THE EXPERT'S VOICE® IN OPEN SOURCE

SECOND EDITION

Pro Puppet

Spencer Krum, William Van Hevelingen, Ben Kero
James Turnbull, and Jeffrey McCune

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Getting Started with Puppet

Puppet is an open source framework and toolset for managing the configuration of computer systems. This book looks at how you can use Puppet to manage your configuration. As the book progresses, we'll introduce Puppet's features and show you how to integrate Puppet into your provisioning and management lifecycle. To do this, we'll take you through configuring a real-world scenario that we'll introduce in Chapter 2. In Chapter 3, we'll show you how to implement a successful Puppet workflow using version control and Puppet environments. In Chapter 4, we'll show you how to build high availability and horizontal scalability into your Puppet infrastructure. The rest of the book will focus on extending what you can do with Puppet and its ecosystem of tools, and on gaining unprecedented visibility into your infrastructure.

In this chapter, you'll find the following:

- A quick overview of Puppet, what it is, how it works, and which release to use.
- How to install Puppet and its inventory tool, Facter, on RedHat, Debian, Ubuntu, Solaris, Microsoft Windows, Mac OS X, and via RubyGems.
- How to configure Puppet and create your first configuration items.
- The Puppet domain-specific language that you use to create Puppet configuration.
- The concept of "modules," Puppet's way of collecting and managing bundles of configuration data.
- How to apply one of these modules to a host using the Puppet agent.

What Is Puppet?

Puppet is Ruby-based configuration management software, licensed as Apache 2.0, and it can run in either client-server or stand-alone mode. Puppet was principally developed by Luke Kanies and is now developed by his company, Puppet Labs. Kanies has been involved with Unix and systems administration since 1997 and developed Puppet from that experience. Unsatisfied with existing configuration management tools, Kanies began working with tool development in 2001, and in 2005 he founded Puppet Labs, an open source development house focused on automation tools. Shortly after this, Puppet Labs released its flagship product, Puppet Enterprise. Puppet has two versions available: the open source version and the Enterprise version. The Enterprise version comes with an automated installer, a web management interface, and support contract. This book will focus on the open source version of Puppet, but since the software at the core of both tools is the same, the information will be valuable to consumers of either product.

Puppet can be used to manage configuration on Unix (including OS X), Linux, and Microsoft Windows platforms. Puppet can manage a host throughout its life cycle: from initial build and installation, to upgrades, maintenance, and finally to end-of-life, when you move services elsewhere. Puppet is designed to interact continuously with your hosts, unlike provisioning tools that build your hosts and leave them unmanaged.

Puppet has a simple operating model that is easy to understand and implement (Figure 1-1). The model is made up of three components:

- Deployment Layer
- Configuration Language and Resource Abstraction Layer
- Transactional Layer

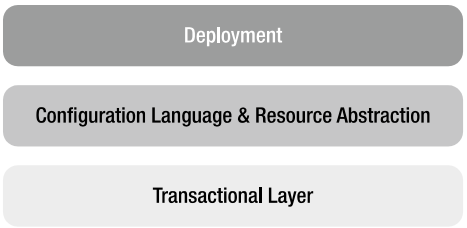

Figure 1-1. *The Puppet model*

Deployment

Puppet is usually deployed in a simple client-server model (Figure 1-2). The server is called a *Puppet master*, the Puppet client software is called an *agent,* and the host itself is defined as a *node*.

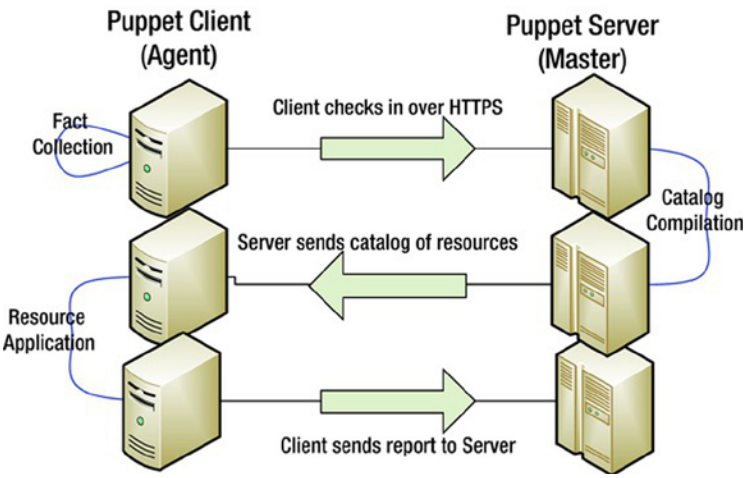

Figure 1-2. *High-level overview of a Puppet configuration run*

The Puppet master runs as a daemon on a host and contains the configuration required for the specific environment. The Puppet agents connect to the Puppet master through an encrypted and authenticated connection using standard SSL, and retrieve or "pull" any configuration to be applied.

Importantly, if the Puppet agent has no configuration available or already has the required configuration, Puppet will do nothing. Puppet will only make changes to your environment if they are required. This property is called *idempotency* and is a key feature of Puppet. The whole process is called a *configuration run*.

Each agent can run Puppet as a daemon, via a mechanism such as `cron`, or the connection can be manually triggered. The usual practice is to run Puppet as a daemon and have it periodically check with the master to confirm that its configuration is up-to-date or to retrieve any new configuration (Figure 1-3). However, many people find that being able to trigger Puppet via a mechanism such as `cron`, or manually, better suits their needs. By default, the Puppet agent will check the master for new or changed configuration once every 30 minutes. You can configure this period to suit your needs.

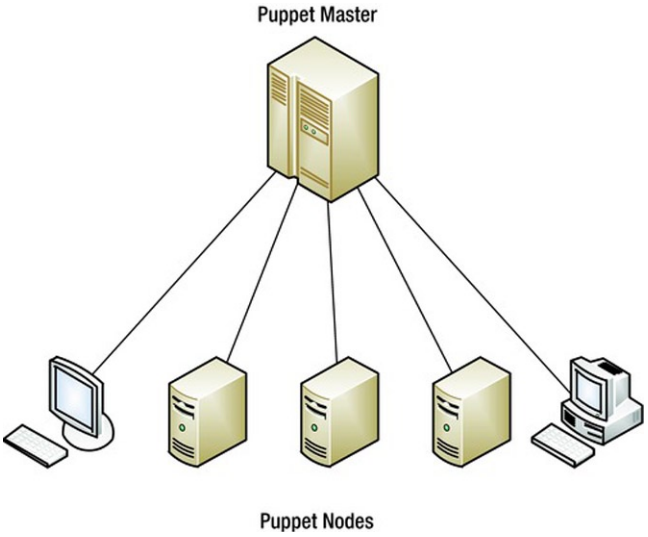

Figure 1-3. *The Puppet client-server model*

Other deployment models also exist. For example, Puppet can run in a stand-alone mode, where no Puppet master is required. Configuration is installed locally on the host and the puppet binary is run to execute and apply that configuration. We discuss this method in Chapter 4.

The Configuration Language and Resource Abstraction Layer

Puppet uses a declarative language, the Puppet language, to define your configuration items, which Puppet calls *resources*. Being declarative creates an important distinction between Puppet and many other configuration tools. A declarative language makes statements about the state of your configuration—for example, it declares that a package should be installed or a service should be started.

Most configuration tools, such as a shell or Perl script, are imperative or procedural. They describe *how* things should be done rather than the desired end state—for example, most custom scripts used to manage configuration would be considered imperative.

Puppet users just declare what the state of their hosts should be: what packages should be installed, what services should be running, and so on. With Puppet, the system administrator doesn't care *how* this state is achieved—that's Puppet's problem. Instead, we abstract our host's configuration into resources.

The Configuration Language

What does this declarative language mean in real terms? Let's look at a simple example. Suppose we have an environment with Red Hat Enterprise Linux, Ubuntu, and Solaris hosts and we want to install the `vim` application on all our hosts. To do this manually, we'd need to write a script that does the following:

- Connects to the required hosts (including handling passwords or keys).
- Checks to see if `vim` is installed.
- If not, uses the appropriate command for each platform to install `vim`, for example on RedHat the `yum` command and on Ubuntu the `apt-get` command.
- Potentially reports the results of this action to ensure completion and success.

■ **Note** This would become even more complicated if you wanted to upgrade `vim` (if it was already installed) or apply a particular version of `vim`.

Puppet approaches this process quite differently. In Puppet, you define a configuration resource for the `vim` package. Each resource is made up of a *type* (what sort of resource is being managed: packages, services, or cron jobs), a *title* (the name of the resource), and a series of *attributes* (values that specify the state of the resource—for example, whether a service is started or stopped).

You can see an example of a resource in Listing 1-1.

Listing 1-1. A Puppet resource

```
package { 'vim':
  ensure => present,
}
```

The resource in Listing 1-1 specifies that a package called `vim` should be installed. It is constructed as follows:

```
type { title:
  attribute => value,
}
```

In Listing 1-1, the resource type is the `package` type. Puppet comes with a number of resource types by default, including types to manage files, services, packages, and cron jobs, among others.

■ **Note** You can see a full list of the types Puppet can currently manage (and their attributes) at `http://docs.puppetlabs.com/references/stable/type.html`. You can also extend Puppet to support additional resource types, as we'll discuss in Chapter 10.

Next is the *title* of the resource, here the name of the package we want to install, `vim`. This corresponds exactly to the argument to the package manager; for example, `.apt-get install vim`.

Last, we've specified a single attribute, `ensure`, with a value of `present`. Attributes tell Puppet about the required state of our configuration resource. Each type has a series of attributes available to configure it. Here the `ensure` attribute specifies the state of the package: installed, uninstalled, and so on. The `present` value tells Puppet we want to install the package. To uninstall the package, we would change the value of this attribute to `absent`.

The Resource Abstraction Layer

With our resource created, Puppet takes care of the details of managing that resource when our agents connect. Puppet handles the "how" by knowing how different platforms and operating systems manage certain types of resources. Each type has a number of *providers*. A provider contains the "how" of managing packages using a particular package management tool.

The `package` type, for example, has more than 20 providers covering a variety of tools, including `yum`, `aptitude`, `pkgadd`, `ports`, and `emerge`.

When an agent connects, Puppet uses a tool called *Facter* (see following sidebar) to return information about that agent, including what operating system it is running. Puppet then chooses the appropriate package provider for that operating system and uses that provider to check if the `vim` package is installed. For example, on Red Hat it would `execute yum`, on Ubuntu it would execute `aptitude`, and on Solaris it would use the `pkg` command. If the package is not installed, Puppet will install it. If the package is already installed, Puppet does nothing. Again, this important feature is called *idempotency*.

Puppet will then report its success or failure in applying the resource back to the Puppet master.

INTRODUCING FACTER AND FACTS

Facter is a system inventory tool, also developed principally by Puppet Labs, that we use throughout the book. It is also open source under the Apache 2.0 license. It returns "facts" about each node, such as its hostname, IP address, operating system and version, and other configuration items. These facts are gathered when the agent runs. The facts are then sent to the Puppet master, and automatically created as variables available to Puppet at *top scope*. You'll learn more about variable scoping in Chapter 2.

You can see the facts available on your clients by running the `facter` binary from the command line. Each fact is returned as a key => value pair:

```
$ facter
operatingsystem => Ubuntu
ipaddress => 10.0.0.10
```

You can then use these values to configure each host individually. For example, knowing the IP address of a host allows you to configure networking on that host.

These facts are made available as variables that can be used in your Puppet configuration. When combined with the configuration you define in Puppet, they allow you to customize that configuration for each host. For example, they allow you to write generic resources, like your network settings, and customize them with data from your agents.

Facter also helps Puppet understand how to manage particular resources on an agent. For example, if Facter tells Puppet that a host runs Ubuntu, then Puppet knows to use `aptitude` to install packages on that agent. Facter can also be extended to add custom facts for specific information about your hosts. We'll be installing Facter shortly after we install Puppet, and we'll discuss it in more detail in later chapters.

The Transactional Layer

Puppet's transactional layer is its engine. A Puppet transaction encompasses the process of configuring each host, including these steps:

- Interpret and compile your configuration.
- Communicate the compiled configuration to the agent.
- Apply the configuration on the agent.
- Report the results of that application to the master.

The first step Puppet takes is to analyze your configuration and calculate how to apply it to your agent. To do this, Puppet creates a graph showing all resources, with their relationships to each other and to each agent. This allows Puppet to work out the order, based on relationships you create, in which to apply each resource to your host. This model is one of Puppet's most powerful features.

Puppet then takes the resources and compiles them into a *catalog* for each agent. The catalog is sent to the host and applied by the Puppet agent. The results of this application are then sent back to the master in the form of a report.

The transaction layer allows configurations to be created and applied repeatedly on the host. Again, Puppet calls this capability idempotency, meaning that multiple applications of the same operation will yield the same results. Puppet configuration can be safely run multiple times with the same outcome on your host, ensuring that your configuration stays consistent.

Puppet is not fully transactional, though; your transactions aren't logged (other than informative logging), and so you can't roll back transactions as you can with some databases. You can, however, model transactions in a "noop," or no-operation mode, that allows you to test the execution of your changes without applying them.

Selecting the Right Version of Puppet

The best version of Puppet to use is usually the latest release, which at the time of writing is the 3.2.x branch of releases; newer ones are currently in development. The biggest advantage of the 3.2.x branch of releases is improved performance and built-in Hiera integration. Hiera is Puppet's external datastore and will be extensively covered in later chapters.

The 3.1.x releases are stable, perform well, have numerous bug fixes not available in previous versions, and contain a wide of variety of new features and functions unavailable in earlier releases.

■ **Note** This book assumes you are using either a 3.1.x or later release. Some of the material will work on 2.7.x versions of Puppet, but not all of it has been tested. Specifically, information about Hiera (see Chapter 12) and functions is unlikely to be backward-compatible to version 2.7.x.

There are a variety of releases, some older than others, packaged for operating systems. The 2.7.x releases are broadly packaged. The 3.1.x releases are packaged and distributed in newer versions of operating systems and platforms. If you can't find later Puppet releases packaged for your distribution, you have the option of rolling your own packages, backporting, or installing from source (though we don't recommend the latter—see the following). Puppetlabs provides the latest rpms, deb packages, msis, and dmg files on their website and repositories.

MIXING RELEASES OF PUPPET

The most common deployment model for Puppet is client-server. Many people ask if you can have different releases of Puppet on the master and as agents. The answer is yes, with some caveats. The first caveat is that the master needs to be a later release than the agents. For example, you can have a version 2.7.20 agent connected to a version 3.1.1 master, but not a version 3.1.1 agent connected to a 2.7.20 master.

The second caveat is that the older the agent release, the less likely it will function correctly with a newer release of the master. Later versions of masters may not be so forgiving of earlier agents, and some functions and features may not behave correctly.

Finally, mixing 3.1.x and later release masters with 2.7.x and earlier agents will mean you won't get the full performance enhancements available in 3.1.x.

Installing Puppet

Puppet can be installed and used on a variety of different platforms, including the following:

- Red Hat Enterprise Linux, CentOS, Fedora, and Oracle Enterprise Linux
- Debian and Ubuntu
- OpenIndiana
- Solaris
- From source
- Microsoft Windows (clients only)
- MacOS X and MacOS X Server
- Other (that is, BSD, Mandrake, and Mandriva)

Most of these are discussed in sections that follow. On these platforms, Puppet manages a variety of configuration items, including but not limited to these:

- Files
- Services
- Packages
- Users
- Groups
- Cron jobs
- SSH keys
- Nagios configuration

For Puppet, the agent and master server installations are very similar, although most operating systems and distribution packaging systems divide the master and agent functions into separate packages. On some operating systems and distributions, you'll also need to install Ruby and its libraries and potentially some additional packages. Most good packaging systems will have most of the required packages, like Ruby, as prerequisites of the Puppet and Facter packages. For other features (including some types of reporting that we'll demonstrate later in this book), you may also need to install additional packages.

We'll also demonstrate how to install Puppet from source, but we don't recommend this approach. It is usually simpler to use your operating system's package management system, especially if you are installing Puppet on a large number of hosts.

Installing on Red Hat Enterprise Linux and Fedora

Add the Extra Packages for Enterprise Linux (EPEL) or Puppet Labs repositories to your host and then install packages, as described in the following sections. Note that at the time of writing you must use the Puppet Labs repository for Puppet 3 packages.

Installing EPEL Repositories

The EPEL repository is a volunteer-based community effort from the Fedora project to create a repository of high-quality add-on packages for Red Hat Enterprise Linux (RHEL) and its compatible spinoffs such as CentOS, Oracle Enterprise Linux, and Scientific Linux.

You can find more details on EPEL, including how to add it to your host, at `http://fedoraproject.org/wiki/EPEL` and `http://fedoraproject.org/wiki/EPEL/FAQ#howtouse`.

You can add the EPEL repository by adding the epel-release RPM(`.rpm` package manager)as follows:

- Enterprise Linux 5:

```
# rpm -Uvh http://dl.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm
```

- Enterprise Linux 6:

```
# rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
```

Installing Puppet Labs Repositories

You can install the Puppet Labs repository on Linux 5 and 6 in a similar fashion:

- Enterprise Linux 5:

```
# rpm -ivh http://yum.puppetlabs.com/el/5/products/i386/puppetlabs-release-5-7.noarch.rpm
```

- Enterprise Linux 6:

```
# rpm -ivh http://yum.puppetlabs.com/el/6/products/i386/puppetlabs-release-6-7.noarch.rpm
```

Installing the EPEL and Puppet Lab Packages

On the master, you need to install the `puppet`, `puppet-server`, and `facter` packages from the EPEL or Puppet Labs repositories:

```
# yum install puppet puppet-server facter
```

The `puppet` package contains the agent, the `puppet-server` package contains the master, and the `facter` package contains the system inventory tool Facter. As mentioned earlier, Facter gathers information, or *facts*, about your hosts that are used to help customize your Puppet configuration.

On the agent, you only need to install the prerequisites and the `puppet` and `facter` packages:

```
# yum install puppet facter
```

Installing Via RubyGems

Like most Ruby-based applications, Puppet and Facter can also be installed via RubyGems. To do this, you'll need to install Ruby and the appropriate RubyGems package for your operating system. On Red Hat, CentOS, Fedora, SUSE/SLES, Debian and Ubuntu, this package is called `rubygems`. Once this package is installed, the `gem` command should be available to use. You can then use `gem` to install Puppet and Facter, as shown here:

```
# gem install puppet facter
```

Installing on Debian and Ubuntu

For Debian and Ubuntu, the `puppet` package contains the Puppet agent, and the `puppetmaster` package contains the master. On the master, you need to install this:

```
# apt-get install puppet puppetmaster
```

On the agent, you only need to install the `puppet` package:

```
# apt-get install puppet
```

■ **Note** Installing the `puppet`, `puppetmaster`, and `facter` packages will also install some prerequisite packages, such as Ruby itself, if they are not already installed.

For the latest version of Puppet you can use the following Puppetlabs repositories:

- Debian Wheezy:

```
# wget http://apt.puppetlabs.com/puppetlabs-release-wheezy.deb
# dpkg -i puppetlabs-release-wheezy.deb
# apt-get update
```

- Ubuntu Precise:

```
# wget http://apt.puppetlabs.com/puppetlabs-release-precise.deb
# dpkg -i puppetlabs-release-precise.deb
# apt-get update
```

Replace "precise" with other code names for different versions of Debian and Ubuntu.

Installing on OpenIndiana

Installing Puppet on OpenIndiana requires installing Ruby first. Then install Puppet and Facter via a RubyGem. Start by using the `pkg` command to install Ruby:

```
# pkg install ruby-18
```

RubyGems is installed by default when the `ruby-18` package is installed. You can use the `gem` command to install Puppet.

```
# gem install puppet facter
```

The `puppet` and `facter` binaries are now installed in this folder:

```
/var/ruby/1.8/gem_home/bin/
```

Installing on Solaris 10 and 11

On Solaris there are no native packages for Puppet, so you will need to install from OpenCSW packages, RubyGems or Source. OpenCSW (http://www.opencsw.org/about) is a community-led software packaging project for Solaris. They have prebuilt Puppet packages for both Solaris 10 and Solaris 11. At the time of writing there is both a `puppet3` package and a `puppet` package. We will use the `puppet3` package to get the stable 3.x version.

1. To begin we will install OpenCSW.

```
# pkgadd -d http://get.opencsw.org/now
```

2. Next install Puppet and dependencies:

```
# pkgutil --install puppet3
```

After installation Puppet will be available in `/opt/csw/bin`.

Installing from Source

You can also install Puppet and Facter from source tarballs. We don't recommend this approach, because it makes upgrading, uninstalling, and generally managing Puppet across many hosts difficult. To do this you'll need to ensure that some prerequisites are installed, for example Ruby and its libraries, using the appropriate packages for your host or via source again.

1. First, download the Facter tarball from the Puppet Labs site:

```
$ cd /tmp
$ wget http://downloads.puppetlabs.com/facter/facter-1.6.18.tar.gz
```

2. Unpack the tarball and run the `install.rb` script to install Facter:

```
$ tar -zxf facter-1.6.18.tar.gz
$ cd facter-1.6.18
# ./install.rb
```

This will install Facter into the default path for Ruby libraries on your host, for example `/usr/lib/ruby/` on many Linux distributions.

3. Next, download and install Puppet using the same process:

```
$ cd /tmp
$ wget http://downloads.puppetlabs.com/puppet/puppet-3.1.1.tar.gz
$ tar -zxf puppet-3.1.1.tar.gz
$ cd puppet-3.1.1
# ./install.rb
```

Like the Facter steps, this will install Puppet into the default path for Ruby libraries on your host.

■ **Note** You can find the latest Puppet and Facter releases at http://puppetlabs.com/misc/download-options/.

Installing on Microsoft Windows

Puppet does not currently support running a Puppet master on Microsoft Windows. You will need a Unix/Linux Puppet master for client-server, or you can run Puppet in masterless mode.

Installing on Microsoft Windows Graphically

To install on Microsoft Windows graphically, follow these steps:

1. Download the latest open source MSI from http://downloads.puppetlabs.com/windows/ (The MSI file bundles all of Puppet's dependencies, including Ruby and Facter).
2. Run the MSI as an administrator and follow the installation wizard (Figure 1-4).

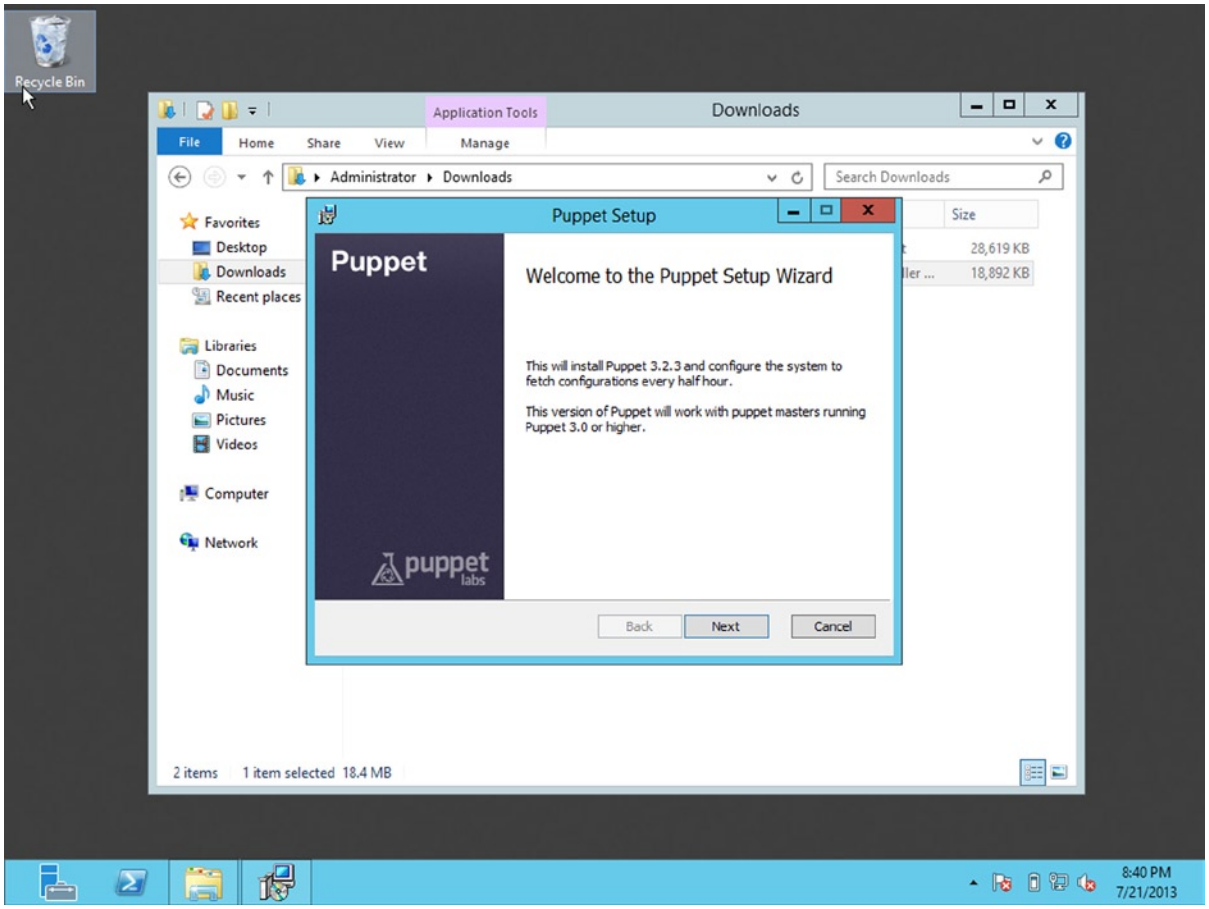

Figure 1-4. *Beginning Puppet installation*

3. You will need to supply the name of your Puppet master to the installer (Figure 1-5). After that, Puppet will begin running as a Windows service. When the installation is complete you will see the screen in Figure 1-6.

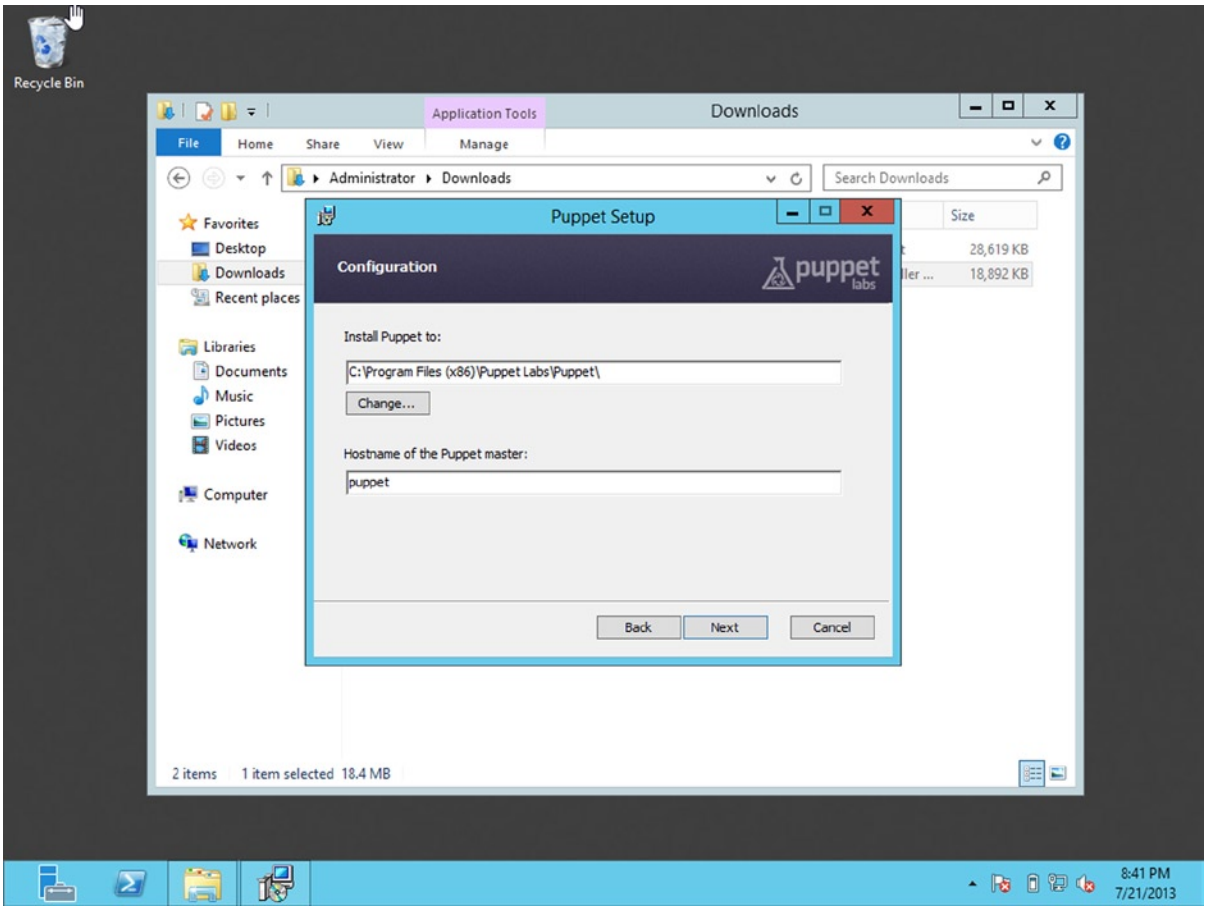

Figure 1-5. *Configuring Puppet*

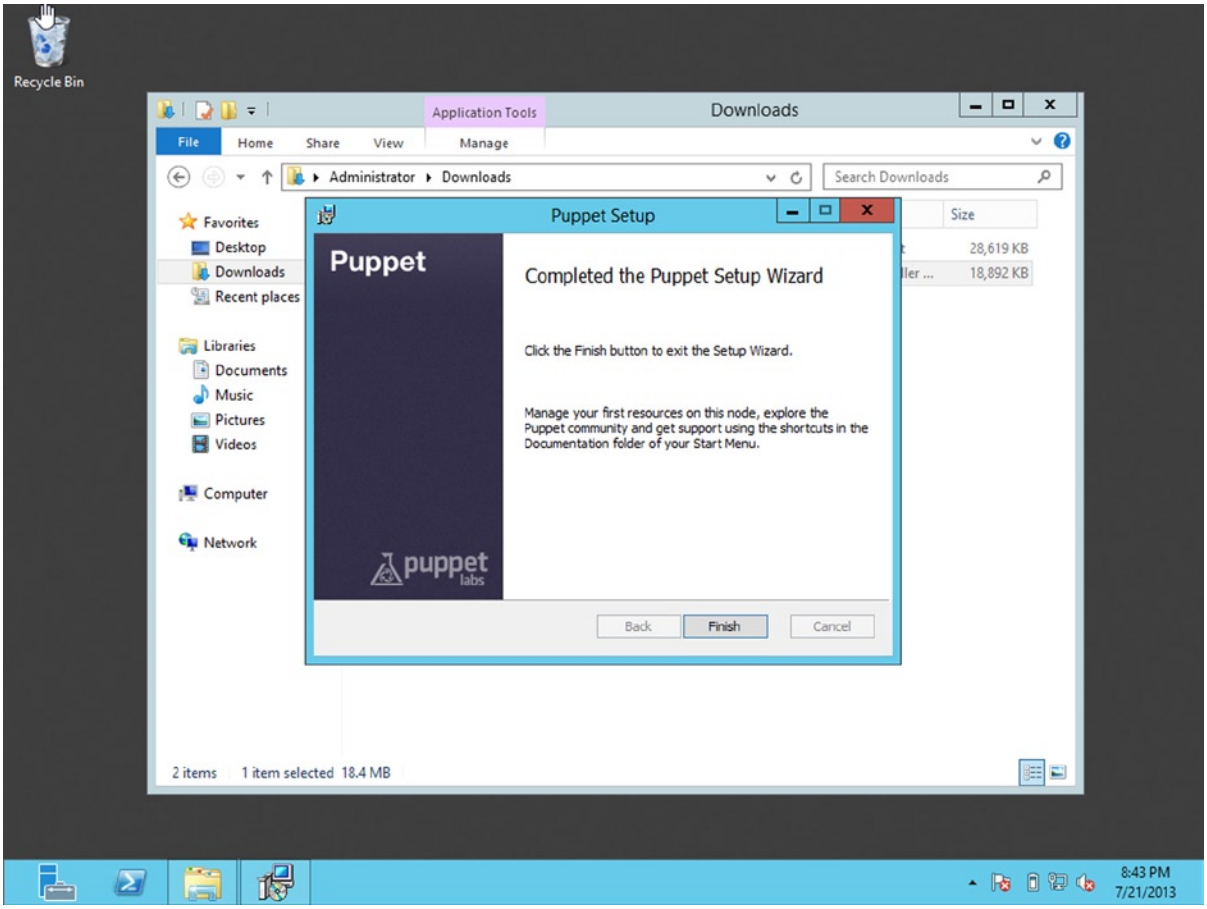

Figure 1-6. *Puppet installation is complete*

For additional information, such as automating installations, refer to the Windows installation documentation at http://docs.puppetlabs.com/windows/installing.html.

Installing on Microsoft Windows Using PowerShell

Many Windows administrators, particularly in cloud operations, have begun using PowerShell for remote administration and scriptable installation. To install open source Puppet, you must first download the Puppet MSI installer from Puppetlabs.com. There are two ways to do this; older PowerShell versions should use the commands in Listing 1-2, while version 3 and greater can use the command in Listing 1-3.

Listing 1-2. Downloading the Puppet MSI on Powershell versions 2 and earlier

```
$downloads = $pwd
$webclient = New-Object System.Net.WebClient
$url = http://puppetlabs.com/downloads/windows/puppet-3.2.3.msi
$file = "$downloads/puppet.msi"
$webclient.DownloadFile($url,$file)
```

PowerShell 3 and greater can use the `Invoke-WebRequest` commandlet to download the MSI, as shown in Listing 1-3.

Listing 1-3. Downloading the Puppet MSI on Powershell versions 3 and later

```
$url = "http://puppetlabs.com/downloads/windows/puppet-3.2.3.msi"
Invoke-WebRequest -Uri $url -OutFile puppet.msi
```

You will see a progress bar across the screen. After the MSI has been downloaded, install it using `msiexec`, as shown next. Here we actually shell out to `cmd.exe`, since `msiexec` doesn't currently work in PowerShell.

```
cmd /c "msiexec /qn /i puppet.msi /l*v install.log"
```

By using the `/qn` argument to `msiexec`, we have made the installation silent and noninteractive, meaning that no questions were asked of us and no dialog popped up. This entire exercise can be completed via remote PowerShell or script. The `/l*v install.log` argument has made the installation send its log to `install.log` in the current directory. If installation is successful, you should see "Installation Successful" at the end of the `install.log`.

We can verify that Puppet has been installed correctly by running the `puppet.bat` script as shown here:

```
& 'C:\Program Files (x86)\Puppet Labs\Puppet\bin\puppet.bat' --version
```

The MSI installer, when run in silent mode, will choose `puppet` as the Puppet master and CA server. If you want to override these variables, you can use the environment variables shown in Listing 1-4.

Listing 1-4. Configuring the puppet installation

```
cmd /c "msiexec /qn PUPPET_MASTER_SERVER=master.example.com
PUPPET_CA_SERVER=puppetcat.example.com /i puppet
.msi /l*v install.log"
```

Unfortunately, at the time of writing, you cannot set other configuration variables, so you would have to modify `puppet.conf` manually with notepad or another editor:

```
notepad 'C:\ProgramData\PuppetLabs\puppet\etc\puppet.conf'
```

Installing on the Mac

In this section we will cover installing Puppet on Mac OS X via the GUI and from the CLI.

Installing Puppet Open Source on Apple Mac OS X via the Graphical Installer

Download the Facter and Puppet `.dmg` files from the Puppet Labs website, `puppetlabs.com`.

Then mount the `.dmg` files and verify that you have two Apple Package installers as shown in Figure 1-7.

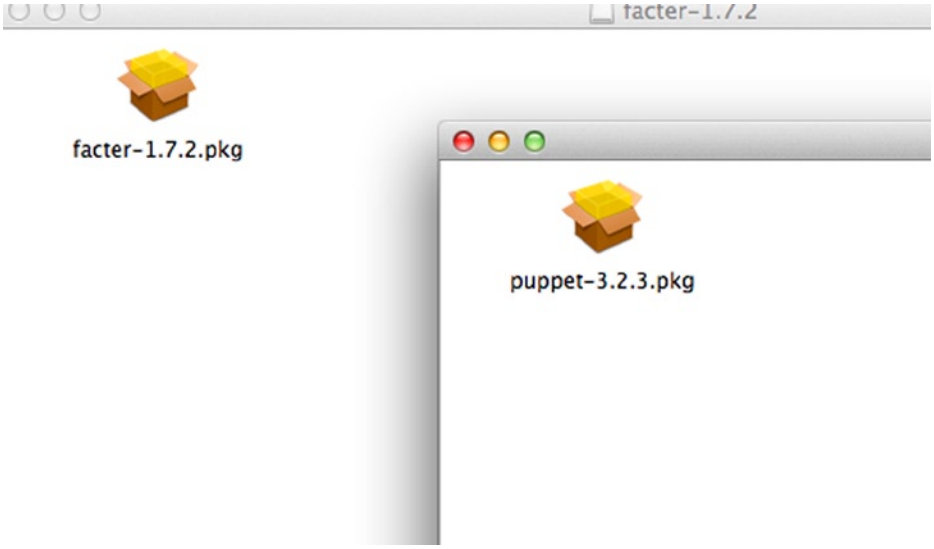

Figure 1-7. *Mac OSX pkg files*

Double-click the Facter cardboard box icon, which brings you to the welcome screen in Figure 1-8.

Figure 1-8. *The Facter Installer*

Assume administrator rights, as shown in Figure 1-9, and click Install Software.

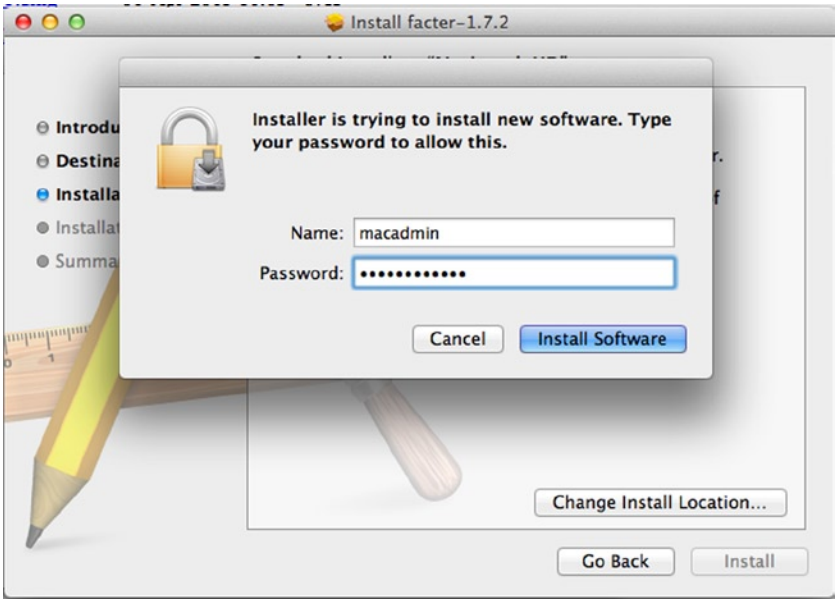

Figure 1-9. *Enter your administrator password*

Once the installation is complete, you'll see the screen in Figure 1-10.

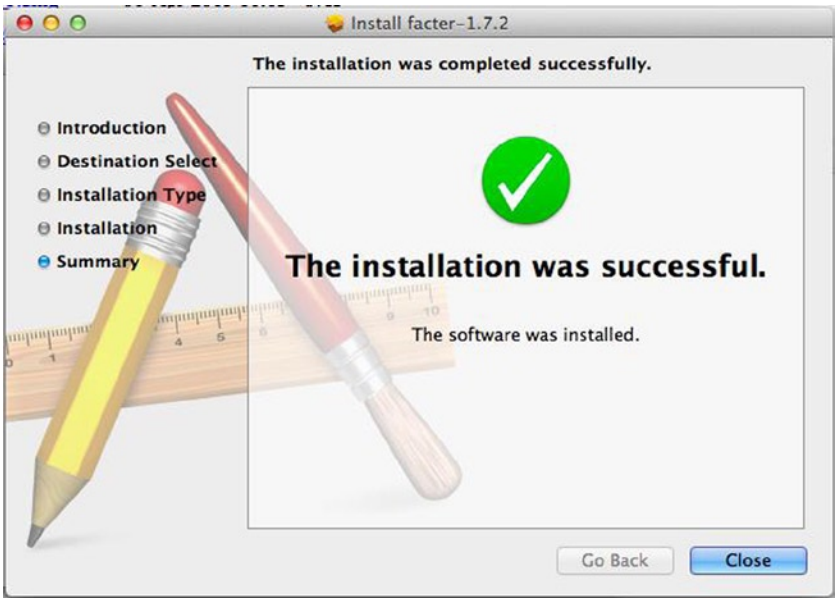

Figure 1-10. *Facter installation is complete*

Excellent! Facter is now installed. Now double-click the Puppet cardboard box to install it as well. The screen in Figure 1-11 will appear.

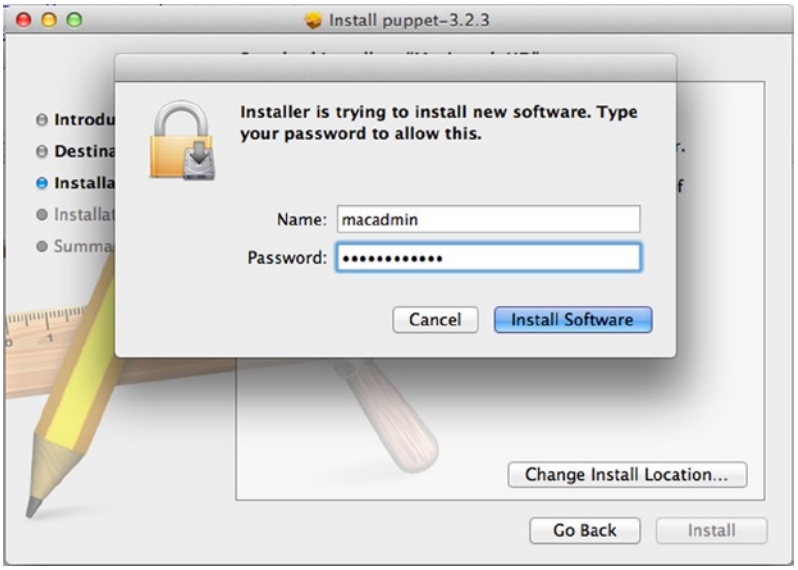

Figure 1-11. *Enter your adminstrator credentials*

Again, assume administrator rights and click the Install Software button (Figure 1-11).

Puppet is now installed! You can verify the installation with `puppet -version`, as shown in Figure 1-12. In this case, the Puppet installer did not prompt for a Puppet master server. If you want to use a server other than the DNS name `puppet`, you must create `/etc/puppet/puppet.conf` with `server=puppet-master.pro-puppet.com`.

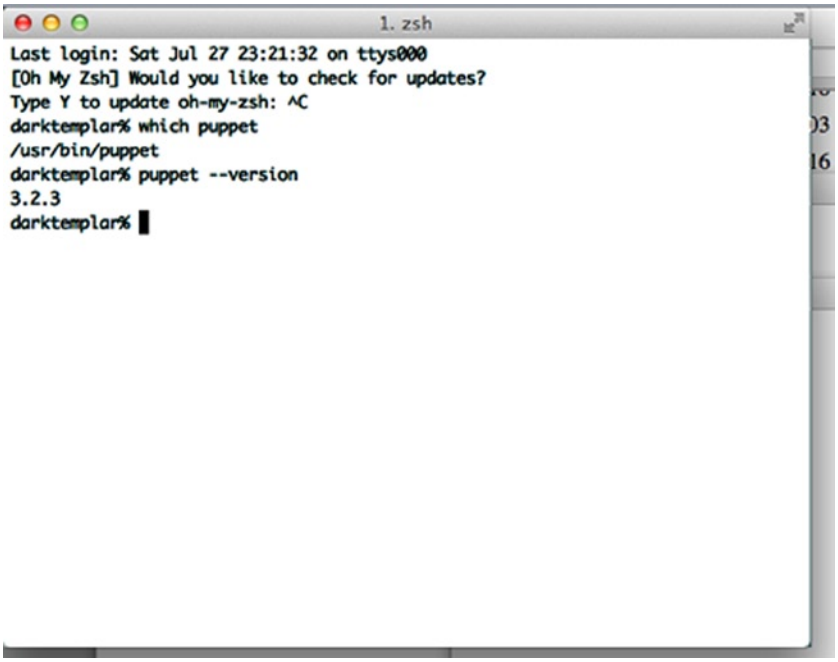

Figure 1-12. *Verifying installation with `puppet -version`*

Installing Puppet Open Source on Apple Mac OS X via the Command Line

Download the latest `facter` and `puppet` packages from http://downloads.puppetlabs.com/mac/.

Once you have downloaded the `.dmg` files, you can install them via the command line with the following instructions:

```
$ curl -O http://downloads.puppetlabs.com/mac/facter-1.7.2.dmg
$ hdiutil mount facter-1.7.2.dmg
$ installer -package /Volumes/facter-1.7.2/facter-1.7.2.pkg/ -target /Volumes/Macintosh\ HD
$ hdutil unmount /Volumes/facter-1.7.2
```

Next install Puppet:

```
$ curl -O  https://downloads.puppetlabs.com/mac/puppet-3.2.3.dmg
$ hdiutil mount puppet-3.2.3.dmg
$ installer -package /Volumes/puppet-3.2.3/puppet-3.2.3.pkg -target /Volumes/Macintosh\ HD
$ hdutil unmount /Volumes/puppet-3.2.3/
```

At this point you can run Puppet by `cron` or with `puppet apply`. To setup a `launchd` job to run it in daemon mode, refer to the official docs:

http://docs.puppetlabs.com/guides/installation.html#mac-os-x

Installing on Other Platforms

We've just explained how to install Puppet on some popular platforms.

Puppet can also be installed on a wide variety of other platforms, including the following:

- SLES/OpenSuSE via http://software.opensuse.org/
- Gentoo via Portage
- Mandrake and Mandriva via the Mandriva contrib repository
- FreeBSD via ports tree
- NetBSD via pkgsrc
- OpenBSD via ports tree
- ArchLinux via ArchLinux AUR

■ **Note** You can find a full list of additional operating systems and specific instructions at https://puppetlabs.com/misc/download-options.

Puppet can also work on some networks such as BIG-IP F5 devices and some Juniper network devices. F5s are an advanced configuration, configured by way of a proxy agent. Read https://puppetlabs.com/blog/managing-f5-big-ip-network-devices-with-puppet/ to get started configuring an F5 with Puppet. Some modern Juniper devices run Puppet natively. Puppet can be installed via a Juniper package called `jpuppet`. Downloads and more information are available at https://puppetlabs.com/solutions/juniper-networks/.

Puppet's tarball also contains some packaging artifacts in the `ext` directory; for example, there are an RPM spec file and OS X build scripts that can allow you to create your own packages for compatible operating systems. Now that you've installed Puppet on your chosen platform, we can start configuring it.

Configuring Puppet

Let's start by configuring a Puppet master that will act as our configuration server. We'll look at Puppet's configuration files, how to configure networking and firewall access, and how to start the Puppet master. Remember that we're going to be looking at Puppet in its client-server mode. Here, the Puppet master contains our configuration data, and Puppet agents connect via SSL and pull down the required configuration (refer back to Figure 1-2).

On most platforms, Puppet's configuration will be located under the `/etc/puppet` directory. Puppet's principal configuration file is called `puppet.conf` and is stored at `/etc/puppet/puppet.conf` on Unix/Linux operating systems and `C:\ProgramData\PuppetLabs\puppet\etc\` on Windows. It is likely that this file has already been created when you installed Puppet, but if it hasn't, you can create a simple file using the following command:

```
$ cd /etc/puppet/
$ puppet master --genconfig > puppet.conf
```

■ **Note** We're assuming your operating system uses the `/etc/` directory to store its configuration files, as most Unix/Linux operating systems and distributions do. If you're on a platform that doesn't, such as Microsoft Windows, substitute the location of your `puppet.conf` configuration file. But remember that the Puppet master cannot be run on Windows.

The `puppet.conf` configuration file is constructed much like an INI-style configuration file and divided into sections. Each section configures a particular element of Puppet. For example, the `[agent]` section configures the Puppet agent, and the `[master]` section configures the Puppet master binary. There is also a global configuration section called `[main]`. All components of Puppet set options specified in the `[main]` section.

At this stage, we're only going to add one entry, `server`, to the `puppet.conf` file. The `server` option specifies the name of the Puppet master. We'll add the server value to the `[main]` section (if the section doesn't already exist in your file, then create it).

```
[main]
server=puppet.example.com
```

Replace `puppet.example.com` with the fully qualified domain name of your host.

■ **Note** We'll look at other options in the `puppet.conf` file in later chapters.

We recommend you also create a DNS CNAME for your Puppet master host, for example `puppet.pro-puppet.com`, and add it to either your `/etc/hosts` file or your DNS configuration:

```
# /etc/hosts
127.0.0.1 localhost
192.168.0.1 puppet.pro-puppet.com puppet
```

Once we've configured appropriate DNS for Puppet, we need to add the `site.pp` file, which holds the basics of the configuration items we want to manage.

The site.pp File

The `site.pp` file tells Puppet where and what configuration to load for our clients. We're going to store this file in a directory called `manifests` under the `/etc/puppet` directory.

■ **Note** *Manifest* is Puppet's term for files containing configuration information. Manifest files have a suffix of `.pp`. The Puppet language is written into these files.

This directory and file is often already created when the Puppet packages are installed. If it hasn't already been created, create this directory and file now:

```
# mkdir /etc/puppet/manifests
# touch /etc/puppet/manifests/site.pp
```

We'll add some configuration to this file later in this chapter, but now we just need the file present.

■ **Note** You can also override the name and location of the `manifests` directory and `site.pp` file using the `manifestdir` and `manifest` configuration options, respectively. These options are set in the `puppet.conf` configuration file in the `[master]` section. See http://docs.puppetlabs.com/references/stable/configuration.html for a full list of configuration options. We'll talk about a variety of other options throughout this book.

Firewall Configuration

The Puppet master runs on TCP port 8140. This port needs to be open on your master's firewall (and any intervening firewalls and network devices), and your client must be able to route and connect to the master. To do this, you need to have an appropriate firewall rule on your master, such as the following rule for the Netfilter firewall:

```
$ iptables -A INPUT -p tcp -m state --state NEW --dport 8140 -j ACCEPT
```

This line allows access from everywhere to TCP port 8140. If possible, you should limit this to networks that require access to your Puppet master. For example:

```
$ iptables -A INPUT -p tcp -m state --state NEW -s 192.168.0.0/24 --dport 8140 -j ACCEPT
```

Here we've restricted access to port 8140 to the 192.168.0.0/24 subnet.

■ **Note** You can create similar rules for other operating systems' firewalls, such as pf or the Windows Firewall. The traffic between Puppet client and Puppet master is encrypted with SSL and authenticated by client x509 certificates.

Starting the Puppet Master

The Puppet master can be started via an init script or other init system, such as `upstart` or `systemd` on most Linux distributions. On Red Hat or Debian, we would run the init script with the `service` command, like so:

```
# service puppetmaster start
```

Other platforms should use their appropriate service management tools.

■ **Note** Output from the daemon can be seen in `/var/log/messages` on Red Hat–based hosts and `/var/log/syslog` on Debian and Ubuntu hosts. Puppet will log via the daemon facility to Syslog by default on most operating systems. You will find output from the daemons in the appropriate location and files for your operating system. On Microsoft Windows, Puppet logs go to `C:\ProgramData\PuppetLabs\puppet\var\log`.

Starting the daemon will initiate your Puppet environment, create a local Certificate Authority (CA), along with certificates and keys for the master, and open the appropriate network socket to await client connections. You can see Puppet's SSL information and certificates in the `/var/lib/puppet/ssl` directory.

```
# ls -l /var/lib/puppet/ssl/
drwxrwx--- 5 puppet puppet 4096 Apr 11 04:05 ca
drwxr-xr-x 2 puppet root   4096 Apr 11 04:05 certificate_requests
drwxr-xr-x 2 puppet root   4096 Apr 11 04:05 certs
-rw-r--r-- 1 puppet puppet  918 Apr 11 04:05 crl.pem
drwxr-x--- 2 puppet root   4096 Apr 11 04:05 private
drwxr-x--- 2 puppet root   4096 Apr 11 04:05 private_keys
drwxr-xr-x 2 puppet root   4096 Apr 11 04:05 public_keys
```

The directory on the master contains your CA, certificate requests from your clients, a certificate for your master, and certificates for all your clients.

■ **Note** You can override the location of the SSL files using the `ssldir` option in `puppet.conf` on the master. There will be much more on the Puppet internal CA in Chapter 4.

You can also run the Puppet master from the command line to help test and debug issues. We recommend doing this when testing Puppet initially. To do this, we start the Puppet master daemon like so:

```
# puppet master --verbose --no-daemonize
```

The `--verbose` option outputs verbose logging and the `--no-daemonize` option keeps the daemon in the foreground and redirects output to standard out. You can also add the `--debug` option to produce more verbose debug output from the daemon.

A SINGLE BINARY

All the functionality of Puppet is available from a single binary, puppet, in the style of tools like Git. This means you can start the Puppet master by running this command:

```
# puppet master
```

The agent functionality is also available in the same way:

```
# puppet agent
```

You can see a full list of the available functionality from the `puppet` binary by running `help`:

```
$ puppet help
```

And you can get help on any Puppet subcommand by adding the `subcommand` option:

```
$ puppet help subcommand
```

Connecting Our First Agent

Once you have the Puppet master configured and started, you can configure and initiate your first agent. On the agent, as we mentioned earlier, you need to install the appropriate packages, usually `puppet` and `facter`, using your operating system's package management system. We're going to install a client on a host called `node1.pro-puppet.com` and then connect to our `puppet.pro-puppet.com` master.

When connecting our first client, we want to run the Puppet agent from the command line rather than as a service. This will allow us to see what is going on as we connect. The Puppet agent daemon is run using `puppet agent`, and you can see a connection to the master initiated in Listing 1-5.

Listing 1-5. Puppet client connection to the Puppet master

```
node1# puppet agent --test --server=puppet.pro-puppet.com
Info: Creating a new SSL key for node1.pro-puppet.com
Info: Caching certificate for ca
Info: Creating a new SSL certificate request for node1.pro-puppet.com
Info: Certificate Request fingerprint (SHA256): 6F:0D:41:14:BD:2D:FC:CE:1C:DC:11:1E:26:07:4C:08:D0:C
7:E8:62:A5:33:E3:4B:8B:C6:28:C5:C8:88:1C:C8
Exiting; no certificate found and waitforcert is disabled
```

In Listing 1-5, we executed the Puppet agent with three options. The first option, `--server`, specifies the name or address of the Puppet master to connect to.

■ **Tip** You can also run a Puppet client on the Puppet master, but we're going to start with the more traditional client-server approach. And yes, that means you can use Puppet to manage itself!

■ **Tip** If we don't specify a server, Puppet will look for a host called "puppet." It's often a good idea to create a CNAME for your Puppet master, such as puppet.pro-puppet.com. Additionally, Puppet has the ability to query SRV records to find where the Puppet master and Puppet CA servers are. More on this in Chapter 4.

We can also specify this in the main section of the /etc/puppet/puppet.conf configuration file on the client:

```
# /etc/puppet/puppet.conf
[main]
server=puppet.pro-puppet.com
```

Your client must be able to resolve the hostname of the master to connect to (this is why it is useful to have a Puppet CNAME or to specify your Puppet master in the /etc/hosts file on your client).The --test option runs the Puppet client in the foreground, outputs to standard out, and exits after the run is complete. By default, the Puppet client runs as a daemon, and the puppet agent command forks off the Puppet daemon into the background and exits immediately.

In Listing 1-5, you can see the output from our connection. The agent has created a certificate signing request and a private key to secure our connection. Puppet uses SSL certificates to authenticate connections between the master and the agent. The agent sends the certificate request to the master and waits for the master to sign and return the certificate.

At this point, the agent has exited after sending in its Certificate Signing Request (CSR). The agent will need to be rerun to check in and run Puppet after the CSR has been signed by the CA. You can configure puppet agent not to exit, but instead stay alive and poll periodically for the CSR to be signed. This configuration is called waitforcert and is generally only useful if you are also auto-signing certificates on the master. More on that later in this chapter.

■ **Note** You can change the time the Puppet agent will wait by using the --waitforcert option. You can specify a time in seconds or 0 to not wait for a certificate, in which case the agent will exit.

Completing the Connection

To complete the connection and authenticate our agent, we now need to sign the certificate the agent has sent to the master. We do this using puppet cert (or the puppetca binary) on the master:

```
puppet# puppet cert list
"node1.pro-puppet.com" (SHA256) 6F:0D:41:14:BD:2D:FC:CE:1C:DC:11:1E:26:07:4C:08:D0:C7:E8:62:A5:33:E3
:4B:8B:C6:28:C5:C8:88:1C:C8
```

■ **Tip** You can find a full list of the binaries that come with Puppet at http://docs.puppetlabs.com/guides/tools.html.

The list option displays all the certificates waiting to be signed. We can then sign our certificate using the sign option:

```
puppet# puppet cert sign node1.pro-puppet.com
Notice: Signed certificate request for node1.pro-puppet.com
Notice: Removing file Puppet::SSL::CertificateRequest node1.pro-puppet.com at
'/var/lib/puppet/ssl/ca/requests/node1.pro-puppet.com.pem'
```

You can sign all waiting certificates with the `puppet cert sign --all` command.

■ **Note** Rather than signing each individual certificate, you can also enable *autosign* mode. In this mode, all incoming connections from specified IP addresses or address ranges are automatically signed. This obviously has some security implications and should only be used if you are comfortable with it. You can find more details at http://docs.puppetlabs.com/guides/faq.html#why-shouldn-t-i-use-autosign-for-all-my-clients.

On the client, two minutes after signing the certificate, you should see the following entries (or you can stop and restart the Puppet agent rather than waiting two minutes):

```
# puppet agent --test
Info: Retrieving plugin
Info: Caching catalog for node1.pro-puppet.com
Info: Applying configuration version '1365655737'
Notice: Finished catalog run in 0.13 seconds
```

The agent is now authenticated with the master, and you may have another message present:

```
# puppet agent -t
Info: Retrieving plugin
Error: Could not retrieve catalog from remote server: Error 400 on SERVER: Could not find default
node or by name with 'node1.example.com, node1' on node node1.pro-puppet.com
Warning: Not using cache on failed catalog
Error: Could not retrieve catalog; skipping run
```

The agent has connected and our signed certificate has authenticated the session with the master. The master, however, doesn't have any configuration available for our puppet node, node1.pro-puppet.com, and hence we have received an error message. We now have to add some configuration for this agent on the master.

■ **Caution** It is important for the time to be accurate on your master and agent. SSL connections rely on the clock on hosts being correct. If the clocks are incorrect, your connection may fail with an error indicating that your certificates are not trusted. You should use something like NTP (Network Time Protocol) to ensure that your host's clocks are accurate. A quick way to sync several servers is to run `ntpdate bigben.cac.washington.edu` on each of them. This will perform a one-time NTP sync.

Creating Our First Configuration Item

Let's get some more understanding of Puppet's components, configuration language, and capabilities. You learned earlier that Puppet describes the files containing configuration data as manifests. Puppet manifests are made up of a number of major components:

- Resources: Individual configuration items
- Files: Physical files you can serve out to your agents
- Templates: Template files that you can use to populate files

- Nodes: Specifies the configuration of each agent
- Classes: Collections of resources
- Definitions: Composite collections of resources

These components are wrapped in a configuration language that includes variables, conditionals, arrays, and other features. Later in this chapter we'll introduce you to the basics of the Puppet language and its elements. In the next chapter, we'll extend your knowledge of the language by taking you through an implementation of a multi-agent site managed with Puppet.

In addition to these components, Puppet also has the concept of a "module," which is a portable collection of manifests that contain resources, classes, definitions, files, and templates. We'll see our first module shortly.

Adding a Node Definition

Let's add our first node definition to `site.pp`. In Puppet manifests, agents are defined using `node` statements.

You can see the node definition we're going to add in Listing 1-6.

Listing 1-6. Our node configuration

```
node 'node1.pro-puppet.com' {
    package { 'vim':
      ensure => present,
    }
}
```

For a node definition we specify the node name, enclosed in single quotes, and then specify the configuration that applies to it inside curly braces { }. The client name can be the hostname or the fully qualified domain name of the client. At this stage, you can't specify nodes with wildcards (for example, `*.pro-puppet.com`), but you can use regular expressions, as shown here:

```
node /^www\d+\.pro-puppet\.com/ {
   ...
}
```

This example will match all nodes from the domain `pro-puppet.com` with the hostnames `www1`, `www12`, `www123`, and so on.

Next, we specify a resource stanza in our node definition. This is the same one from earlier when we were introducing the Puppet DSL. It will make sure that the `vim` package is installed on the host `node1.pro-puppet.com`.

We can run Puppet on `node1` and see what action it has performed:

```
root@node1:~# puppet agent --test
Info: Retrieving plugin
Info: Caching catalog for node1.pro-puppet.com
Info: Applying configuration version '1375079547'
Notice: /Stage[main]//Node[node1]/Package[vim]/ensure: ensure changed 'purged' to 'present'
Notice: Finished catalog run in 4.86 seconds
```

We can see that Puppet has installed the `vim` package. It is not generally best practice to define resources at node level; those belong in classes and modules. Let's strip out our `vim` resource and include the `sudo` class instead (Listing 1-7).

Listing 1-7. Our Node Configuration

```
node 'node1.pro-puppet.com' {
  include sudo
}
```

Here we specify an `include` directive in our node definition; it specifies a collection of configurations, called a *class*, that we want to apply to our host. There are two ways to include a class:

```
node /node1/ {
  include ::sudo
}

node /node2/ {
  class { '::sudo':
     users => ['tom', 'jerry'],
  }
}
```

The first syntax is bare and simple. The second syntax allows *parameters* to be passed into the class. This feature, generally called *parameterized classes*, allows classes to be written generally and then utilized specifically, increasing the reusability of Puppet code. Notice that the syntax for including a class is very similar to the syntax for a normal Puppet resource. *Modules* are self-contained collections of Puppet code, manifests, Puppet classes, files, templates, facts, and tests, all for a specific configuration task. Modules are usually highly reusable and shareable. The double-colon syntax explicitly instructs Puppet to use top scope to look up the `sudo` module. You will learn much more about this in Chapter 2.

■ **Note** Puppet also has an inheritance model in which you can have one node inherit values from another node. You should avoid doing this. Refer to `http://docs.puppetlabs.com/puppet/latest/reference/lang_node_definitions.html#inheritance` for more information.

Creating Our First Module

The next step in our node configuration is to create a `sudo` module. Again, a module is a collection of manifests, resources, files, templates, classes, and definitions. A single module would contain everything required to configure a particular application. For example, it could contain all the resources (specified in manifest files), files, and associated configuration to configure Apache or the `sudo` command on a host. We will create a `sudo` module and a `sudo` class.

Each module needs a specific directory structure and a file called `init.pp`. This structure allows Puppet to automatically load modules. To perform this automatic loading, Puppet checks a series of directories called the *module path*. This path is configured with the `modulepath` configuration option in the `[master]` section of the `puppet.conf` file. By default, Puppet looks for modules in the `/etc/puppet/modules` and `/usr/share/puppet/modules` directories, but you can add additional locations if required:

```
[master]
modulepath = /etc/puppet/modules:/var/lib/puppet/modules:/opt/modules
```

Module Structure

Let's start by creating a module directory and file structure in Listing 1-8. We're going to create this structure under the directory `/etc/puppet/modules`. We will name the module `sudo`. Modules (and classes) must be normal words containing only letters, numbers, underscores, and dashes.

Listing 1-8. Module structure

```
# mkdir -p /etc/puppet/modules/sudo/{files,templates,manifests}
# touch /etc/puppet/modules/sudo/manifests/init.pp
```

The `manifests` directory will hold our `init.pp` file and any other configuration. The `init.pp` file is the core of your module, and every module should have one. The `files` directory will hold any files we wish to serve as part of our module. The `templates` directory will contain any templates that our module might use.

The init.pp file

Now let's look inside our `sudo` module, starting with the `init.pp` file, which we can see in Listing 1-9.

Listing 1-9. The `sudo` module's init.pp file

```
class sudo {

  package { 'sudo':
    ensure => present,
  }

  if $::osfamily == 'Debian' {
    package { 'sudo-ldap':
      ensure  => present,
      require => Package['sudo'],
    }
  }

  file { '/etc/sudoers':
    owner   => 'root',
    group   => 'root',
    mode    => '0440',
    source  => "puppet://$::server/modules/sudo/etc/sudoers",
    require => Package['sudo'],
  }
}
```

Our `sudo` module's `init.pp` file contains a single class, also called `sudo`. There are three resources in the class, two packages and a file resource. The first package resource ensures that the `sudo` package is installed, `ensure => present`. The second package resource uses Puppet's if/else syntax to set a condition on the installation of the `sudo-ldap` package.

■ **Note** Puppet also has two other conditional statements, a case statement and a selector syntax. You can see more details of Puppet's conditional syntaxes at http://docs.puppetlabs.com/guides/more_language.html#conditionals.
