Essential configuration tasks for open source Puppet users

> Included in Puppet Enterprise 2017.2.

The following are common configuration tasks that you can manage with open source Puppet. These steps provide an excellent introduction to the capabilities of Puppet.

NTP quick start guide

NTP is one of the most crucial, yet easiest, services to configure and manage with Puppet. Follow this guide to properly get time synced across all your Puppet-managed nodes.

DNS quick start guide

This guide provides instructions for getting started managing a simple DNS nameserver file with Puppet. A nameserver ensures that the "human-readable" names you type in your browser (for example, `example.com`) resolve to IP addresses that computers can read.

Sudo users quick start guide

Managing sudo on your agents allows you to control which system users have access to elevated privileges. This guide provides instructions for getting started managing sudo privileges across your nodes, using a module from the Puppet Forge in conjunction with a simple module you will write.

Firewall quick start guide

Follow the steps in this guide to get started managing firewall rules with a simple module you'll write that defines those rules.

NTP quick start guide

> Included in Puppet Enterprise 2017.2.

Welcome to the Open Source Puppet NTP Quick Start Guide. This document provides instructions for getting started managing an NTP service using the Puppet NTP module.

The clocks on your servers are not inherently accurate. They need to synchronize with something to let them know what the right time is. NTP is a protocol designed to synchronize the clocks of computers over a network. NTP uses Coordinated Universal Time (UTC) to synchronize computer clock times to within a millisecond.

Your entire datacenter, from the network to the applications, depends on accurate time for many different things, such as security services, certificate validation, and file sharing across Puppet agents. If the time is wrong, your Puppet master might mistakenly issue agent certificates from the distant past or future, which other agents will treat as expired.

NTP is one of the most crucial, yet easiest, services to configure and manage with Puppet. Using the Puppet NTP module, you can do the following tasks:

- Ensure time is correctly synced across all the servers in your infrastructure.
- Ensure time is correctly synced across your configuration management tools.
- Roll out updates quickly if you need to change or specify your own internal NTP server pool.

This guide will step you through the following tasks:

- Install the `puppetlabs-ntp` module.
- Add classes to the `default` node in your main manifest.
- View the status of your NTP service.
- Use multiple nodes in the main manifest to configure NTP for different permissions.

> For this walk-through, log in as root or administrator on your nodes.

> **Prerequisites**: This guide assumes you've already installed Puppet, and have installed at least one *nix agent.

> **Note**: You can add the NTP service to as many agents as needed. For ease of explanation, we will describe only one.

Install the puppetlabs-ntp module

The puppetlabs-ntp module is part of the supported modules program; these modules are supported, tested, and maintained by Puppet. You can learn more about the puppetlabs-ntp module by visiting http://forge.puppetlabs.com/puppetlabs/ntp.

To install the puppetlabs-ntp module:

From the Puppet master, run `puppet module install puppetlabs-ntp`.

You should see output similar to the following:

```
Preparing to install into /etc/puppetlabs/puppet/modules ...
Notice: Downloading from http://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/puppet/environments/production/modules
└── puppetlabs-ntp (v3.1.2)
```

> That's it! You've just installed the puppetlabs-ntp module.

Add classes from the NTP module to the main manifest

The NTP module contains several **classes**. Classes are named chunks of Puppet code and are the primary means by which Puppet configures nodes. The NTP module contains the following classes:

- `ntp` : the main class; this class includes all other NTP classes (including the classes in this list).
- `ntp::install` : this class handles the installation packages.
- `ntp::config` : this class handles the configuration file.
- `ntp::service` : this class handles the service.

You're going to add the `ntp` class to the `default` node in your main manifest. Depending on your needs or infrastructure, you might have a different group that you'll assign NTP to, but you would take similar steps.

To create the NTP class:

1. From the command line on the Puppet master, navigate to the main manifest: `cd /etc/puppetlabs/code/environments/production/manifests`.
2. Use your text editor to open `site.pp`.
3. Add the following Puppet code to `site.pp`:

```
node default {
         class { 'ntp':
         servers => ['nist-time-server.eoni.com','nist1-lv.ust
         }
}
```

> **Note**: If you already have a default node, just add the `class` and `servers` lines to it. To see a list of other time servers, visit http://www.pool.ntp.org/.

4. From the command line on your Puppet agent, trigger a Puppet run with `puppet agent -t` .

> That's it! You've successfully configured Puppet to use NTP.

To check if the NTP service is running, run `puppet resource service ntpd` on your Puppet agent. The output should be:

```
service { 'ntpd':
              ensure => 'running',
              enable => 'true',
    }
```

Use multiple nodes to configure NTP for different permissions

Until now, you've been using the default node in this Quick Start Guide. If you want to configure the NTP service to run differently on different nodes, you can set up NTP differently in multiple nodes in the `site.pp` file.

In the example below, two ntp servers in the organization are allowed to talk to outside time servers ("kermit" and "grover"). Other ntp servers get their time data from these two servers. One of the primary ntp servers, "kermit", is very cautiously configured — it can't afford outages, so it's not allowed to automatically update its ntp server package without testing. The other servers are more permissively configured.

The other ntp servers ("snuffie," "bigbird," and "hooper") will use our two primary servers to sync their time.

The `site.pp` looks like this:

```
    node "kermit.example.com" {
      class { "ntp":
            servers    => [ '0.us.pool.ntp.org iburst','1.us.poo
```

```
                autoupdate => false,
                restrict   => [],
                enable     => true,
     }
    }

        node "grover.example.com" {
          class { "ntp":
                servers    => [ 'kermit.example.com','0.us.pool.ntp.
                autoupdate => true,
                restrict   => [],
                enable     => true,
     }
    }

        node "snuffie.example.com", "bigbird.example.com", "hooper.e
          class { "ntp":
                servers    => [ 'grover.example.com', 'kermit.exampl
                autoupdate => true,
                enable     => true,
     }
    }
```

In this fashion, it is possible to create multiple nodes to suit your needs.

Other resources

For more information about working with the puppetlabs-ntp module, check out our How to Manage NTP webinar.

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. We've noted a few below. Head over to the learning Puppet page to discover more.

- The Puppet workshop contains a series of self-paced, online lessons that cover a variety of topics on Puppet basics. You can sign up at the learning page.

DNS Quick Start Guide

> Included in Puppet Enterprise 2017.2.

Welcome to the Open Source Puppet DNS Quick Start Guide. This document provides instructions for getting started managing a simple DNS nameserver file with Puppet. A nameserver ensures that the "human-readable" names you type in your browser (for example, `example.com`) can be resolved to IP addresses that computers can read.

Sysadmins typically need to manage a nameserver file for internal resources that aren't published in public nameservers. For example, let's say you have several employee-maintained servers in your infrastructure, and the DNS network assigned to those servers use Google's public nameserver located at `8.8.8.8`. However, there are several resources behind your company's firewall that your employees need to access on a regular basis. In this case, you'd build a private nameserver (say at `10.16.22.10`), and then use Puppet to ensure all the servers in your infrastructure have access to it.

In this exercise, you will learn how to do the following steps:

- Write a simple module that contains a class called `resolver` to manage a nameserver file called `/etc/resolv.conf`.
- Enforce the desired state of that class from the command line of your puppet agent.

> Before starting this walk-through, complete the previous exercise in the essential configuration tasks, which is setting up NTP. Log in as root or administrator on your nodes.

> **Note**: You can add the DNS nameserver class to as many agents as needed. For ease of explanation, this guide will describe only one agent.

Write the `resolver` class

Some modules can be large, complex, and require a significant amount of trial and error, while others often work right out of the box. This module will be a very simple module to write, as it contains just one class and one template.

A quick note about modules

By default, Puppet keeps modules in an environment's `modulepath`, which for the production environment defaults to `/etc/puppetlabs/code/environments/production/modules`. This includes modules that Puppet installs, those that you download from the Forge, and those you write yourself.

Note: Puppet also creates another module directory: `/opt/puppetlabs/puppet/modules`. Don't modify or add anything in this directory, including modules of your own.

There are plenty of resources about modules and the creation of modules that you can reference. Check out Module Fundamentals, the Beginner's Guide to Modules, and the Puppet Forge.

Modules are directory trees. For this task, you'll create the following files:

- `resolver` (the module name)
  - `templates/`
    - `resolv.conf.erb` (contains template for `/etc/resolv.conf`, the contents of which will be populated after you add the class and run Puppet.)

To write the `resolver` class:

1. From the command line on the Puppet master, navigate to the modules directory: `cd /etc/puppetlabs/code/environments/production/modules`.
2. Run `mkdir -p resolver/templates` to create the new module directory and its templates directory.
3. Use your text editor to create the `resolver/templates/resolv.conf.erb` file.
4. Edit the `resolv.conf.erb` file to add the following Ruby code. This Ruby code is a template for populating `/etc/resolv.conf` correctly, no matter what changes are manually made to `/etc/resolv.conf`, as we will see in a later example.

```
# Resolv.conf generated by Puppet

<% [@nameservers].flatten.each do |ns| -%>
nameserver <%= ns %>
<% end -%>

# Other values can be added or hard-coded into the template as
```

5. Save and exit the file.

That's it! You've created a Ruby template to populate `/etc/resolv.conf`.

Add the resolv.conf file to your main manifest

1. On the Puppet master, open `/etc/resolv.conf` with your text editor, and copy the IP address of your master's nameserver (in this example, the nameserver is `10.0.2.3`).
2. On the Puppet master, navigate to the main manifest: `cd /etc/puppetlabs/code/environments/production/manifests`.
3. Use your text editor to open the `site.pp` file and add the following Puppet code to the `default` node, editing your nameserver value to match the one you found in `/etc/resolv.conf`:

```
$nameservers = ['10.0.2.3']

file { '/etc/resolv.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('resolver/resolv.conf.erb'),
}
```

4. From the command line on your Puppet agent, run `puppet agent -t`.
5. From the command line on your Puppet agent, run `cat /etc/resolv.conf`. The result should reflect the nameserver you added to your main manifest in step 3.

That's it! You've written a module that contains a class that will ensure your agents resolve to your internal nameserver.

Note the following about your new class:

- It ensures the creation of the file `/etc/resolv.conf`.
- The content of `/etc/resolv.conf` is modified and managed by the template, `resolv.conf.erb`.

Enforce the desired state of the `resolver` class

Finally, let's take a look at how Puppet will ensure the desired state of the `resolver` class on your agents. In the previous task, you set the nameserver IP address. Now imagine a scenario where a member of your team changes the contents of `/etc/resolv.conf` to use a different nameserver and can no longer access any internal resources.

1. On any agent to which you applied the `resolv.conf` class, edit `/etc/resolv.conf` to be any nameserver IP address other than the one you desire

to use.
2. Save and exit the file.
3. From the command line on your Puppet agent, run `puppet agent -t --onetime`.
4. From the command line on your Puppet agent, run `cat /etc/resolv.conf`, and notice that Puppet has enforced the desired state you specified on your Puppet master.

> That's it — Puppet has enforced the desired state of your agent!

Other resources

For more information about working with Puppet and DNS, check out our Dealing with Name Resolution Issues blog post.

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. We've noted a few below. Head over to the Learning Puppet page to discover more.

- The Puppet workshop contains a series of self-paced, online lessons that cover a variety of topics on Puppet basics. You can sign up at the learning page.
- Learn about Puppet DNS through this online training workshop.

Sudo users quick start guide

> Included in Puppet Enterprise 2017.2.

Welcome to the Open Source Puppet Sudo Users Quick Start Guide. This document provides instructions for getting started managing sudo privileges across your Puppet deployment, using a module from the Puppet Forge in conjunction with a simple module you will write.

In most cases, managing sudo on your agents involves controlling which users have access to elevated privileges. Using this guide, you will learn how to do the following tasks:

- Install the `saz-sudo` module as the foundation for managing sudo privileges.
- Write a simple module that contains a class called `privileges` to manage a resource that sets privileges for certain users, which will be managed by the `saz-sudo` module.
- Add classes from the privileges and sudo modules to your agents.

> Before starting this walk-through, complete the previous exercises in the essential configuration tasks. Log in as root or administrator on your nodes.

> **Prerequisites**: This guide assumes you've already installed Puppet, and have installed at least one *nix agent.

> **Note**: You can add the sudo and privileges classes to as many agents as needed, although we describe only one for ease of explanation.

Install the `saz-sudo` module

The `saz-sudo` module, available on the Puppet Forge, is one of many modules written by a member of the Puppet user community. You can learn more about the module by visiting http://forge.puppetlabs.com/saz/sudo.

To install the `saz-sudo` module:

As the root user on the Puppet master, run `puppet module install saz-sudo`.

You should see output similar to the following:

Preparing to install into /etc/puppetlabs/code/environments/production/modules …

```
Notice: Downloading from http://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/puppet/modules
└── saz-sudo (v2.3.6)
      └── puppetlabs-stdlib (3.2.2) [/opt/puppet/share/puppet/mo
```

That's it! You've just installed the `saz-sudo` module.

Write the `privileges` class

Some modules can be large, complex, and require a significant amount of trial and error as you create them, while others often work right out of the box. This module will be a very simple module to write. It contains just one class.

A quick note about modules directories

By default, Puppet keeps modules in an environment's `modulepath`, which for the production environment defaults to `/etc/puppetlabs/code/environments/production/modules`. This includes modules that Puppet installs, those that you download from the Forge, and those you write yourself.

Note: Puppet also creates another module directory: `/opt/puppetlabs/puppet/modules`. Don't modify or add anything in this directory, including modules of your own.

There are plenty of resources about modules and the creation of modules that you can reference. Check out Module Fundamentals, the Beginner's Guide to Modules, and the Puppet Forge.

Modules are directory trees. For this task, you'll create the following files:

- `privileges/` (the module name)
  - `manifests/`
    - `init.pp` (contains the `privileges` class)

To write the `privileges` class:

1. From the command line on the Puppet master, navigate to the modules directory: `cd /etc/puppetlabs/code/environments/production/modules`.

2. Run `mkdir -p privileges/manifests` to create the new module directory and its manifests directory.
3. From the `manifests` directory, use your text editor to create the `init.pp` file, and edit it so it contains the following Puppet code:

```
class privileges {

  sudo::conf { 'admins':
  ensure  => present,
  content => '%admin ALL=(ALL) ALL',
  }

}
```

4. Save and exit the file.

That's it! You've written a module that contains a class that, once applied, ensures that your agents have the correct sudo privileges set for the root user and the "admins" and "wheel" groups.

Note the following about the resource in the `privileges` class:

- The `sudo::conf 'admins'` line creates a sudoers rule to ensure that members of the `admins` group have the ability to run any command using sudo. This resource creates configuration fragment file to define this rule in `/etc/sudoers.d/`. It will be called something like `10_admins`.

Add the privileges and sudo classes

1. From the command line on the Puppet master, navigate to the main manifest: `cd /etc/puppetlabs/code/environments/production/manifests`.
2. Open `site.pp` with your text editor and add the following Puppet code to the `default` node:

```
class { 'sudo': }
sudo::conf { 'web':
  content  => "web ALL=(ALL) NOPASSWD: ALL",
}
class { 'privileges': }
sudo::conf { 'jargyle':
  priority => 60,
  content  => "jargyle ALL=(ALL) NOPASSWD: ALL",
}
```

1. Save and exit the file.

2. From the command line on your Puppet master, run `puppet parser validate site.pp` to ensure that there are no errors. The parser will return nothing if there are no errors.
3. From the command line on your Puppet agent, run `puppet agent -t` to trigger a Puppet run.

> That's it! You have successfully installed the Sudo module and applied privileges and classes to it.
>
> Note the following about your new resources in the `site.pp` file:
>
> - `sudo::conf 'web'` : Creates a sudoers rule to ensure that members of the web group have the ability to run any command using sudo. This resource creates a configuration fragment file to define this rule in `/etc/sudoers.d/` .
> - `sudo::conf 'admins'` : Creates a sudoers rule to ensure that members of the admins group have the ability to run any command using sudo. This resource creates a configuration fragment file to define this rule in `/etc/sudoers.d/` . It will be called something like `10_admins` .
> - `sudo::conf 'jargyle'` : Creates a sudoers rule to ensure that the user `jargyle` has the ability to run any command using sudo. This resource creates a configuration fragment to define this rule in `/etc/sudoers.d/` . It will be called something like `60_jargyle` .

From the command line on the Puppet agent, run `sudo -l -U jargyle` to confirm it worked. The results should resemble the following:

```
 Matching Defaults entries for jargyle on this host:
!visiblepw, always_set_home, env_reset, env_keep="COLORS DISPLAY HOS
INPUTRC KDEDIR LS_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME LA
LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT LC
env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE", en
LC_ALL LANGUAGE LINGUAS _XKB_CHARSET XAUTHORITY",
secure_path=/usr/local/bin\:/sbin\:/bin\:/usr/sbin\:/usr/bin

 User jargyle may run the following commands on this host:
(ALL) NOPASSWD: ALL
```

Other resources

For more information about working with Puppet and Sudo Users, check out our Module of The Week: saz/sudo - Manage sudo configuration blog post.

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. We've noted one below; head over to the learning Puppet page to discover more.

- The Puppet workshop contains a series of self-paced, online lessons that cover a variety of topics on Puppet basics. You can sign up at the learning page.
- Learn about Managing sudo Privileges through this online training workshop.

Firewall quick start guide

> Included in Puppet Enterprise 2017.2.

Welcome to the Open Source Puppet Firewall Quick Start Guide. This document provides instructions for getting started managing firewall rules with Puppet.

With a firewall, admins define a set of policies (also known as firewall rules) that consist of things like application ports (TCP/UDP), network ports, IP addresses, and an accept/deny statement. These rules are applied in a "top-to-bottom" approach. For example, when a service, say SSH, attempts to access resources on the other side of a firewall, the firewall applies a list of rules to determine if or how SSH communications are handled. If a rule allowing SSH access can't be found, the firewall will deny access to that SSH attempt.

To best manage such rules with Puppet, you want to divide these rules into `pre` and `post` groups to ensure Puppet checks firewall rules in the correct order.

Using this guide, you will learn how to do the following tasks:

- Install the puppetlabs-firewall module.
- Write a simple module to define the firewall rules for your Puppet-managed infrastructure.
- Add the firewall module to the main manifest.
- Enforce the desired state of the `my_firewall` class.

> Before starting this walk-through, complete the previous exercises in the essential configuration tasks. Log in as root or administrator on your nodes.

> **Prerequisites**: This guide assumes you've already installed Puppet, and have installed at least one *nix agent.

> You should still be logged in as root or administrator on your nodes.

Install the `puppetlabs-firewall` module

The firewall module, available on the Puppet Forge, introduces the firewall resource, which is used to manage and configure firewall rules from with Puppet. Learn more about the module by visiting http://forge.puppetlabs.com/puppetlabs/firewall.

To install the firewall module:

From the Puppet master, run `puppet module install puppetlabs-firewall`.

You should see output similar to the following:

```
Preparing to install into /etc/puppetlabs/puppet/environments/pr
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/puppet/environments/production/modules
└── puppetlabs-firewall (v1.6.0)
```

That's it! You've just installed the firewall module.

Write the `my_firewall` module

Some modules can be large, complex, and require a significant amount of trial and error. This module, however, will be a very simple module to write. It contains just three classes.

A quick note about module directories

By default, Puppet keeps modules in an environment's `modulepath`, which for the production environment defaults to `/etc/puppetlabs/code/environments/production/modules`. This includes modules that Puppet installs, those that you download from the Forge, and those you write yourself.

Note: Puppet also creates another module directory: `/opt/puppetlabs/puppet/modules`. Don't modify or add anything in this directory, including modules of your own.

There are plenty of resources about modules and the creation of modules that you can reference. Check out Module Fundamentals, the Beginner's Guide to Modules, and the Puppet Forge.

Modules are directory trees. For this task, you'll create the following files:

- `my_firewall/` (the module name)

- manifests/
  - pre.pp
  - post.pp

To write the `my_firewall` module:

1. From the command line on the Puppet master, navigate to the modules directory: `cd /etc/puppetlabs/code/environments/production/modules`.
2. Run `mkdir -p my_fw/manifests` to create the new module directory and its manifests directory.
3. From the `manifests` directory, use your text editor to create `pre.pp`.
4. Edit `pre.pp` so it contains the following Puppet code. These rules allow basic networking to ensure that existing connections are not closed.

```
class my_fw::pre {
  Firewall {
    require => undef,
  }
```

Default firewall rules

```
  firewall { '000 accept all icmp':
    proto  => 'icmp',
    action => 'accept',
  }
  firewall { '001 accept all to lo interface':
    proto   => 'all',
    iniface => 'lo',
    action  => 'accept',
  }
  firewall { '002 reject local traffic not on loopback inter
    iniface     => '! lo',
    proto       => 'all',
    destination => '127.0.0.1/8',
    action      => 'reject',
  }
  firewall { '003 accept related established rules':
    proto  => 'all',
    state  => ['RELATED', 'ESTABLISHED'],
    action => 'accept',
  }
}
```

5. Save and exit the file.
6. From the `manifests` directory, use your text editor to create `post.pp`.

7. Edit `post.pp` so it contains the following Puppet code. This drops any requests that don't meet the rules defined in `pre.pp` or your rules defined in `site.pp` (see next section).

```
class my_fw::post {
    firewall { '999 drop all':
      proto  => 'all',
      action => 'drop',
      before => undef,
    }
  }
```

8. Save and exit the file.

> That's it! You've written a module that contains a class that, once applied, ensures your firewall has rules in it that will be managed by Puppet. Note the following about your new class:
>
> - `pre.pp` defines the "pre" group rules the firewall applies when a service requests access. It is run before any other rules.
> - `post.pp` defines the rule for the firewall to drop any requests that haven't met the rules defined by `pre.pp` or in `site.pp` (see next section).

Add the firewall module to the main manifest

1. On your Puppet master, navigate to the main manifest: `cd /etc/puppetlabs/code/environments/production/manifests`.
2. Use your text editor to open `site.pp`.
3. Add the following Puppet code to your `site.pp` file. This will clear any existing rules and make sure that only rules defined in Puppet exist on the machine.

```
resources { 'firewall':
  purge => true,
}
```

4. Add the following Puppet code to your `site.pp` file. These defaults will ensure that the `pre` and `post` classes are run in the correct order to avoid locking you out of your box during the first Puppet run, and declaring `my_fw::pre` and `my_fw::post` satisfies the specified dependencies.

```
Firewall {
  before  => Class['my_fw::post'],
  require => Class['my_fw::pre'],
}
```

```
class { ['my_fw::pre', 'my_fw::post']: }
```

5. Add the `firewall` class to your `site.pp` to ensure the correct packages are installed:

```
class { 'firewall': }
```

That's it! To check your firewall configuration, run `iptables --list` from the command line of your Puppet agent. The result should look similar to this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     icmp --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
REJECT     all  --  anywhere             loopback/8
ACCEPT     all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Enforce the desired state of the `my_firewall` class

Lastly, let's take a look at how Puppet ensures the desired state of the `my_firewall` class on your agents. In the previous task, you applied your firewall class. Now imagine a scenario where a member of your team changes the contents of the `iptables` to allow connections on a random port that was not specified in `my_firewall`.

1. Select an agent on which you applied the `my_firewall` class, and run `iptables --list`.
2. Note that the rules from the `my_firewall` class have been applied.
3. From the command line, insert a new rule to allow connections to port **8449** by running `iptables -I INPUT -m state --state NEW -m tcp -p tcp --dport 8449 -j ACCEPT`.
4. Run `iptables --list` again and note that this new rule is now listed.
5. Run `puppet agent -t --onetime` to trigger a Puppet run on that agent.

6. Run `iptables --list` on that node once more, and notice that Puppet has enforced the desired state you specified for the firewall rules.

> That's it–Puppet has enforced the desired state of your agent!

Other resources

You can learn more about the Puppet Firewall module by visiting the Puppet Forge.

Check out the other quick start guides in our Puppet QSG series:

- NTP quick start guide
- DNS quick start guide
- Sudo users quick start guide

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. We've noted one below; head over to the learning Puppet page to discover more.

- The Puppet workshop contains a series of self-paced, online lessons that cover a variety of topics on Puppet basics. You can sign up at the learning page.