add the following content:

```bash
development:
  '*':
    - user
```

How it works...

In Salt, a pillar is a feature used to store data such as keys and passwords or any other type of data such as repetitive directory paths or usernames that are then accessed from states. All minion-specific data to be seen only by the minion is stored in pillars and is visible to the minion for which the data is meant and configured. In this recipe, we created pillar data in the development environment for a user state, which we will configure in the next recipe.

First, we created a directory similar to the name of the state that we will configure in the pillar directory path of the development environment:

```bash
[root@salt-master ~]# mkdir \
/opt/salt-cookbook/pillar/development/user
```

Next, we created a file called init.sls in the directory created earlier, where we will create the pillar data. The default file in a pillar directory is named init.sls, where .sls is the file extension for all Salt state, pillar, and top files. Salt manages the SLS files internally. If the user pillar is referenced, we must understand that it's the content of the init.sls file in the user directory that is being referred to. The contents of all SLS files in Salt are in the YAML format, and indentations are very important both for parsing by Salt and to keep the configurations organized.
Salt Architecture and Components

In the `init.sls` file, we populated the pillar data that we need. Basically, we configured the first user of a user list of developers mentioning various parameters for the user, such as their username, user ID, password in hash format (note that it is enclosed in quotes to avoid problems with special characters), and the comment:

```yaml
dev_user:
  name: thomas
  password: "$1$PG1inys0$kB2I83KzEVzVs9G7xLHjA1"
  uid: 1001
  comment: "Thomas"
```

If we are planning to add multiple users, the format can be as shown here:

```yaml
dev_user:
  thomas:
    password: "$1$PG1inys0$kB2I83KzEVzVs9G7xLHjA1"
    uid: 1001
    comment: "Thomas"
```

The methods to parse this kind of YAML definition will be discussed later in the book.

Next, we created a file called `top.sls` in the base pillar directory of the development environment. The contents of the `top.sls` file determine which nodes or minions will have access to which pillar data. We created the following contents in the file:

```yaml
development:
  '*':
    - user
```

The first line mentions the environment. Without this line, the base environment will be used as the default. The `*` operator in the second line is a wildcard, which means that all minions will have access to the `user` pillar data. This line can be manipulated to add various types of matchers to target minions that will be discussed later in the book. The third line mentions the name of the pillar directory that we created. If only the name of the directory is mentioned, we have to understand that the contents of the `init.sls` files are being referred to. If we create a file called `devs.sls` in the `user` directory, then the contents of that file can be mentioned in the `top.sls` file as `user.devs`, shown as follows:

```yaml
development:
  '*':
    - user.devs
```

When the states are run on the minions, this file is checked and these definitions determine if that particular minion is allowed to access the contents of this pillar.
See also

- The Writing and retrieving pillar data and Using pillar data in states recipes in Chapter 2, Writing Advanced Salt Configurations, to learn more about using pillars
- The Understanding and writing Salt states recipe, to learn how to use this pillar data in states

Understanding and writing Salt states

In this recipe, we will learn one of the most important components of Salt states. We will understand what states are, how to write them, and how to use pillar data in writing states. We will configure a state called user to add a user to the development environment.

How to do it...

1. First, we will create a directory for the state and populate it with state data:

   ```bash
   [root@salt-master ~]# mkdir /opt/salt-cookbook/development/user
   [ root@salt-master ~]# touch /opt/salt-cookbook/development/user/init.sls
   ```

2. Edit `/opt/salt-cookbook/development/user/init.sls` and add the following contents:

   ```sls
   generic_user:
   user.present:
    - name: {{ pillar['dev_user']['name'] }}
    - shell: /bin/bash
    - home: /home/{{ pillar['dev_user']['name'] }}
    - uid: {{ pillar['dev_user']['uid'] }}
    - password: {{ pillar['dev_user']['password'] }}
   ```

How it works...

In Salt, states are written to determine which and what kind of resources will be added to a system. States are written with the help of built-in state modules, a full list of which can be found at http://docs.saltstack.com/en/latest/ref/states/all/. Using these state modules, we can configure a massive amount of system resources, a few being users, groups, files, repositories, packages, services, code repositories, and so on. These states are then configured to be available to desired nodes or minions based on various properties.

In this recipe, we configured a user to be added to a system in the development environment, irrespective of the operating system.
Salt Architecture and Components

First, we created a directory with the name of the state, that is, the user in the base directory of the development environment. Then, we created a file named init.sls, which is the default file in a state directory. Salt State File or SLS is used as the only file type of all Salt configuration files.

In the first line, we defined a custom name of this resource that can be any name, we named it generic_user. This name can be used in subsequent resources, when referencing this resource and have to be unique in the complete state space being used by the minion. However, this is optional as the definition could also be written as follows:

```yaml
{{ pillar['dev_user']['name'] }}:
  user.present:
    - shell: /bin/bash
    - home: /home/{{ pillar['dev_user']['name'] }}
```

Here, we also learn about how to refer to pillar data when writing states. It is usually in the following format:

```yaml
{{ pillar<first_key>[<second_key>]... }}
```

However, do note that, in the reference, the directory name of the pillar, that is, user is never used. Instead, the reference begins from the first YAML key found in the file, that is, dev_user, followed by the next mentioned key. Also, note that the first keys are always unique in the environment for a minion, that is, dev_user can only be used once as the first key. If a second file is created, called devs.sls, and the first key is named dev_user, there will be a conflict of key names and Salt synchronization will fail.

The following method can also be used to achieve the same objective:

```yaml
{{ pillar['pillar.get']['dev_user:name', 'defaultuser'] }}:
```

This is a better method as it also gives us the option to mention a default value if the pillar is not available.

The second line mentions the built-in state module name, that is, user and the function is present. There can be multiple functions for a module name and this can be looked up in the URL mentioned earlier in this recipe. For the user module, the two functions available are present and absent, which in turn take multiple parameters such as name, uid, gid, password, and so on.

Next, we see how each of the function parameters are populated for the user using static data, as well as pillar data:

```yaml
- name: {{ pillar['dev_user']['name'] }}
- shell: /bin/bash
- home: /home/{{ pillar['dev_user']['name'] }}
- uid: {{ pillar['dev_user']['uid'] }}
```
See also

- Chapter 2, Writing Advanced Salt Configurations, to learn more about advanced state configurations
- The Understanding and writing the top file recipe, to learn about how to make Salt states available to minions

Understanding and writing the top file

In this recipe, we will understand and learn how to write the top file, the file which determines how the states are applied to the nodes or minions in Salt. We will apply the state that we configured in the previous recipe to minions.

How to do it...

The state configured in the previous recipe will now be applied to minions in the development environment.

1. Create a file named top.sls in the base directory of the development environment:

   ```
   [root@salt-master ~]# touch /opt/salt-cookbook/development/top.sls
   ```

2. Edit the /opt/salt-cookbook/development/top.sls file and add the following contents:

   ```yaml
   development:
   '***':
   - user
   'salt-minion':
   - match: list
   - hostconfig
   ```

How it works...

The top.sls file in Salt determines which state will be applied to which minions. It has an extension of .sls similar to all other files in Salt. It is written in the YAML format and takes minion matchers in the form of wildcards, nodegroups, lists, grains, and so on. We will look at all of the ways to match minions throughout the book when we go through each of these components.
First, we created the `top.sls` file in the base directory of the development environment, that is, `/opt/salt-cookbook/development`, and then we populated it with the required definitions.

The first line indicates the environment we are configuring the top file for, that is, `development`. Without this, the base environment will be used as default. The next line indicates a wildcard `*` meaning that it will match all minions and apply the listed state to all of them. Then, we listed the `user` state that we already configured:

```
development:
  '*':
    - user
```

We can also match minions when applying states in the `top.sls` file in the following manner:

```
development:
  '*':
    - user
  'salt-minion':
    - match: list
    - hostconfig
```

Here, we see a second block of node definition and state that introduces a new key called `match`. Instead of the wildcard in the preceding block, this block has the name of a minion node called `salt-minion`, which is the hostname of the minion node that will synchronize with the master and the second line in the block:

```
  - match: list
```

This line indicates that the type of minion parameter to match is a list, which is nothing but the hostname of the minion. The line that has the name of the minion can be a comma-separated line of multiple hostnames:

```
  'webserver,dbserver,appserver,proxyserver'
```

The list matcher will match the line as a list of minion names. Then, we mentioned a state called `hostconfig`, which is also the name of a state not configured in the book so far. This example means that the `hostconfig` state will be applied to all the minions in the list, which in this case is the minion named `salt-minion`.

**There's more...**

In Salt, minions can be matched in more ways than shown in this recipe, such as nodegroups, grains, IP/subnets, and so on. We will learn how to apply them in Chapter 3, *Modules, Orchestration, and Scaling Salt.*
See also

- The *Installing and configuring the Salt minion* recipe, to learn how to install and configure minions

## Installing and configuring the Salt minion

In this recipe, we will learn about minions, how they work, and how to install and configure them.

### How to do it...

We will install the Salt minion on a second node and name the node *salt-minion*. First, we will install the *salt-minion* package.

#### Installing the Salt minion on RedHat/CentOS/Fedora

1. Install the *epel-release* rpm to configure the EPEL repository:

   ```shell
   [root@salt-minion ~]# rpm -ihv http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
   ```

2. After the EPEL release package has been installed, we will install the *salt-minion* package with the following command and the dependencies should automatically be fetched from the repository:

   ```shell
   [root@salt-minion ~]# yum -y install salt-minion
   ```

#### Installing the Salt minion on Ubuntu

When installing the Salt minion in Ubuntu, the SaltStack PPA repository needs to be added to the system. It is to be noted that the following commands need to be executed as a privileged user, that is, either the *root* user can be used, or the *sudo* command needs to be added before the mentioned commands.

1. The following command adds the *add-apt-repository* binary to the system:

   ```shell
   [root@salt-minion ~]# apt-get -y install python-software-properties
   ```

2. Now, we will add the repository with the following command:

   ```shell
   [root@salt-minion ~]# add-apt-repository ppa:saltstack/salt
   ```

3. The Salt minion package then needs to be installed with the following command:

   ```shell
   [root@salt-minion ~]# apt-get -y install salt-minion
   ```
Configuring the Salt minion

The configuration file for the Salt minion is /etc/salt/minion. It is also a good practice to create additional configuration files in /etc/salt/minion.d/ with the .conf extension, and they will get read along with all other files when the Salt minion daemon starts.

1. In /etc/salt/minion, uncomment and edit the following parameter:
   master: salt-master

2. Start the salt-minion service daemon and configure it to start automatically at boot time:
   On RedHat/CentOS/Fedora:
   [root@salt-minion ~]# service salt-minion start
   [root@salt-minion ~]# chkconfig salt-minion on
   
   On Ubuntu, the installation process automatically starts the daemon, hence the daemon needs to be restarted:
   [root@salt-minion ~]# service salt-minion restart
   [root@salt-minion ~]# update-rc.d salt-minion defaults

How it works...

In Salt, the client nodes on which the configured states are applied are known as minions. The minion configuration file is /etc/salt/minion, which contains the configurable parameters for the minion, and most of the parameters use the default values as mentioned in the file and are commented. Any change in the parameters can be made by uncommenting and editing the parameters.

In this recipe, we configured the repositories from which to fetch the salt-minion package. We then installed the package and the dependencies get automatically fetched from the repositories.

Next, we edited the /etc/salt/minion file, uncommented the master parameter, and edited it to have the value salt-master (as we named our salt master salt-master in the earlier recipes). Do note that, for this parameter to work on the minions, there needs to be a DNS entry or an entry in /etc/hosts of the minions for the salt-master pointing to the IP of the Salt master. This parameter enables the minion to communicate with the Salt master on the TCP ports 4505 and 4506, as we configured in the first recipe of this chapter.

The rest of the parameters can be left as defaults. Finally, we started the salt-master daemon and configured it to start automatically at boot time.
There's more...

There is one more parameter in the /etc/salt/minion file, which may be interesting for some. The id parameter enables us to set the minion ID explicitly, without which the hostname of the minion is taken up as the minion ID by default. The minion ID is the name that the Salt master is able to see when the minion requests its certificate to be signed and authenticated by the master. After the certificate signing is complete, the master knows the minion by the same minion ID and any communication which involves the minion name from the master to the minion happens using this minion ID.

For example, the minion node can have a hostname prodapp01, but we can set its minion ID in the /etc/salt/minion file, as follows:

```
id: appserver
```

This parameter will make this node known to the Salt master as appserver and not prodapp01, although the hostname is prodapp01. Without this parameter, the master will know this node as prodapp01.

The masterless minion

There is a feature in Salt that enables the minions to run in a masterless mode. In this case, the minion acts as its own master. The minion can be configured for this by changing the value of the file_client parameter in the /etc/salt/minion file from remote to local and configuring the paths to states and pillars.

Uncomment and edit the following parameters in /etc/salt/minion. The value of local for the file_client parameter enables the minion to look for states locally on it:

```
file_client:  local
```

Next, the state and pillar paths need to be set for the minion to find the state and pillar data on the system. The state and pillar files then need to be placed in these directories as discussed in earlier recipes:

```
file_roots:
  base:
    - /opt/salt-cookbook/base
  development:
    - /opt/salt-cookbook/development

pillar_roots:
  base:
    - /opt/salt-cookbook/pillar/base
  development:
    - /opt/salt-cookbook/pillar/development
```
Salt Architecture and Components

We will learn the technique to run Salt to get the state and pillar data in the last recipe of this chapter.

See also

- The Installing and configuring the Salt master and Configuring the Salt environment and pillar paths recipes, to learn about the master, its port configurations, state, and pillar configurations
- The Configuring environments and grains on the minion and Applying Salt states to minions recipes, to learn about advanced minion configurations and synchronizing minions with masters

Configuring environments and grains on the minion

In this recipe, we will learn about some advanced minion configurations, such as environments and grains. We will understand how they work and how to configure them on the minion.

How to do it...

Uncomment and edit the `/etc/salt/minion` file to set the environment parameter:

```
environment: development
```

Setting grains in `/etc/salt/minion`

Uncomment and edit the `/etc/salt/minion` file to set the grains parameter:

```
grains:
    environment: development
    location: datacenter1
    server_type: webserver
```

Setting grains in `/etc/salt/grains`

Create the file `/etc/salt/grains` and populate it as follows:

```
environment: development
location: datacenter1
server_type: webserver
```
How it works...

The `environment` parameter in a Salt minion determines which environment the minion belongs to. This parameter helps Salt to determine which directory path it should look for on the master to fetch the correct configuration for the minion as per the environment specified.

As we specified in this recipe:

```yaml
environment: development
```

When the minion calls and requests the master for its configuration, it looks for the `development` environment configured on the master and, if it finds one, it then looks for its base path, which in our case is `/opt/salt-cookbook/development`. It then looks for the relevant files in this directory.

However, do note that this `environment` parameter is useful when the minion calls the master, that is, the pull mechanism takes place. When the master pushes the configurations to minions, we can specify the environment while doing so but it only tells the master which directory path it should serve the files from. However, it does not tell the master which minions to target. To target minions from the `development` environment when pushing configurations from the master, we have to use grains, which we will discuss next.

In Salt, grains are information about system properties that helps Salt to target minions and configure pillars and states based on these properties. Salt makes a lot of grains available to us by default, a few being `os`, `cpu`, `memory`, `hostname`, `domain name`, `IP addresses`, `MAC addresses`, and so on. However, this feature becomes more useful when we are able to configure custom grains to suit our needs such as `location`, `server type`, `application name`, `database name`, and so on.

Grains can be configured in a couple of ways. The first option is to configure grains in the minion configuration file, that is, `/etc/salt/minion`, and the second option is to do this in a different file called `/etc/salt/grains`. In both cases, grains are configured as YAML key-value pairs.

**Setting grains in `/etc/salt/minion`**

For the first option, that is, in the `/etc/salt/minion` file, we have to configure the grains as we have done in this recipe:

```yaml
grains:
    environment: development
    location: datacenter1
    server_type: webserver
```

Note the `grains` key in the first line. This is because the file `/etc/salt/minion` has a number of other parameters, and we have to explicitly specify that this section is for grains followed by the actual key-value pairs that determine our custom configured data.
Salt Architecture and Components

Setting grains in /etc/salt/grains
For the second option, that is, the file /etc/salt/grains, we populated it as follows:

   environment: development
   location: datacenter1
   server_type: webserver

Note the absence of the grains key here. The grains key is not needed in this case as this file is only for grain data. Next, we added the key-value pairs determining our custom data.

However, we have to keep in mind that grain data is static. If the grain data is modified or new grains are added, the minion has to be refreshed for the master to see the new data. This can be done with the help of execution modules, which we will learn later in the book. As of now, a restart of the salt-minion daemon should make the new data available.

See also
- The Installing and configuring the Salt minion recipe, to learn how to configure minions
- The Targeting minions recipe, in Chapter 2, Writing Advanced Salt Configurations, to learn about how to target minions
- The Applying Salt states to minions recipe, to learn how to synchronize minions with masters

Applying Salt states to minions
In this recipe, we will learn how to synchronize minions with the master and apply the states to minions that are configured on the master.

How to do it...
We will perform both the push and the pull mechanisms for applying states to minions.

1. Check if the minion has got automatically signed by the master:

   [root@salt-master ~]# salt-key -l accepted
   Accepted Keys:
   salt-minion

   If all the previous recipes were followed correctly, then the preceding output will be found and here salt-minion is the minion name whose key has been accepted.
2. The communication between the master and the minion can be checked with a couple of Salt commands:

```
[root@salt-master ~]# salt 'salt-minion' test.ping
salt-minion:
  True
```

```
[root@salt-master ~]# salt 'salt-minion' grains.items
salt-minion:
  biosreleasedate: 07/31/2013
  biosversion: 6.00
  .
  .
  zmqversion: 3.2.2
```

3. The user state that we configured can now be applied to the minion.

**Applying states using the push mechanism from master to minion**

States can be applied from master to minion as follows:

```
[root@salt-master ~]# salt 'salt-minion' state.sls user \ saltenv=development
salt-minion:
----------
  ID: generic_user
  Function: user.present
  Name: thomas
  Result: True
  Comment: New user thomas created
  Changes:
  ----------
    fullname:

    gid:
      1001
groups:
      - thomas
  home:
    /home/thomas
```
homephone:

name:
  thomas
passwd:
  x
password:
  $1$PG1inys0$kB2I83KzEVzVs9G7xLHjA1
roomnumber:

shell:
  /bin/bash
uid:
  1001
workphone:

Summary
--------
Succeeded: 1
Failed: 0
--------
Total: 1

There is a second option of applying the state from master to minion, as follows:

[root@salt-master ~]# salt 'salt-minion' state.highstate \
saltenv=development

If fewer lines of output are needed, then the following command can be used:

[root@salt-master]# salt 'salt-minion' state.highstate \ 
saltenv=development --state-output=terse
salt-minion:
  Name: thomas - Function: user.present - Result: Changed

Summary
--------
Succeeded: 1
Failed: 0
--------
Total: 1

**Applying states using the pull mechanism by minion from master**

States can be fetched by minion from master, as follows:

```
root@salt-minion:~# salt-call state.highstate
[INFO    ] Loading fresh modules for state activity
[INFO    ] Fetching file from saltenv 'development', ** skipped **
[INFO    ] latest already in cache 'salt://top.sls'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/modules'
[INFO    ] Syncing modules for environment 'development'
[INFO    ] Loading cache from salt://_modules, for development)
[INFO    ] Caching directory '_modules/' for environment 'development'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/states'
[INFO    ] Syncing states for environment 'development'
[INFO    ] Loading cache from salt://_states, for development)
[INFO    ] Caching directory '_states/' for environment 'development'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/grains'
[INFO    ] Syncing grains for environment 'development'
[INFO    ] Loading cache from salt://_grains, for development)
[INFO    ] Caching directory '_grains/' for environment 'development'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/renderers'
[INFO    ] Syncing renderers for environment 'development'
[INFO    ] Loading cache from salt://_renderers, for development)
[INFO    ] Caching directory '_renderers/' for environment 'development'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/returners'
[INFO    ] Syncing returners for environment 'development'
[INFO    ] Loading cache from salt://_returners, for development)
[INFO    ] Caching directory '_returners/' for environment 'development'
[INFO    ] Creating module dir '/var/cache/salt/minion/extmods/outputters'
[INFO    ] Syncing outputters for environment 'development'
[INFO    ] Loading cache from salt://_outputters, for development)
[INFO    ] Caching directory '_outputters/' for environment 'development'
[INFO    ] Loading fresh modules for state activity
```
Salt Architecture and Components

[INFO    ] Fetching file from saltenv 'development', ** skipped **
          latest already in cache 'salt://user/init.sls'
[INFO    ] Running state [thomas] at time 19:04:47.080222
[INFO    ] Executing state user.present for thomas
[INFO    ] Executing command 'useradd -s /bin/bash -u 1001 -m -d
          /home/thomas thomas' in directory '/root'
[INFO    ] {'shell': '/bin/bash', 'workphone': '', 'uid': 1001,
          'passwd': 'x', 'roomnumber': '', 'groups': ['thomas'], 'home':
          '/home/thomas', 'password': '$1$PG1inys0$kB2I83KzEVzVs9G7xLHjA1',
          'name': 'thomas', 'gid': 1001, 'fullname': '', 'homephone': ''}
[INFO    ] Completed state [thomas] at time 19:04:47.209616
local:
--------
    ID: generic_user
Function: user.present
     Name: thomas
  Result: True
Comment: New user thomas created
Changes:
--------
  fullname:

  gid:
    1001
groups:
   - thomas
home:
   /home/thomas
homephone:

  name:
    thomas
passwd:
  x
password:
    $1$PG1inys0$kB2I83KzEVzVs9G7xLHjA1
roomnumber:
Chapter 1

shell:
  /bin/bash
uid:
  1001
workphone:

Summary
--------
Succeeded: 1
Failed: 0
--------
Total: 1

The same task can also be done by using the following command:

root@salt-minion:~# salt-call state.sls user saltenv=development

How it works...

In this recipe, we have seen how to apply Salt states to minions. Let’s now look at the steps in details.

First, we tested if the certificate request for the minion had been accepted and signed by the master because we had configured it to accept all minions by defining the wildcard in the /etc/salt/autosign.conf file.

Next, we tested communication between the master and the minion with the salt binary and using what we call built-in execution modules in Salt. The test.ping module checks if the minion is available and is responding to the master and grains.items returns all the built-in and custom grains on the minion. Here, test.ping and grains.items are the execution modules and are just a couple of the numerous modules available in Salt.

Next, we look at how to actually apply the configured states to minions.

First, we see the push mechanism from the master to the minion:

[root@salt-master ~]# salt 'salt-minion' state.sls user \ saltenv=development
Salt Architecture and Components

Here, we see the salt binary being used along with the name (minion id) of the minion, followed by another execution module state.sls, and then the name of the state that we had configured. Finally, the name of the environment is specified without which Salt will look for the base environment and fail. Any configured state can be used in this format to apply to minions, and this method is independent of the top.sls file that we configured in the base directory of the development environment.

Next, we see an alternate method to apply states to minions:

[root@salt-master -]# salt 'salt-minion' state.highstate saltenv=development

Here we see the same command, without the state name and a new execution module called state.highstate. When state.highstate is mentioned, it applies all the states that have been configured for the minion in the top.sls file of that environment. Only the user state gets applied to the minion because that is the only state we have configured in top.sls so far.

The third command produces a slightly different output:

[root@salt-master]# salt 'salt-minion' state.highstate saltenv=development --state-output=terse

This command has a new option and its value is --state-output=terse. This option is specified if we do not want a detailed output as the first and second commands. However, it does the same task as the first and second commands.

Next, we look at the method of doing the same task by fetching the states on the minion from the master by using the pull mechanism:

root@salt-minion:~# salt-call state.highstate

Here, we used the salt-call binary, which is available only on Salt minions along with the state.highstate execution module. This command will automatically take the environment from the minion configuration file, that is, /etc/salt/minion and look for the minion states in the development environment and apply all the states configured for this minion in the top.sls file. Note the output, we see the minion caching all required data in the system from the master before applying the states. This caching process will take place each time the data in the master changes and the salt-call command is run.

The other method (not used very often) to apply specific states to the minion and from the minion is shown next. These states can also be part of highstate, if added to the top file:

root@salt-minion:~# salt-call state.sls user saltenv=development

Here, we see the minion using the salt-call binary, but mentioning the state.sls execution module to specify a particular state instead of all the states, that is, the user state. However, do note that, for this to be successful, the Salt environment must be mentioned as shown, without which it will look for the base environment and fail.
There's more...

Now let's see, how to apply Salt states on masterless minions.

**Applying Salt states to masterless minions**

In the *Installing and configuring the Salt minion* recipe, you learned how to configure a Salt minion to be its own master. Here, we will see how to apply states to a Salt minion that acts as its own master.

The following command applies the states of a minion by fetching the state configurations from itself:

```
root@salt-minion:~# salt-call --local state.highstate
```

Here, we see that the same `salt-call` command that was used in fetching the Salt states from a separate Salt master is used to do this task, but the `--local` option has been added to let Salt know that it needs to look for the state files in itself rather than in a separate Salt master.

**See also**

- The *Installing and configuring the Salt master* and *Installing and configuring Salt minion* recipes, to learn how to configure the master and minions
- The *Targeting minions* recipe in Chapter 2, *Writing Advanced Salt Configurations*, to learn about how to target minions from the master
Where to buy this book

You can buy Salt Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.