Pro Linux System Administration

James Turnbull, Peter Lieverdink,
Dennis Matotek

Apress®

Pro Linux System Administration

ISBN-13 (pbk): 978-1-4302-1912-5

ISBN-13 (electronic): 978-1-4302-1913-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Contributors: Sander van Vugt, Donna Benjamin
Lead Editors: Michelle Lowman, Frank Pohlmann
Technical Reviewer: Jaime Sicam
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editors: Ami Knox, Nicole Flores
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Kinetic Publishing Services, LLC
Proofreaders: April Eddy, Dan Shaw
Indexer: BIM Indexing & Proofreading Services
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 19

Configuration Management

By James Turnbull

In this chapter, we're going to look at two facets of configuration management:

- Automated provisioning and installation of new hosts
- Automated management of your configuration including files, users, and packages

The first process we're going to examine, automated provisioning or installation of new hosts, is sometimes called *bootstrapping*. In the Red Hat world, bootstrapping is often referred to as *kickstarting* (after the Kickstart tool used to perform it). On Ubuntu and Debian, the process is called *preseeding*.

Provisioning is a way of automatically installing a distribution to a host. When we first looked at installing distributions in Chapter 2, we demonstrated how to do it manually. You inserted a DVD and followed the onscreen prompts to install your distribution. Automated provisioning is a way of installing a distribution without being prompted by the configuration questions. This makes provisioning quick and simple, and it also has the advantage of ensuring every build is identical.

Tip You can use provisioning for both server hosts and desktop hosts. Not only is it a quick way of building (or rebuilding) server hosts, but it can also be a quick way to automatically install desktops for your users.

The second process we're going to examine is configuration management and automation. By now you've seen that you can accumulate a lot of installed packages, users, configuration files, and other settings. Your environment can quickly get complicated and difficult to manage if you don't take steps to control and automate it. Configuration management allows you to centralize your configuration, document it, and automate it. This allows you to manage and control changes to your environment and protects you against accidental or malicious configuration changes.

Both provisioning and configuration management are particularly useful if you have deployed a lot of hosts, but they are also useful in smaller environments, to save time and effort in managing your hosts.

Provisioning

We've talked a little about what provisioning is, but how you go about it varies between distributions. We are going to explain how to automatically provision both Red Hat and Ubuntu hosts.

Provisioning is usually a two-stage process:

1. Boot your host and send it the files required for installation.
2. Automate the installation steps.

The process starts with a host booting up. Remember in Chapter 5 when we told you about the boot sequence? On many hosts, you can configure that boot sequence to look in alternative places to get its boot instructions, for example, boot from a DVD or a USB stick. In addition to these methods, you can also get your boot instructions from a network source.

The technology behind this boot process is called Preboot Execution Environment (PXE). A network boot server is hence called a PXE boot (pronounced "pixie") server. The host that we intend to build uses a network query to find a PXE boot server, usually a network query to a DHCP server, that might offer it the files required to boot and then transfers those files to the host using a file transfer protocol called Trivial File Transfer Protocol (TFTP).

■ **Note** You can read more about PXE at `http://en.wikipedia.org/wiki/Preboot_Execution_Environment`.

Once this initial boot takes place, your provisioning process continues by installing a prepackaged version of your distribution, usually with a series of automated scripted responses to the various configuration questions you are prompted for when installing.

■ **Note** We're using network-based provisioning to create our hosts rather than any of the alternatives, such as CD or DVD. This is because we believe network-based provisioning is the simplest, easiest, and most efficient way to automatically build hosts.

In this chapter, we're going to introduce you to some useful tools. For Red Hat provisioning, we're going to look at Cobbler, an automated build framework. Cobbler also makes use of Red Hat's installation automation tool, Kickstart. For Ubuntu, we're going to show you how to set up a network boot server and also how to use Kickstart (supplemented with some elements of Ubuntu's Preseed provisioning tool) to build your hosts.

Provisioning with Red Hat Cobbler

Red Hat has a variety of tools for provisioning hosts, ranging from the most basic, Kickstart, which automates installations, to full-featured GUI management tools for host configuration such as Cobbler (`https://fedorahosted.org/cobbler/`), Spacewalk (`http://www.redhat.com/spacewalk/`), and Genome (`http://genome.et.redhat.com/`).

We're going to look at a combination of two tools:

- *Kickstart*: An installation automation tool
- *Cobbler*: A provisioning server that provides a PXE boot server

We'll take you through the process of creating a Cobbler server and a build to install. Later in this chapter, we'll show you how to configure Kickstart to automate your configuration and installation options.

Installing Cobbler

Let's start by installing Cobbler on your host. To run Cobbler, you need some prerequisite packages:

```
$ sudo yum install yum-utils createrepo dhcp tftp-server httpd
```

Here we've installed some additional Yum utilities and the `createrepo` package, which assist in repository management. We've also installed some additional packages Cobbler uses: the DHCP daemon, a TFTP server, and the Apache web server. You may already have these packages installed, in which case Yum will skip them.

■ **Note** We talk about DHCP in Chapter 9 and Apache in Chapter 11.

You next need to install the latest version of Cobbler and another required package, `python-cheetah`, which assists with Kickstart configuration. You can download and install these packages from the Fedora EPEL repository.

```
$ sudo rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/cobbler-1.4.1-➥
1.el5.noarch.rpm http://download.fedora.redhat.com/pub/epel/5/i386/➥
python-cheetah-2.0.1-1.el5.i386.rpm
```

Or you can add the EPEL repository to your host and then install packages from that repository. You add the EPEL repository, if it's not already added to your Yum configuration, by adding the `epel-release` RPM.

```
$ sudo rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/➥
epel-release-5-3.noarch.rpm
```

You can then install the `cobbler` package.

```
$ sudo yum install cobbler
```

Configuring Cobbler

After you've installed the required packages, you need to configure Cobbler. Cobbler comes with a very handy check function that tells you what needs to be done to configure it. To see what needs to be done, run the following:

```
$ sudo cobbler check
The following potential problems were detected:
#0: The 'server' field in /etc/cobbler/settings must be set to something other than➥
 localhost, or kickstarting features will not work.  This should be a resolvable ➥
hostname or IP for the boot server as reachable by all machines that will use it.
#1: For PXE to be functional, the 'next_server' field in /etc/cobbler/settings must➥
 be set to something other than 127.0.0.1, and should match the IP of the boot ➥
server on the PXE network.
#2: service cobblerd is not running
#3: change 'disable' to 'no' in /etc/xinetd.d/tftp
#4: since iptables may be running, ensure 69, 80, 25150, and 25151 are unblocked
#5: fencing tools were not found, and are required to use the (optional) power ➥
management features. install cman to use them
```

You can see there are a few things you need to do to get Cobbler running. Let's work through each of these issues.

First, you configure the `/etc/cobbler/settings` file. You need to update two fields in this file, `server` and `next_server`. You need to replace the existing values (usually 127.0.0.1) with the IP address of your host, so a PXE-booted host can find your Cobbler host. In our case, we specify the following:

```
server 192.168.0.1
next_server 192.168.0.0.1
```

To update Cobbler's configuration, you then run this:

```
$ sudo cobbler sync
```

■ **Note** You need to run the `$ sudo cobbler sync` command anytime you change the `/etc/cobbler/settings` file. Common errors include leaving trailing spaces after options in the `settings` file. Make sure you delete any extra spaces from the file.

You also need to configure a DHCP server (like the one we introduced in Chapter 9). You have two choices here: you can get Cobbler to manage your existing DHCP server or you can tell your existing DHCP server to point to Cobbler.

Cobbler Managing Your DHCP

If you want to enable Cobbler to manage your DHCP server, then you need to enable another option in the `/etc/cobbler/settings` file:

```
manage_dhcp: 1
```

You also need to update a template file that Cobbler will use to configure your DHCP server, `/etc/cobbler/dhcp.template`. Listing 19-1 shows an example of this file.

Listing 19-1. *The /etc/cobbler/dhcp.template File*

```
# ******************************************************************
# Cobbler managed dhcpd.conf file
#
# generated from cobbler dhcp.conf template ($date)
# Do NOT make changes to /etc/dhcpd.conf. Instead, make your changes
# in /etc/cobbler/dhcp.template, as /etc/dhcpd.conf will be
# overwritten.
#
# ******************************************************************

allow booting;
allow bootp;

ddns-update-style interim;
ddns-ttl 3600;
default-lease-time 600;
max-lease-time 7200;
log-facility local7;

ignore client-updates;
set vendorclass = option vendor-class-identifier;

key dynamic-update-key {
    algorithm hmac-md5;
    secret "3PDRnypPtzJqpbQvbw/B7bhPuHqpUeOSdi95Z4Ez/IzhS61dzcK6MJ6CdFHkkegp➥
TN1kmXOM6GggRNE24aPmOw==";
}

zone 0.168.192.in-addr.arpa. {
    key dynamic-update-key;
    primary 192.168.0.1;
}

zone example.com. {
    key dynamic-update-key;
    primary 192.168.0.1;
}

subnet 192.168.0.0 netmask 255.255.255.0 {
    option routers 192.168.0.254;
    option domain-name "example.com";
    option domain-name-servers 192.168.0.1;
    option broadcast-address 192.168.0.255;
```

```
    next-server  $next_server;
    filename "/pxelinux.0";
    group "static" {
        use-host-decl-names on;
        host au-mel-rhel-1 {
            hardware ethernet 00:16:3E:15:3C:C2;
            fixed-address au-mel-rhel-1.example.com;
        }
    }
    pool {
        range 192.168.0.101 192.168.0.150;
        deny unknown clients;
    }
    pool {
        range 192.168.0.151 192.168.0.200;
        allow unknown clients;
        default-lease-time 7200;
        max-lease-time 21600;
    }
}
```

If you have an existing DHCP server with a configuration, you should update this template to reflect that configuration. You can see we've adjusted the template in Listing 19-1 to reflect the DHCP configuration we used in Chapter 9. We've added two settings:

```
allow booting;
allow bootp;
```

These two options tell the DHCP server to respond to queries from hosts who request network boots.

The other two important settings to note in Listing 19-1 are the `next-server` and `filename` configuration options. The `next-server` option is set to `$next_server`. This value will be replaced by the IP address we just configured in the `next_server` option in the `/etc/cobbler/settings` file. This tells our DHCP server where to route hosts who request a net boot.

The `filename` option is set to `/pxelinux.0`, which is the name of the boot file PXE-booted hosts should look for to start their boot process. We'll set up this file shortly.

Now, after changing these files, you need to run the following command:

```
$ sudo cobbler sync
```

■ **Caution** If you have an existing DHCP server, this template will *overwrite* its configuration by overwriting the `/etc/dhcpd.conf` configuration file. Only do this if you are sure you know what you are doing, and make a copy of your existing `/etc/dhcpd.conf` file *before* running the command.

Cobbler Not Managing Your DHCP

If you don't want Cobbler to manage your DHCP, then you just need to adjust your existing DHCP configuration file, `/etc/dhcpd.conf`, to add the `next-server` and `filename` options. Let's update the relevant portions of the configuration we created in Chapter 9 with this option, as shown in Listing 19-2.

Listing 19-2. *Existing dhcpd.conf Configuration File*

```
allow booting;
allow bootp;

subnet 192.168.0.0 netmask 255.255.255.0 {
    option routers 192.168.0.254;
    option domain-name "example.com";
    option domain-name-servers 192.168.0.1;
    option broadcast-address 192.168.0.255;
    filename "/pxelinux.0";
    next-server 192.168.0.1;
    group "static" {
        use-host-decl-names on;
        host au-mel-rhel-1 {
            hardware ethernet 00:16:3E:15:3C:C2;
            fixed-address au-mel-rhel-1.example.com;
        }
    }
    pool {
        range 192.168.0.101 192.168.0.150;
        deny unknown clients;
    }
    pool {
        range 192.168.0.151 192.168.0.200;
        allow unknown clients;
        default-lease-time 7200;
        max-lease-time 21600;
    }
}
```

You can see we've added two options to the start of the DHCP section:

```
allow booting;
allow bootp;
```

These two options tell the DHCP server to respond to queries from booting clients.

We've also added the `next-server` option to our subnet definition.

```
next-server 192.168.0.1
```

The `next-server` option tells DHCP where to send hosts who request a PXE network boot. We need to specify the IP address of our Cobbler server.

Lastly, we've added the `filename` option, set to `/pxelinux.0`, which is the name of the boot file PXE-booted hosts should look for to start their boot process. We'll set up this file shortly.

■ **Tip** After configuring your DHCP server, you will need to restart the Cobbler server for the new configuration to be applied.

Starting Cobbler and Apache

Next, you need to start the Cobbler daemon:

```
$ sudo service cobblerd start
```

You also need to ensure the Apache web server is started:

```
$ sudo service httpd start
```

Configuring TFTP

Once the daemon is started, you need to enable your TFTP server to send your boot file to the host to be installed. To do this, you edit the `/etc/xinet.d/tftp` file to enable a TFTP server. Inside this file find this line:

```
disable = yes
```

and change it to this:

```
disable = no
```

Next, you enable the TFTP server like so:

```
$ sudo chkconfig tftp on
```

■ **Note** You don't need to start the service because it runs under the `xinetd` service that is already running. If you run the `chkconfig --list` command, you can see the `xinetd` services at the end of the listing, and you can also see that the `tftp` service is enabled.

You need to ensure your hosts can connect to the Cobbler server through your firewall by opening some required ports, 69, 80, 25150, and 25151, for example, by creating `iptables` rules such as the following:

```
-A Firewall-eth0-INPUT -s 192.168.0.0/24 -p udp -m state --state NEW --dport 69 -j ➥
ACCEPT
-A Firewall-eth0-INPUT -p tcp -m state --state NEW --dport 80 -j ACCEPT
-A Firewall-eth0-INPUT -s 192.168.0.0/24 -p tcp -m state --state NEW --dport 25150 ➥
-j ACCEPT
```

```
-A Firewall-eth0-INPUT -s 192.168.0.0/24 -p tcp -m state --state NEW --dport 25151 ➥
-j ACCEPT
```

These rules allow access for any host on the 192.168.0.0/24 subnet to the boot server on the appropriate ports. You can find more information on firewall rules in Chapter 6.

Using Cobbler

Once you've configured Cobbler, you can start to make use of it. Cobbler allows you to specify a distribution you'd like to build hosts with, imports that distribution's files, and then creates a profile. You can then build hosts using this distribution and profile.

Let's start by creating our first profile using the `import` command.

```
$ sudo cobbler import --mirror=/media/cdrom --name=RHEL5 --arch=i386
```

You issue the `cobbler` command with the `import` option. The `--mirror` option specifies the source of the distribution you want to package—in our case, we have used our DVD drive, `/media/cdrom`, and have assumed our distribution CD or DVD is mounted on this drive.

You can also specify an online repository, for example:

```
$ sudo  cobbler import --mirror=rsync://ftp.iinet.net.au/pub/fedora/linux/releases/➥
10/Fedora/i386/ --name=Fedora10 --arch=i386
```

Here we've specified a Fedora 10 build available via `rsync` (a type of simple file transfer). Cobbler will download the required files and create a distribution and a profile for our Fedora 10 build.

■ **Tip** You will need sufficient disk space on your host to copy whatever distributions you want to keep. For example, for the RHEL 5 build, you will need about 3GB of space.

Cobbler will run the import process and then return you to the prompt. Depending on the performance of your host (and, if you are importing over the network, the speed of your connection), this may take some time.

The last two options in our import, `--name` and `--arch`, are the name of the profile we want to create (e.g., RHEL5) and the architecture (e.g., i386, x86_64, etc.) of the distribution being imported. Cobbler can usually detect this, but it is safer to specify it. The architecture will then be suffixed to the name of the profile you are creating (e.g., `RHEL5-i386`).

After you've created your distribution and profile, you can see it in Cobbler using the `report` option, as shown in Listing 19-3.

Listing 19-3. *A Cobbler Report*

```
$ sudo cobbler report
distro                  : RHEL5-i386
architecture            : i386
breed                   : redhat
created                 : Tue Feb 24 21:46:33 2009
```

```
comment              : rhel5.2
initrd               : /var/www/cobbler/ks_mirror/RHEL5/images/pxeboot/initrd.img
kernel               : /var/www/cobbler/ks_mirror/RHEL5/images/pxeboot/vmlinuz
kernel options       : {}
ks metadata          : {'tree': 'http://@@http_server@@/cblr/links/RHEL5-i386'}
tree build time      : Thu May  1 09:23:47 2008
modified             : Tue Feb 24 21:52:03 2009
mgmt classes         : []
os version           : rhel5
owners               : ['admin']
post kernel options  : {}
redhat mgmt key      : <<inherit>>
template files       : {}
profile              : RHEL5-i386
distro               : RHEL5-i386
comment              :
created              : Tue Feb 24 21:46:33 2009
dhcp tag             : default
enable menu          : True
kernel options       : {}
kickstart            : /var/lib/cobbler/kickstarts/sample.ks
ks metadata          : {}
mgmt classes         : []
modified             : Tue Feb 24 21:46:33 2009
name servers         : []
owners               : ['admin']
post kernel options  : {}
redhat mgmt key      : <<inherit>>
repos                : []
server               : <<inherit>>
template_files       : {}
virt bridge          : xenbr0
virt cpus            : 1
virt file size       : 5
virt path            :
virt ram             : 512
virt type            : qemu
```

This option displays all the distributions and their profiles currently imported into Cobbler.

■ **Note** You may see more than one distribution and profile created from importing a distribution. For example, importing RHEL 5 will add the vanilla RHEL 5 distribution and the Xen (which is a type of virtualization we discuss in Chapter 20) version of RHEL 5.

Listing 19-3 shows our vanilla RHEL 5 distribution and the profile we created, `RHEL5-i386`. Most of the information in Listing 19-3 isn't overly important to us, but we do need to make note of the `kickstart` option, which has a value of `/var/lib/cobbler/kickstarts/sample.ks`. This Kickstart file will automate any build of a distribution; we'll look at it in more detail later in this chapter.

You can change what Kickstart file this profile uses (or edit other profile values) by editing your profile using the `cobbler profile` command. You can also list all the profiles by using the `cobbler profile list` command.

```
$ sudo cobbler profile edit --name=RHEL5-i386 --kickstart=/var/lib/cobbler/➥
kickstarts/custom.ks
```

Here we've edited the `RHEL5-i386` profile to use the Kickstart file `/var/lib/cobbler/kickstarts/custom.ks`.

You can also remove a profile using the `remove` command or copy one using the `copy` command.

```
$ sudo cobbler profile copy --name=RHEL5-i386 --newname=RHEL5-i386-new
$ sudo cobbler profile remove --name=RHEL5-i386-new
```

The first command will copy the `RHEL5-386` profile to `RHEL5-i386-new`, and the second command will delete the `RHEL5-i386-new` profile and its files.

■ **Note** You can see the other options you can edit on your profile by looking at the `cobbler` command's `man` page.

Building a Host with Cobbler

Now that you've added a profile and a distribution, you can boot a host and install your distribution. Choose a host (or virtual machine) you wish to build and reboot it. Your host may automatically search for a boot device on your network, but more likely you will need to adjust its BIOS settings to adjust the boot order. In order to boot from Cobbler, you need to specify that your host boots from the network first.

When your host boots, it will request an IP address from the network and get an answer from your DHCP server, as you can see in Figure 19-1.

```
CLIENT MAC ADDR: 00 0C 29 3B 22 46  GUID: 564DDFDB-733C-708F-9D49-FA93213B2246
CLIENT IP: 192.0.2.162  MASK: 255.255.255.0  DHCP IP: 192.0.2.156
GATEWAY IP: 192.0.2.1

PXELINUX 3.11 2005-09-02  Copyright (C) 1994-2005 H. Peter Anvin
UNDI data segment at:   0009C7F0
UNDI data segment size: 24D0
UNDI code segment at:   0009ECC0
UNDI code segment size: 0A0D
PXE entry point found (we hope) at 9ECC:0106
My IP address seems to be C00002A2 192.0.2.162
ip=192.0.2.162:192.0.2.156:192.0.2.1:255.255.255.0
TFTP prefix: /
Trying to load: pxelinux.cfg/01-00-0c-29-3b-22-46
Trying to load: pxelinux.cfg/C00002A2
Trying to load: pxelinux.cfg/C00002A
Trying to load: pxelinux.cfg/C00002
Trying to load: pxelinux.cfg/C0000
Trying to load: pxelinux.cfg/C000
Trying to load: pxelinux.cfg/C00
Trying to load: pxelinux.cfg/C0
Trying to load: pxelinux.cfg/C
Trying to load: pxelinux.cfg/default
boot: _
```

Figure 19-1. *Network boot*

Your host will boot to a command line appropriately called `boot:`. From here, you can launch the Cobbler menu by typing **menu**. You can see an example of this menu in Figure 19-2.

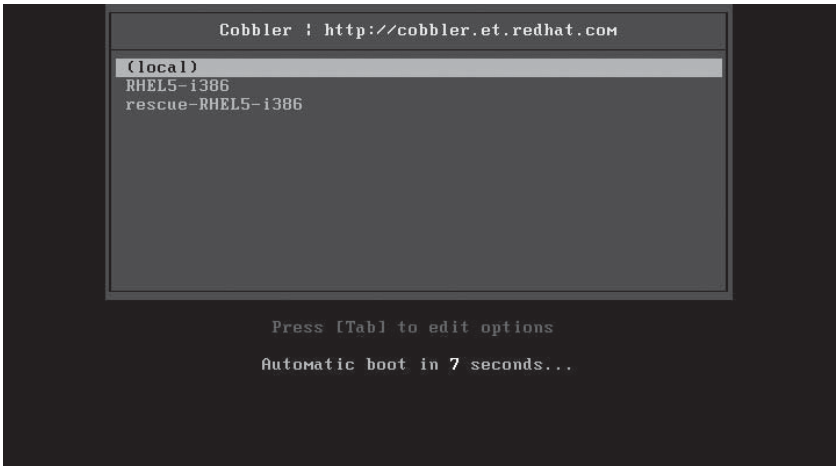

Figure 19-2. *The Cobbler menu*

From this menu, you can select the profile you'd like to install (e.g., `RHEL5-i386`). If you don't select a profile to be installed, Cobbler will automatically launch the first item on the menu, (`local`), which continues the boot process on the local host.

■ **Note** If you don't have an operating system installed on this host, this boot process will obviously fail.

If you've selected a profile, then this profile will start the installation process using the instructions contained in the associated Kickstart file. If you are watching your installation process, you will see the installation screens progress—all without requiring input from you to continue or select options.

Using Cobbler, you can also specify configuration options for particular hosts. You don't need to do this, but it is useful if you have a specific role in mind for a host and want to specify a particular profile or Kickstart configuration. To do this, you add hosts to Cobbler, identifying them via their MAC or IP addresses, using the `system` command.

```
$ sudo cobbler system add --name=gateway.example.com --mac=00:0C:29:3B:22:46➥
--profile=RHEL5-i386 --kickstart=gateway.ks
```

Here we've added a system named gateway.example.com with the specified MAC address.

■ **Note** You can usually see your MAC address during the network boot process, or you can often find it printed on a label on your network card.

The new host uses the `RHEL5-i386` profile and a Kickstart file called `gateway.ks`. If a host with the appropriate MAC address connects to our Cobbler host, then Cobbler will use these configuration settings to provision the host.

You can list the configured hosts using the `list` and `report` options.

```
$ sudo cobbler system list
gateway.example.com
```

A full listing of the gateway.example.com system definition can be seen using the `report` option.

```
$ sudo cobbler system report –name=gateway.example.com
```

We can also delete a system using the `remove` command:

```
$ sudo cobbler system remove --name=gateway.example.com
```

■ **Note** You can read about additional Cobbler capabilities on the `cobbler` command's `man` page.

Cobbler Web Interface

Cobbler also has a simple web interface you can use to manage some of its options. It's pretty simple at this stage, and the command-line interface is much more fully featured, but it is available if you wish to implement it. You can find instructions at `https://fedorahosted.org/cobbler/wiki/CobblerWebInterface`.

Troubleshooting Cobbler

You can troubleshoot the network boot process by monitoring elements on your host, including your log files, and by using a network monitoring tool like the `tcpdump` or `tshark` command.

You can start by monitoring the output of the DHCP process by looking at the `/var/log/messages` log files. Cobbler also logs to the `/var/log/cobbler/cobbler.log` file and the files contained the `kickstep` and `syslog` directories also under `/var/log/cobbler`.

You can also monitor the network traffic passing between your booting host and the boot server. You can use a variety of network monitoring tools for this:

```
$ sudo tcpdump port tftp
```

Cobbler has a wiki page available that contains documentation at `https://fedorahosted.org/cobbler/wiki/UserDocs`. The documentation includes some useful tips for troubleshooting at `https://fedorahosted.org/cobbler/wiki/UserDocs#Troubleshooting`. The Cobbler community also has a mailing list at `https://fedorahosted.org/mailman/listinfo/cobbler` and an active IRC channel on Freenode at #cobbler.

Provisioning with Ubuntu

Like Red Hat, Ubuntu can automatically provision hosts, but it lacks a server solution like Cobbler. In order to achieve the same result on Ubuntu as on Red Hat, you will have to do a little more manual configuration of your Ubuntu host to network boot your hosts.

■ **Note** Ubuntu can automate the installation process using a tool called Preseed. Ubuntu, however, also supports Kickstart. We're going to show you how to use a combination of Kickstart and Preseed to automate your installation in the next section.

COBBLER ON UBUNTU

Cobbler is also supported on Ubuntu; however, there are currently no packages built for Ubuntu. If you want to use Cobbler for Ubuntu, you can build it from source using the instructions at `https://fedorahosted.org/cobbler/wiki/DownloadInstructions` or create a package using the instructions in Chapter 7.

Installing Packages

We're going to start by setting up a PXE boot server on Ubuntu and installing some packages. You will need to install a TFTP server (to transfer the files to your target host) and the `inetutils-inetd` package to run the TFTP server. If it is not already installed on your host, you will need to install a DHCP server using the `dhcp3-server` package. Lastly, to deliver the distribution's files to the host to be built, we'll use the Apache web server. If you don't have Apache installed, you'll need to install the `apache2` package.

```
$ sudo apt-get install inetutils-inetd tftpd-hpa dhcp3-server apache2
```

■ **Note** We show how to configure DHCP in Chapter 9.

Configuring the DHCP Server

Next, you need to configure your DHCP server. If you already have a DHCP server configured, we recommend updating the configuration to specify a PXE boot server for your existing DHCP ranges; otherwise you'll need to configure DHCP to provide addresses in an appropriate range to provision hosts. For example, you could create a DHCP range for provisioning hosts.

To configure our Ubuntu DHCP server, we update the `/etc/dhcp3/dhcpd.conf` configuration file. We add the required configuration to the example configuration we created in Chapter 9, as shown in Listing 19-4.

Listing 19-4. *Ubuntu Example DHCP Server Configuration*

```
allow booting;
allow bootp;
ddns-update-style interim;
ddns-ttl 3600;
default-lease-time 600;
max-lease-time 7200;
log-facility local7;

key dynamic-update-key {
    algorithm hmac-md5;
    secret "3PDRnypPtzJqpbQvbw/B7bhPuHqpUeOSdi95Z4Ez/IzhS61dzcK6MJ6CdFHkkegp➥
TN1kmXOM6GggRNE24aPmOw==";
}

zone 0.168.192.in-addr.arpa. {
    key dynamic-update-key;
    primary 192.168.0.1;
}

zone example.com. {
    key dynamic-update-key;
    primary 192.168.0.1;
}

subnet 192.168.0.0 netmask 255.255.255.0 {
    filename "pxelinux.0";
    next-server 192.168.0.1;
    option routers 192.168.0.254;
    option domain-name "example.com";
    option domain-name-servers 192.168.0.1;
    option broadcast-address 192.168.0.255;
    group "static" {
```

```
        use-host-decl-names on;
        host au-mel-rhel-1 {
            hardware ethernet 00:16:3E:15:3C:C2;
            fixed-address au-mel-rhel-1.example.com;
        }
    }
    pool {
        range 192.168.0.101 192.168.0.150;
        deny unknown clients;
    }
    pool {
        range 192.168.0.151 192.168.0.200;
        allow unknown clients;
        default-lease-time 7200;
        max-lease-time 21600;
    }
}
```

Here we add the `allow` directive for `bootp` and `booting`, which tells the DHCP server to accept network boot requests at the top of our file. In the `subnet` directive, we add two options, `filename` and `next-server`.

The `filename` option specifies the name of the file that the DHCP server will deliver to the host that wishes to net boot. This file contains the initial instructions to boot the host, and you should specify `pxelinux.0` here. We'll install this file shortly.

The `next-server` option tells our net-booting host the server from which to retrieve the boot files. You should specify the IP address of the PXE boot server.

■ **Tip** After changing your DHCP configuration, you will need to restart the DHCP service.

Configure the TFTP Server

We installed the `tftp-hpa` package that contains the TFTP server. We now need to configure this server so it can transfer our boot files to the target host. To do this, we edit the `/etc/default/tftp-hpa` file:

```
#Defaults for tftpd-hpa
RUN_DAEMON="yes"
OPTIONS="-l -s /var/lib/tftpboot"
```

In the preceding code, we enable the `RUN_DAEMON` option by setting it to `yes`. Notice the `OPTIONS` line that specifies the `-l` and `-s` options. The `-l` option runs the daemon in standalone listen mode. The `tftpd` service usually runs under `inetd`, which is a type of daemon manager; the `-l` option tells it to run like a normal daemon. The `-s` option tells the daemon the location of our boot files, here `/var/lib/tftpboot`. This is where we'll store our boot files.

Once we've configured the `tftpd-hpa` package, we need to restart the `tftpd-hpa` service.

```
$ invoke-rc.d tftpd-hpa restart
```

Installing the Boot Files

You now need to install the files required to boot your host. These files will be installed into the `/var/lib/tftpboot` directory. Ubuntu comes with a collection of boot and kernel files specifically designed for network booting. These are available in the `install/netboot` directory on your Ubuntu media:

```
$ sudo cp -r /media/cdrom/install/netboot/* /var/lib/tftpboot/
```

or you can download them from the Ubuntu online repositories.

```
$ cd /tmp
$ lftp -c "open http://archive.ubuntu.com/ubuntu/dists/hardy/main/installer-i386/➥
current/images/; mirror netboot/"
$ mv netboot/* /var/lib/tftpboot
$ rm -fr netboot
```

Here we used the `lftp` command to copy the contents of the `netboot` directory for the Ubuntu 8.04 or Hardy release to our `/tmp` directory. We then moved the contents of this directory to the `/var/lib/tftpboot` directory.

■ **Note** The `lftp` command is a more sophisticated version of the FTP file transfer command. To see how it works, refer to its `man` page.

Configuring the PXE Boot Loader

Now that you have the network boot files, you can configure the boot loader for your environment. Let's look at the contents of the `/var/lib/tftpboot` directory.

```
boot.img.gz
mini.iso
netboot.tar.gz
pxelinux.0 -> ubuntu-installer/i386/pxelinux.0
pxelinux.cfg -> ubuntu-installer/i386/pxelinux.cfg
ubuntu-installer
```

The directory contains five files (two of which are symlinked) and a directory called `ubuntu-installer`. The `pxelinux.0` (which we configured in our DHCP server's `filename` option), `boot.img.gz`, and `netboot.tar.gz` files will do the initial boot of our host. The `mini.iso` file is an ISO (burnable to CD) image of the boot files. If you don't use network-based provisioning, you can burn this image to a CD and use it to boot and perform a minimal installation. The `pxelinux.cfg` directory contains the configuration files used by PXE to select what boot image to load (we'll adjust this for our environment shortly), and the `ubuntu-installer` directory contains files required for our PXE boot.

If you look in the pxelinux.cfg directory, you'll find a single file called default. This is the default configuration file used when booting hosts. Let's look at its contents:

```
DISPLAY ubuntu-installer/i386/boot-screens/boot.txt

F1 ubuntu-installer/i386/boot-screens/f1.txt
F2 ubuntu-installer/i386/boot-screens/f2.txt
F3 ubuntu-installer/i386/boot-screens/f3.txt
F4 ubuntu-installer/i386/boot-screens/f4.txt
F5 ubuntu-installer/i386/boot-screens/f5.txt
F6 ubuntu-installer/i386/boot-screens/f6.txt
F7 ubuntu-installer/i386/boot-screens/f7.txt
F8 ubuntu-installer/i386/boot-screens/f8.txt
F9 ubuntu-installer/i386/boot-screens/f9.txt
F0 ubuntu-installer/i386/boot-screens/f10.txt

DEFAULT install

LABEL install
   kernel ubuntu-installer/i386/linux
   append ks=http://192.0.2.161/ks.cfg vga=normal initrd=ubuntu-installer/i386/➥
initrd.gz --
LABEL linux
   kernel ubuntu-installer/i386/linux
   append vga=normal initrd=ubuntu-installer/i386/initrd.gz --
LABEL cli
   kernel ubuntu-installer/i386/linux
   append tasks=standard pkgsel/language-pack-patterns= pkgsel/install-language-➥
support=false vga=normal initrd=ubuntu-installer/i386/initrd.gz --

LABEL expert
   kernel ubuntu-installer/i386/linux
   append priority=low vga=normal initrd=ubuntu-installer/i386/initrd.gz --
LABEL cli-expert
   kernel ubuntu-installer/i386/linux
   append tasks=standard pkgsel/language-pack-patterns= pkgsel/install-language-➥
support=false priority=low vga=normal initrd=ubuntu-installer/i386/initrd.gz --

LABEL rescue
   kernel ubuntu-installer/i386/linux
   append vga=normal initrd=ubuntu-installer/i386/initrd.gz rescue/enable=true --

PROMPT 0
TIMEOUT 0
```

The file specifies our boot environment, first by specifying the text that will be displayed when our host boots, via the DISPLAY option, and when each function key is pressed (the F1 to F0 options). It then uses the LABEL option to specify the various types of boots that we can perform. The DEFAULT option specifies the default boot label, in our case install.

Each label specifies the kernel to load and the options to pass to that kernel.

■ **Tip** You can find documentation on the boot options at `https://help.ubuntu.com/community/BootOptions`.

Lastly, the `PROMPT` and `TIMEOUT` options control how our boot sequence will operate. The `PROMPT` option, if set to `1`, will wait at the boot prompt (see Figure 19-3) until a boot label is selected.

Figure 19-3. *The Ubuntu boot prompt*

If the `PROMPT` option is set to `0`, then the default boot option will be immediately taken without waiting for a keypress. The `TIMEOUT` specifies how long in seconds it will wait at the boot prompt for that selection. If no option is selected before the timeout expires, then the default label will be booted. If you want your host to automatically boot and install, then you should set the `PROMPT` value to `0`.

Rather than using the default configuration option, you can specify individual files in this directory for the specific host you wish to boot and build. To do this, create a file name using the target host's MAC address. For example, if your host's MAC address is 00:0A:E4:2E:A6:42, you could create a file called `000AE42EA642` by copying the default file and editing it. When the host with this MAC address requests a network boot, this configuration file will be used rather than the `default` file.

Configuring Apache for Provisioning

Next, you need to add the contents of the distribution you'd like to install to a directory and enable access via the Apache web server. After the boot process, these distribution files will be used to install your new host.

To do this, copy the files from the Ubuntu media by creating a directory called `ubuntu` under the `/var/www` directory and copying the contents of the Ubuntu installation media to it.

```
$ sudo mkdir /var/www/ubuntu
$ sudo cp -r /media/cdrom/* /var/www/ubuntu/
```

You can achieve the same result by directly mounting the CD or DVD. Using the `mount` command, use the `--bind` option to make the mounted CD or DVD available in the `/var/www/ubuntu` directory. This means you don't need to install the files onto your PXE boot host. The following command mounts whatever CD or DVD is mounted at `/media/cdrom` again at `/var/www/ubuntu`.

```
$ sudo mount --bind /media/cdrom/ /var/www/ubuntu/
```

The Apache web server should then serve these files out via HTTP. In our Kickstart configuration file, we'll tell the host how to find these files when it wants to install our distribution.

■ **Note** We talk about how to configure and run Apache in Chapter 11.

Firewall Configuration

You also need to ensure your firewall is configured to allow the required access for network booting. To do this, you need to have port 69 open for the TFTP server and port 80 open for HTTP traffic. The following are some appropriate `iptables` rules:

```
-A Firewall-eth0-INPUT –s 192.168.0.0/24 -p udp -m state --state NEW --dport 69 ➥
-j ACCEPT
-A Firewall-eth0-INPUT -p tcp -m state --state NEW --dport 80 -j ACCEPT
```

These rules allow access for any host on the 192.168.0.0/24 subnet to the boot server on the appropriate ports. You can read more about firewall rules in Chapter 6.

Specifying the Kickstart File

To finish this configuration, you'll create and specify a Kickstart file to automate the actual installation process. You'll store this file in `/var/www/ubuntu` and make it available via the Apache web server, just like your distribution's installation files. Let's create a file now:

```
$ sudo touch /var/www/ubuntu/ks.cfg
```

We created a file called `ks.cfg` in the `/var/www/ubuntu` directory.

■ **Note** If you mounted a DVD directory (as described previously), storing the `ks.cfg` file in the `/var/www/ubuntu` directory won't work, and you'll need to locate the file elsewhere (e.g., by creating another directory and placing it there).

Let's now edit the `ks.cfg` file to include some very basic configuration.

```
install
url --url http://192.168.0.1/ubuntu/
```

Our example file has two Kickstart options, `install` and `url`. The `install` option tells Kickstart to install rather than the alternative, `upgrade`, which upgrades an existing installation. The `url` option tells Kickstart where to find the files required to install the distribution; these are the files we are serving out via our Apache web server.

■ **Note** When you boot a host with this example file, the installation starts, but since you haven't specified any answers to installation questions, you still need to answer each question. You'll add to this file in the next section and answer the required questions.

To use our `ks.cfg` configuration file, we need to tell our boot server where to find it. We do this by adding an additional option to one or more labels in our PXE configuration file, for example, `/var/lib/tftpboot/pxelinux.cfg/default`. Let's look at the install label from the default file:

```
LABEL install
   kernel ubuntu-installer/i386/linux
   append ks=http://192.168.0.1/ubuntu/ks.cfg vga=normal➥
initrd=ubuntu-installer/i386/initrd.gz --
```

You can see we've added the `ks` option to our kernel boot options. The `ks` option tells the boot server where to find our Kickstart file, in our case via HTTP at `http://192.168.0.1/ubuntu/ks.cfg`.

Network Booting an Ubuntu Host

You have set up your Ubuntu boot server and specified your distribution's installation files, and now you can boot and install hosts. To do this, you'll configure your host to boot from the network, usually using the appropriate BIOS setting.

The booting host will attempt to acquire an IP address from your DHCP server. If it gets an IP address, then it will request boot instructions. The DHCP server will provide the `pxelinux.0` file and direct the booting host to the PXE server. The PXE server will provide the appropriate boot files and display the Ubuntu boot screen, as shown in Figure 19-4.

Figure 19-4. *The Ubuntu boot screen*

After bootup, the Kickstart configuration file will be retrieved and the installation process will be initiated. We'll look at automating that process in the next section.

Troubleshooting Ubuntu Network Booting

Troubleshooting the network boot process requires monitoring several elements on your host, including your log files, and using a network monitoring tool like the `tcpdump` or `tshark` command.

Let's start by monitoring the output of the DHCP process by looking at `/var/log/daemon.log`. You can also monitor the network traffic passing between your booting host and the boot server using a variety of network monitoring tools:

```
$ sudo tshark port tftp
Capturing on eth0
  0.000000  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: pxelinux.0\000, ➥
Transfer type: octet\000
  0.126924  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: pxelinux.0\000, ➥
Transfer type: octet\000
  0.238396  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/564ddfdb-733c-708f-9d49-fa93213b2246\000, Transfer type: octet\000
  0.242177  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/01-00-0c-29-3b-22-46\000, Transfer type: octet\000
  0.244886  192.168.0.1 -> 192.168.0.161  TFTP Read Request, ➥
File: pxelinux.cfg/C00002A4\000, Transfer type: octet\000
  0.246654  192.168.0.1 -> 192.168.0.161  TFTP Read Request, ➥
File: pxelinux.cfg/C00002A\000, Transfer type: octet\000
```

```
  0.248279  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/C00002\000, Transfer type: octet\000
  0.289393  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
 pxelinux.cfg/C0000\000, Transfer type: octet\000
  0.306368  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/C000\000, Transfer type: octet\000
  0.311124  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/C00\000, Transfer type: octet\000
  0.315184  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/C0\000, Transfer type: octet\000
  0.318966  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/C\000, Transfer type: octet\000
  0.333135  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
pxelinux.cfg/default\000, Transfer type: octet\000
  0.355684  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
ubuntu-installer/i386/boot-screens/boot.txt\000, Transfer type: octet\000
  0.361184  192.168.0.1 -> 192.168.0.161  TFTP Read Request, File: ➥
ubuntu-installer/i386/boot-screens/splash.rle\000, Transfer type: octet\000
```

Here we used the `tshark` command (installed via the `tshark` package) to monitor all traffic on port 69 (the TFTP port). You can see the request for the `pxelinux.0` file and our `pxelinux.cfg/default` configuration file, and finally the display of the Ubuntu boot screen.

■ **Note** You can find more information about network booting at `https://help.ubuntu.com/8.04/installation-guide/i386/install-methods.html`.

Kickstart and Preseed

On Red Hat, the language used to automatically install your host is called Kickstart. On Ubuntu, it is called Preseed. Kickstart, however, is also supported on Ubuntu in a form called Kickseed. The current Kickseed support can't configure Ubuntu completely, but helpfully it can also use selected Preseed directives to address any gaps. For simplicity's sake and because it's an easier language to use, we're going to show you how to use Kickstart to automate your installation for both Red Hat and Ubuntu. Where something isn't supported on Ubuntu, we'll show you how to use Preseed to configure it.

■ **Note** Kickstart support for Ubuntu is growing regularly. The 8.10 and the Jaunty release (i.e., the future 9.04 release) will enhance this support further.

A Kickstart configuration file contains the instructions required to automate the installation process. It's a simple scripted process for most installation options, but it can be extended to do some complex configuration. Kickstart is heavily used on Red Hat, and more recently on

Ubuntu, so it's well documented. You can find detailed documentation for Kickstart on RHEL 5 at `https://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Installation_Guide/ch-kickstart2.html`. This manual is also useful for provisioning Ubuntu hosts, as it uses many of the same directives. Again, when a particular directive isn't supported in Ubuntu, we'll provide you with the Preseed equivalent.

You can find documentation on Preseed and its directives at `https://help.ubuntu.com/8.04/installation-guide/i386/appendix-preseed.html`. We'll work with a few of these directives later in this section.

You've already seen how to specify Kickstart files to your provisioning environments, using both Cobbler on Red Hat and the PXE boot configuration on Ubuntu. Let's start by looking at some of the contents of a simple Kickstart file in Listing 19-5.

Listing 19-5. *A Kickstart File*

```
install
# System authorization information
auth  --useshadow  --enablemd5
# System bootloader configuration
bootloader --location=mbr
# Partition clearing information
clearpart --all --initlabel
# Use text mode install
text
```

Listing 19-5 shows a list of configuration directives starting with the `install` option, which dictates the behavior of the installation process by performing an installation. The alternative is `upgrade`, which automates an upgrade of a host.

You can then see configuration directives with options, for example, `auth --useshadow --enablemd5`, which tell Kickstart how to answer particular installation questions. The `auth` statement has the values `--useshadow` and `--enablemd5` here, which enable shadow passwords and use of MD5, respectively.

The option that follows, `bootloader` with a value of `--location=mbr`, tells Kickstart to install the boot loader into the MBR. Next is the directive `clearpart`, which clears all partitions on the host and creates default labels for them. The final option, `text`, specifies we should use text-based installation as opposed to the GUI.

■ **Tip** You can use Kickstart to upgrade hosts as well as install them. If you have an existing host, you can network boot from a new version of your operating system and use a Kickstart file to script and upgrade.

There are too many directives to discuss them individually, so we show you in Table 19-1 the directives that must be specified and some of the other major directives that you may find useful.

Table 19-1. *Required Kickstart Directives*

Directive	Description
`auth`	Configures authentication.
`bootloader`	Configures the boot loader.
`keyboard`	Configures the keyboard type.
`lang`	Configures the language on the host.
`part`	Configures partitions. This is required for installation, but not if upgrading.
`rootpw`	Specifies the password of the root user.
`timezone`	Specifies the time zone the host is in.

You can also find a useful list of the available directives with explanations at `http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Installation_Guide-en-US/s1-kickstart2-options.html`.

■ **Tip** If you are on Red Hat, you can see an example Kickstart file that was created when you installed your host in the `/root/anaconda-ks.cfg` file. This will show you how your current host is built and can be used as an example to build similar hosts.

Installation Source

You've already seen the `install` and `upgrade` directives that specify the behavior of the installation. You can also specify the source of your installation files. In the Ubuntu provisioning section, you saw the `url` directive, which tells Kickstart to look for its installation files at an HTTP URL (e.g., `http://192.168.0.1/ubuntu`).

```
url --url http://192.168.0.1/ubuntu/
```

For Cobbler, we define a variable to specify the location of our installation source.

```
url --url=$tree
```

The `url` directive can also be used to specify an FTP server:

```
url --url ftp://jsmith:passsword@192.168.0.1/ubuntu
```

We can specify some alternative sources, including `cdrom`, when installing from a locally mounted CD or DVD and hard drive to install from a local partition.

```
harddrive --dir=/ubuntu --partition=/installsource
```

Keyboard, Language, and Time Zone

The next snippet we're going to show you configures our keyboard, language, and time zone.

```
# System keyboard
keyboard us
# System language
lang en_AU
# System timezone
timezone  Australia/Melbourne
```

Here we've specified `us` as the value for the `keyboard` directive to indicate a US keyboard. We've specified our language as `en_AU` (English Australian) and our time zone as `Australia/Melbourne`.

■ **Tip** The keyboard, language, and time zone options are identical on Red Hat and Ubuntu.

Managing Users

You can also set the `root` user's password with the Kickstart `rootpw` directive.

```
rootpw --iscrypted $1$V.rhw$VUj.euMxoV9WkcQSanpGi0
```

The `rootpw` directive is a required Kickstart option for all Kickstart files. It can take either a plain-text value or an encrypted value for the `root` user's password when the `--iscrtypted` option is specified. You can create this encrypted password using the `grub-md5-crypt` command like so:

```
$ grub-md5-crypt
Password:
Retype password:
$1$V.rhw$VUj.euMxoV9WkcQSanpGi0
```

Specify the password you'd like to be encrypted and then retype it when prompted. You should then cut and paste the encrypted password into the Kickstart file.

On Ubuntu, the `rootpw` directive defaults to the `--disabled` option, in keeping with Ubuntu's approach of disabling the `root` user.

```
rootpw --disabled
```

■ **Note** The `--disabled` option is not available on Red Hat hosts.

On Ubuntu, Kickstart can also create a new user with the `user` directive.

```
user jsmith --fullname "John Smith" --password password
```

The preceding code creates a new user called `jsmith`, with a full name of `John Smith` and a password of *password*. By adding the `--iscrypted` option, you can add a user with an encrypted password. We would create our encrypted password as we did with the `rootpw` directive.

Firewall and Network

On Red Hat, you can configure your host's initial firewall and network configuration.

```
# Firewall configuration
firewall --enabled --http --ssh --smtp
# SELinux configuration
selinux --disabled
```

Here we enabled the firewall with the `firewall` option and allowed access via HTTP, SSH, and SMTP. (You can disable the firewall with the `--disabled` option.) We also disabled SELinux using the `selinux --disabled` option.

Ubuntu cannot change firewall configuration through either Kickstart or Preseed, so you should just set this value as follows:

```
firewall --disabled
```

On both Red Hat and Ubuntu, you can configure your network connections with Kickstart like so:

```
# Network information
network --bootproto=static --device=eth0 --gateway=192.168.0.254 ➥
--ip=192.168.0.1 --nameserver=192.168.0.1 --netmask=255.255.255.0 --onboot=on
```

You can also specify network configuration for one or more interfaces using the `network` option. You can see we've set the various options required to configure the eth0 interface. You can also specify DHCP, for example:

```
network --bootproto=dhcp --device=eth0 --onboot=on
```

On Red Hat with Cobbler, if you're working with a specific host (one created with the `cobbler system` command), you can pass specific network configuration values to the Cobbler system configuration.

```
$ sudo cobbler system edit --name=gateway.example.com --mac=00:0C:29:3B:22:46➥
--profile=RHEL5 --interface=0 --ip=192.168.0.1 --subnet=255.255.255.0 --➥
gateway=192.168.0.254 --hostname=gateway --static=1
```

Here we've specified the `edit` command to change an existing Cobbler-defined system and passed network configuration values to our system. This would define a static network configuration for interface eth0. We specify that the configuration is static using the `--static=1` option; we would specify `--static=0` for a DHCP configuration. The interface to be configured is specified using the `--interface=0` option.

Then, instead of specifying a `network` line, in our Kickstart file we specify what Cobbler calls a *snippet*.

```
$SNIPPET('network_config')
```

When building your host, Cobbler passes the network configuration you've specified to this snippet and a template it contains. This is then converted into the appropriate `network` line and your host is configured.

■ **Tip** This snippet is a simple use of Cobbler's snippet system. You can define a variety of other actions using snippets, and you can see a selection of these in the `/var/lib/cobbler/snippets` directory, including the `network_config` snippet we used in this section. You can see how to use these snippets in the `sample.ks` file, and you can find instructions on how to make use of templates and snippets at `https://fedorahosted.org/cobbler/wiki/KickstartTemplating` and `https://fedorahosted.org/cobbler/wiki/KickstartSnippets`.

Disks and Partitions

You've already seen one option Kickstart uses to configure disks and partitions, `clearpart`, which clears the partitions on the host. You can then use the `part` option to configure partitions on the host like so:

```
# Partition clearing information
clearpart --all --initlabel
part /boot --asprimary --bytes-per-inode=4096 --fstype="ext3" --size=150
part / --asprimary --bytes-per-inode=4096 --fstype="ext3" --size=4000
part swap --bytes-per-inode=4096 --fstype="swap" --size=512
```

■ **Note** On Red Hat, you can create a similar configuration just by specifying the `autopart` option. The `autopart` option automatically creates three partitions. The first partition is a 1GB or larger root (`/`) partition, the second is a swap partition, and the third is an appropriate boot partition for the architecture. One or more of the default partition sizes can be redefined with the `part` directive.

You use the `part` option to create specific partitions. In the preceding code, we first created two partitions, `/boot` and `/`, both ext3. We specified a size of 150MB for the `/boot` partition and a size of 4000MB (or 4GB) for the `/` or root partition. We also created a swap partition with a size of 512MB.

Using Kickstart on Red Hat, we can create software RAID configurations, for example:

```
part raid.01 --asprimary --bytes-per-inode=4096 --fstype="raid" --grow --ondisk=sda ➥
--size=1
part raid.02 --asprimary --bytes-per-inode=4096 --fstype="raid" --grow --ondisk=sdb ➥
--size=1
part raid.03 --asprimary --bytes-per-inode=4096 --fstype="raid" --grow --ondisk=sdc ➥
--size=1
part raid.04 --asprimary --bytes-per-inode=4096 --fstype="raid" --grow --ondisk=sdd ➥
--size=1
part raid.05 --asprimary --bytes-per-inode=4096 --fstype="raid" --grow --ondisk=sde ➥
--size=1
raid / --bytes-per-inode=4096 --device=md0 --fstype="ext3" --level=5 raid.01 raid.02 ➥
raid.03 raid.04 raid.05
```

We specified five RAID disks, and each disk uses its entire contents as indicated by the `--grow` option. The respective disk to be used is specified with the `--ondisk` option, here ranging from `sda` to `sde`. Lastly, we used the `raid` option to specify the `md0` RAID disk as the `/` or root partition.

■ **Caution** On Ubuntu, Kickstart doesn't support RAID, although Preseed does partially. In version 8.04 of Ubuntu Preseed, RAID support is largely experimental and we don't recommend you provision and configure hosts with RAID configurations using it.

You can also create partitions using LVM during an automated installation. On Red Hat, for example, you would create them like so:

```
part /boot --fstype ext3 --size=150
part swap --size=1024
part pv1 --size=1 --grow
volgroup vg_root pv1
logvol  /  --vgname=vg_root  --size=81920 --name=lv_root
```

In the preceding sample, we created a 150MB boot partition, a 1GB swap partition, and a physical volume called pv1 on the remainder of the disk, using the `--grow` option to fill the rest of the disk. We then created an 80GB LVM logical volume called `vg_root`.

On Ubuntu, you can use the `preseed` directive to use Preseed to configure LVM inside your Kickstart file.

```
preseed --owner d-i partman-auto/method string lvm
preseed --owner d-i partman-lvm/device_remove_lvm boolean true
preseed --owner d-i partman-lvm/confirm boolean true
preseed --owner d-i partman-auto/choose_recipe select atomic
preseed --owner d-i partman/confirm_write_new_label boolean true
preseed --owner d-i partman/choose_partition select finish
preseed --owner d-i partman/confirm boolean true
```

The `preseed` directive is very simple and is a way of including Preseed directives in your Kickstart configuration. The directive is structured as follows:

```
preseed --owner owner key/sub-key value
```

The first part, `preseed`, tells Kickstart you are using a Preseed directive. The `--owner` option tells Preseed which application or function the directive belongs to—for example, `d-i` indicates that the Preseed directive belongs to the Debian installer.

■ **Note** If you don't specify an `--owner` option, Preseed defaults to a value of `d-i`.

The key/subkey values specify the Preseed option to set; these values are structured as a collection of keys with a series of subkeys beneath each key. For example, the `partman` key contains all the Preseed options for configuring partitions. Here, underneath this key, you can see the `confirm` and `confirm_write_new_label` subkeys.

```
partman/confirm
partman/confirm_write_new_label
```

The `type` field contains the type of the value, for example, a string or a Boolean entry. The `value` field contains the actual value of the setting.

Let's apply Kickstart's `preseed` directive to our LVM configuration. We first select the LVM partitioning method:

```
preseed --owner d-i partman-auto/method string lvm
```

Then we remove any existing LVM devices using the `device_remove_lvm` directive and skip the confirmation message using the `confirm` directive.

```
preseed --owner d-i partman-lvm/device_remove_lvm boolean true
preseed --owner d-i partman-lvm/confirm boolean true
```

We then choose our partitioning method, in this case `atomic`, using the `partman/choose_recipe` key.

```
preseed --owner d-i partman-auto/choose_recipe select atomic
```

The `atomic` method creates all files in one partition, the recommended default for Ubuntu installations. Instead of `atomic`, we could specify `home` for a separate `/home` partition, and `multi` for separate `/home`, `/usr`, `/var`, and `/tmp` partitions.

We next tell Kickstart to automatically finish and write the LVM configuration.

```
preseed --owner d-i partman/choose_partition select finish
preseed --owner d-i partman/confirm boolean true
```

■ **Tip** Support for native Kickstart—not using the `preseed` directive—for LVM configuration will be available in Ubuntu release 9.04.

You can also customize the exact disk and partition configuration using recipes you can read about at `https://help.ubuntu.com/8.04/installation-guide/i386/preseed-contents.html`.

■ **Note** We discuss partitioning, software RAID, and LVM in Chapter 8.

Package Management

Using Kickstart, you can specify the packages you wish to install. On Red Hat, you specify a section starting with `%packages` and then the list of package groups or packages you wish to install.

```
%packages
@ Administration Tools
@ Server Configuration Tools
@ System Tools
@ Text-based Internet
dhcp
```

We specify an at symbol (@), a space, and then the name of the package group we wish to install, for example, Administration Tools. We can also specify individual packages by listing them by name without the @ symbol and space, as we have here with the `dhcp` package.

Ubuntu uses a similar setup:

```
%packages
@ kubuntu-desktop
dhcp-client
```

Here we've installed the Kubuntu-Desktop package group and the `dhcp-client` package.

■ **Note** We discuss package groups in Chapter 7.

Installation Behavior

You can configure some of the behavior of the installation process, for example:

```
# Run the Setup Agent on first boot
firstboot –disable
# Skip installation key
key --skip
# Reboot after installation
reboot
```

Both the `firstboot` and `key` directives are specific to Red Hat and are not relevant on Ubuntu. The `firstboot` directive specifies whether the postinstallation menus that normally run on your first boot are enabled. Use `--enable` to have the menus run or `--disable` to skip them. The `key` directive controls entering a Red Hat installation key; use the `--skip` option to skip the entry screen.

RED HAT INSTALLATION KEYS

Kickstart also supports specifying Red Hat installation keys in your build (as discussed in Chapter 2). To do so, add the following values to your Kickstart file.

```
# RHEL Install Key
key $getVar('rhel_key', '--skip')
```

You can then specify some Kickstart metadata using the Cobbler system command for each host like so:

```
$ sudo cobbler system edit --name=00:0C:29:3B:22:46 ➥
--ksmeta="rhel_key=4ea373203ae2bd77"
```

You replace the key, 4ea373203ae2bd77, with your installation key for that host. If you don't specify a key, Kickstart will use the `--skip` option to skip the application of an installation key.

The `reboot` directive tells Kickstart to reboot the host after installation. You can also specify the `shutdown` directive to tell the host to shut down after installation.

Pre- and Postinstallation

You can run scripts before and after Kickstart installs your host. The prerun scripts run after the Kickstart configuration file has been parsed, but before your host is configured. Any prerun script is specified at the end of the Kickstart file and prefixed with the line `%pre`.

The postrun scripts are triggered after your configuration is complete and your host is installed. They should also be specified at the end of the Kickstart file and prefixed by a `%post` line. This is the `%post` section from our `sample.ks` configuration file:

```
%post
$SNIPPET('post_install_kernel_options')
$SNIPPET('post_install_network_config')
$SNIPPET('redhat_register')
```

Here we've specified three postrun Cobbler snippets that configure kernel and network options and register the host with Red Hat (as we are using RHN).

This postrun scripting space is useful to run any required setup applications or scripts.

Kickstart Configurator

Also available to create Kickstart files is the GUI-based Kickstart Configurator (see `http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Installation_Guide-en-US/ch-redhat-config-kickstart.html`). You can install this application using the `system-config-kickstart` package on Red Hat:

```
$ sudo yum install system-config-kickstart
```

On Ubuntu, you would use the following:

```
$ sudo apt-get install system-config-kickstart
```

You can then launch Kickstart Configurator from the Applications ➤ System Tools ➤ Kickstart menu or via the following:

```
$ system-config-kickstart
```

Figure 19-5 shows the application's interface.

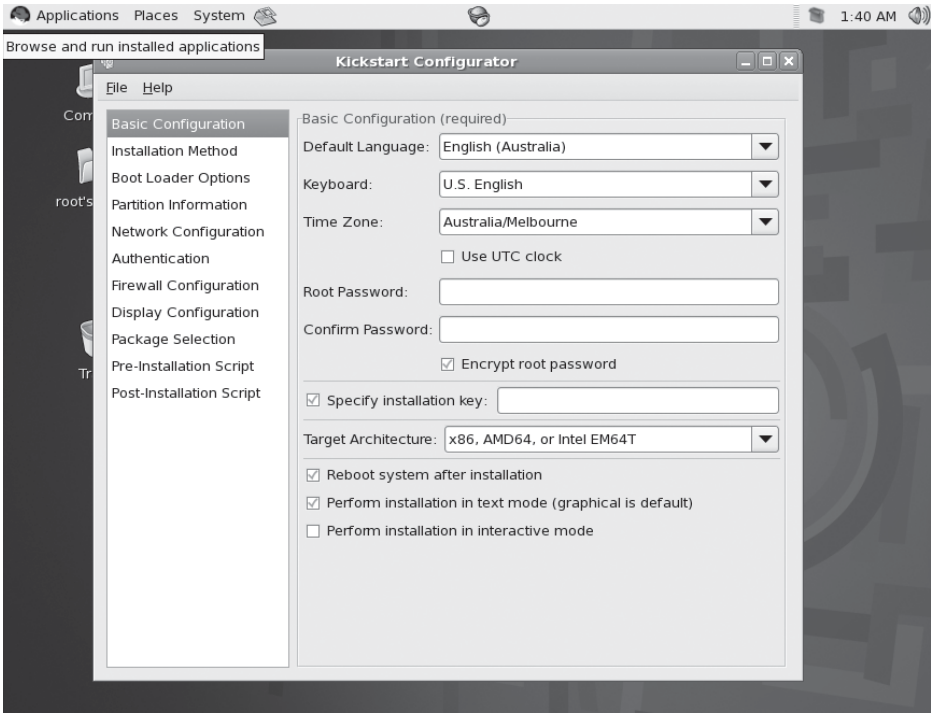

Figure 19-5. *The Kickstart Configurator interface*

Using Kickstart Configurator is often an easy way to quickly create simple Kickstart configurations if you don't wish to edit them on the command line.

Complete Kickstart Configurations

You've seen snippets of Kickstart (and some Preseed) configurations thus far. Let's now look at complete examples for both Red Hat and Ubuntu.

First up is a complete example for Red Hat:

```
install
reboot
url --url=$tree
key --skip
firstboot --disable
auth --enablemd5 --useshadow
bootloader --loader=mbr
keyboard us
```

```
lang en_AU
timezone  Australia/Melbourne
rootpw --iscrypted $1$V.rhw$VUj.euMxoV9WkcQSanpGi0
firewall --enabled --http --ssh --smtp
selinux --disabled
network --bootproto=dhcp --device=eth0 --onboot=on
clearpart --all –initlabel
autopart
%packages
@ Administration Tools
@ Server Configuration Tools
@ System Tools
@ Text-based Internet
dhcp
```

Here we created a very simple installation script. We're performing an installation and rebooting after the installation. We configured the required values like the keyboard, language, and time zone, and we created a password for the `root` user. We also enabled the firewall including access to the HTTP, SSH, and SMTP ports, and we requested a DHCP address for the eth0 interface. We cleared any existing partitions and then used the `autopart` directive to automatically partition the first disk on the host. Lastly, we installed a series of package groups as well as the `dhcp` package.

Here's a complete configuration on Ubuntu:

```
install
reboot
url --url http://192.0.2.161/ubuntu/
auth --useshadow --enablemd5
bootloader --location=mbr
keyboard us
lang en_AU
timezone  Australia/Melbourne
rootpw --disabled
user jsmith --fullname "John Smith" --password password
selinux --disabled
firewall --disabled
network --bootproto=dhcp --device=eth0 --onboot=on
preseed --owner d-i partman-auto/method string lvm
preseed --owner d-i partman-lvm/device_remove_lvm boolean true
preseed --owner d-i partman-lvm/confirm boolean true
preseed --owner d-i partman-auto/choose_recipe select atomic
preseed --owner d-i partman/confirm_write_new_label boolean true
preseed --owner d-i partman/choose_partition select finish
preseed --owner d-i partman/confirm boolean true
%packages
@ kubuntu-desktop
dhcp-client
```

Our Ubuntu configuration is very similar to our Red Hat configuration, but it's customized to suit the distribution—for example, we disabled the `root` user using the `rootpw` directive. You can also see where we used the `preseed` directive to directly specify some Preseed options to automatically configure LVM on our host.

■ **Note** Both of the complete Kickstart files presented in this section are available with the source code for this book, which you can find in the Source Code area of the Apress website (`http://www.apress.com`).

Configuration Management

We've shown you throughout this book that configuring a Linux server includes quite a few tasks, for example, configuring hosts; creating users; and managing applications, daemons, and services. These tasks can be repeated many times in the life cycle of one host in order to add new configurations or remedy a configuration that has changed through error, entropy, or development. They can also be time-consuming and are generally not an effective use of time and effort.

The usual first response to this issue is to try to automate the tasks, which leads to the development of custom-built scripts and applications. Very few scripts developed in this ad hoc manner are ever published, documented, or reused, so the same tool is developed over and over again. These scripts also tend not to scale well, and they often require frequent maintenance.

Configuration management tools can automate these tasks efficiently and allow a consistent and repeatable life cycle for your hosts. We're going to show you how to use one of these tools, Puppet, to automate your configuration.

Introducing Puppet

Puppet (`http://reductivelabs.com/`) is an open source configuration management tool that relies on a client/server deployment model. It is licensed using the GPLv2 license. We're going to give you an overview of Puppet and how to use it to configure your environment and your hosts.

When using Puppet, central servers, called Puppet *masters*, are installed and configured. Client software is then installed on the target hosts, called *puppets* or *nodes*, that you wish to manage. Configuration is defined on the Puppet master, compiled, and then applied to the Puppet clients when they connect.

To provide client/server connectivity, Puppet uses XML-RPC web services running over HTTPS on TCP port 8140. To provide security, the sessions are encrypted and authenticated with internally generated self-signed certificates. Each Puppet client generates a self-signed certificate that is then validated and authorized on the Puppet master.

Thereafter, each client contacts the server—by default every 30 minutes, but this interval is customizable—to confirm that its configuration is up to date. If a new configuration is available or the configuration has changed, it is recompiled and then applied to the client. If required, a configuration update can also be triggered from the server, forcing configuration down to the client. If any existing configuration has varied on the client, it is corrected with the original configuration from the server. The results of any activity are logged and transmitted to the server.

At the heart of how Puppet works is a language that allows you to articulate and express your configuration. Your configuration components are organized into entities called *resources*, which in turn can be grouped together in *collections*. Resources consist of the following:

- Type
- Title
- Attributes

Listing 19-6 shows an example of a simple resource.

Listing 19-6. *A Puppet Resource*

```
file { "/etc/passwd":
       owner => "root",
       group => "root",
      mode => 0644,
}
```

The resource in Listing 19-6 is a `file` type resource. The `file` resource configures the attributes of files under management. In this case, it configures the `/etc/passwd` file and sets its owner and group to the `root` user and its permissions to 0644.

The resource type tells Puppet what kind of resource you are managing—for example, the `user` and `file` types are used for managing user and file operations on your nodes, respectively. Puppet comes with a number of resource types by default, including types to manage files, services, packages, cron jobs, and file systems, among others.

■ **Tip** You can see a full list of the built-in resource types at `http://reductivelabs.com/trac/puppet/wiki/TypeReference`. You can also develop your own types in the Ruby programming language.

The resource's title identifies it to Puppet. Each title is made up of the name of the resource type (e.g., `file`) and the name of the resource (e.g., `/etc/passwd`). These two values are combined to make the resource's title (e.g., `File["/etc/passwd"]`).

■ **Note** In a resource title, the name of the resource type is capitalized (`File`), and the name of the resource is encapsulated in block brackets and double quotes (`["/etc/passwd"]`).

Here the name, `/etc/passwd`, also tells Puppet the path of the file to be managed. Each resource managed by Puppet must be unique—for example, there can be only one resource called `File["/etc/passwd"]`.

The attributes of a resource describe the configuration details being managed, such as defining a particular user and the attributes of that user (e.g., the groups the user belongs to or the location of the user's home directory). In Listing 19-6, we are managing the owner, group,

and mode (or permissions) attributes of the file. Each attribute is separated from its value with the => symbols and is terminated with a comma.

Puppet also uses the concept of collections, which allow you to group together many resources. For example, an application such as Apache is made up of a package, a service, and a number of configuration files. In Puppet, each of these components would be represented as a resource (or resources) and then collected together and applied to a node. We'll look at some of these collection types later in this chapter.

Installing Puppet

Let's start by installing Puppet. For Puppet, the client and server installations are slightly different, and we'll show you how to install each.

Red Hat Installation

On Red Hat, on both servers and clients, you need to install some prerequisites, including the Ruby programming language.

```
$ sudo yum install ruby ruby-shadow
```

Next, you'll add the EPEL repository to your host and then install a number of packages from that repository. You can add the EPEL repository, if it's not already added to your Yum configuration, by adding the `epel-release` RPM.

```
$ sudo rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/➥
epel-release-5-3.noarch.rpm
```

On the server or master, you install the `puppet`, `puppet-master`, and `facter` packages from the EPEL repository.

```
$ sudo yum install puppet puppet-server facter
```

The `puppet` package contains the client, the `puppet-master` package contains the server, and the `facter` package contains a system inventory tool called Facter. Facter gathers information or facts about your hosts that is used to help customize your Puppet configuration.

On the client, you need to install only the `puppet` and `facter` packages.

```
$ sudo yum install puppet facter
```

Ubuntu Installation

On Ubuntu, the required packages are `puppet`, `puppetmaster`, and `facter`. The `puppet` package contains the Puppet client, the `puppetmaster` package contains the master, and the `facter` package contains the Facter system inventory tool.

On the server or master, you need to install this:

```
$ sudo apt-get install puppet puppetmaster facter
```

On the client, you need the following:

```
$ sudo apt-get install puppet facter
```

■ **Note** Installing the `puppet`, `puppetmaster`, and `facter` packages will also install some prerequisite packages.

Configuring Puppet

We'll start configuring Puppet by setting up our Puppet master. Our configuration, including our *manifests* (the files containing our host configuration), will be located under the `/etc/puppet` directory. Puppet's principal configuration file is located at `/etc/puppet/puppet.conf`.

We're going to store our actual configuration in a directory called `manifests` under the `/etc/puppet` directory. This directory is created when the Puppet packages are installed. The manifests directory needs to contain a file called `site.pp` that is the root of our configuration. Let's create that now.

```
$ sudo touch /etc/puppet/manifests/site.pp
```

■ **Note** Manifest files containing configuration have a suffix of `.pp`.

We're also going to create three more directories, `classes`, `nodes`, and `files`, that will hold additional configuration files.

```
$ sudo mkdir /etc/puppet/manifests/{classes,files,nodes}
```

The `files` directory will hold any files we want to send to our managed clients. The `nodes` directory will contain definitions of our clients or nodes. The `classes` directory will contain our classes. *Classes* are collections of resources—for example, an Apache class containing all the resources needed to configure Apache.

We'll continue our configuration by defining these new directories in our `site.pp` file, as shown in Listing 19-7.

Listing 19-7. *The `site.pp` File*

```
import "nodes/*.pp"
import "classes/*.pp"

$puppetserver = "puppet.example.com"
```

The `import` statement tells Puppet to load all files with a suffix of `.pp` in both the `nodes` and `classes` directories into Puppet. The `$puppetserver` statement sets a variable. In Puppet, configuration statements starting with a dollar sign ($) are variables and can be used to specify values in a Puppet configuration.

In Listing 19-7, we've created a variable that contains the fully qualified domain name of our Puppet server, enclosed in double quotes.

■ **Note** You can find quoting rules for Puppet at `http://reductivelabs.com/trac/puppet/wiki/LanguageTutorial#quoting`.

We recommend you create a DNS `CNAME` for your Puppet host (e.g., puppet.example.com), or add it to your `/etc/hosts` file:

```
# /etc/hosts
127.0.0.1 localhost
192.168.0.1 au-mel-ubuntu-1 au-mel-ubuntu-1.example.com puppet puppet.example.com
```

■ **Note** We cover how to create `CNAME`s in Chapter 9.

We also need to specify the fully qualified domain name in our `/etc/puppet/puppet.conf` configuration file. The configuration file is divided into sections, and each section configures a particular element of Puppet. For example, the `[puppetd]` section configures the Puppet client, and the `[puppetmasterd]` section configures the Puppet master or server. We're going to add only one entry, `certname`, to this file to get started. We'll add the `certname` value to the `[puppetmasterd]` section (if the section doesn't already exist in your file, then create the section).

```
[puppetmasterd]
certname=puppet.example.com
```

■ **Note** Replace puppet.example.com with the fully qualified domain name of your host.

Adding the `certname` option addresses a bug with the Ruby SSL code present on many Ubuntu and Red Hat hosts. You can read more about the precise bug at `http://reductivelabs.com/trac/puppet/wiki/RubySSL-2007-006`.

Setting Up Puppet File Serving

In addition to configuring a variety of resources, Puppet can also serve out files—for example, it can deliver configuration files to a node. This file server is configured via the `/etc/puppet/fileserver.conf` configuration file. You can see a sample of this file in Listing 19-8.

Listing 19-8. *The `fileserver.conf` Configuration File*

```
[files]
  path /etc/puppet/manifests/files
  allow 192.168.0.0/24
  allow 127.0.0.1
```

File server configuration is very simple. We specify a file share—in our case called `files`—and enclose it in square brackets [ ]. Next, we specify the `path` for the file share, which here is the directory we created earlier, `/etc/puppet/manifests/files`. We can then specify `allow` and/or `deny` statements to control access to our file share. Here we've allowed access to the file share from anyone in the 192.168.0.0/24 subnet and from the localhost, 127.0.0.1.

■ **Tip** You can read more about file serving at `http://reductivelabs.com/trac/puppet/wiki/FileServingConfiguration`.

Puppet Firewall Configuration

The Puppet master runs on TCP port 8140. This port needs to be open on your master's firewall, and your client must be able to route and connect to the master. To do this, you need to have some appropriate firewall rules on your master, such as the following:

```
-A Firewall-eth0-INPUT -p tcp -m state --state NEW --dport 8140 -j ACCEPT
```

The preceding line allows access from everywhere to TCP port 8140.

Starting Puppet Server

The Puppet master can be started via an `init` script. On Red Hat, we run the `init` script with the `service` command like so:

```
$ sudo service puppetmaster start
```

On Ubuntu, we run it using the `invoke-rc.d` command.

```
$ sudo invoke-rc.d puppetmaster start
```

■ **Note** Output from the daemon can be seen in `/var/log/messages` on Red Hat hosts and `/var/log/daemon.log` on Ubuntu hosts.

Connecting Our First Client

Once you have the Puppet master configured and started, you can configure and initiate your first client. On the client, as we mentioned earlier, you need to install the `puppet` and `facter` packages using your distribution's package management system. We're going to install a client on the gateway.example.com host and then connect to our puppet.example.com host. This installation will also create a `/etc/puppet` directory with a `puppet.conf` configuration file.

When connecting our client, we first want to run the Puppet client from the command line rather than as a service. This will allow us to see what is going on when we connect. The Puppet client binary is called `puppetd`, and you can see a connection to the master initiated in Listing 19-9.

Listing 19-9. *Puppet Client Connection to the Puppet Master*

```
gateway$ puppetd --server=puppet.example.com --no-daemonize --verbose
info: Creating a new certificate request for gateway.example.com
info: Creating a new SSL key at /var/lib/puppet/ssl/private_keys/gateway.example.com➥
.pem
warning: peer certificate won't be verified in this SSL session
notice: Did not receive certificate
```

In Listing 19-9, we executed the `puppetd` binary with a number of options. The first option, `--server`, specifies the name or address of the Puppet master to connect to. We can also specify this in the `main` section of the `/etc/puppet/puppet.conf` configuration file on the client.

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

The `--no-daemonize` option runs the Puppet client in the foreground and prevents it from running as a daemon, which is the default behavior. The `--verbose` option enables verbose output from the client.

■ **Tip** The `--debug` option provides further output that is useful for troubleshooting.

In Listing 19-9, you can see the output from our connection. The client 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 client. The client is now waiting for the master to sign its certificate and enable the connection. At this point, the client is still running and awaiting the signed certificate. It will continue to check for a signed certificate every two minutes until it receives one or is canceled (using Ctrl+C or the like).

■ **Note** You can change the time the Puppet client will wait using the `--waitforcert` option. You can specify a time in seconds or `0` to not wait for a certificate.

Now on the master, we need to sign the certificate. We do this using the `puppetca` binary.

```
puppet$ puppetca --list
gateway.example.com
```

■ **Tip** You can find a full list of the binaries that come with Puppet at `http://reductivelabs.com/trac/puppet/wiki/PuppetExecutables`.

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

```
puppet$ puppetca --sign gateway.example.com
Signed gateway.example.com
```

■ **Note** You can sign all waiting certificates with the `puppetca --sign --all` command.

On the client, two minutes after we've signed our certificate, we should see the following entries:

```
notice: Got signed certificate
notice: Starting Puppet client version 0.24.7
err: Could not retrieve catalog: Could not find default node or by name with➥
'gateway.example.com, gateway' on node gateway.example.com
```

The client is now authenticated with the master, but we have another message present:

```
err: Could not retrieve catalog: Could not find default node or by name with➥
'gateway.example.com, gateway' on node gateway.example.com
```

The client has connected, but because we don't have anything configured for the client, we received an error message.

■ **Caution** It is important that the time is accurate on your master and client. SSL connections rely on the clock on hosts being correct. If the clocks are incorrect, then your connection may fail with an error, indicating that your certificates are not trusted. You can use NTP, which we discuss in Chapter 9, to ensure your host's clocks are accurate.

Creating Our First Configuration

Now our client has connected and we're going to add some configuration for it. On the Puppet master, we need to add a node definition and some configuration to apply to our client.

We'll start with the node configuration. To do this, we're going to create a file called `gateway.example.com.pp` in our `/etc/puppet/manifests/nodes/` directory. You can see the contents of this file in Listing 19-10.

Listing 19-10. *Our Node Configuration*

```
node "gateway.example.com" {
    include sudo
}
```

The node directive defines a node or client configuration to Puppet. Each client needs a node directive, and inside the node you define the configuration that applies to the client. You specify the client name, enclosed in double quotes, and then you specify the configuration that applies to it inside curly braces { }.

■ **Note** You can also specify a special node called default. If no node definition exists, then the contents of this node are applied to the client.

You can specify multiple clients in a node directive by separating each with a comma like so:

```
node "gateway.example.com", "headoffice.example.com" {
    include sudo
}
```

■ **Note** At this stage, you can't specify nodes with wildcards (e.g., *.example.com). Puppet, however, does have an inheritance model in which you can have one node inherit values from another node. You can read about node inheritance at http://reductivelabs.com/trac/puppet/wiki/LanguageTutorial#nodes.

Inside our node definition you can see the include directive. The include directive adds classes (collections of resources) to our client's configuration. In this case, we're adding a class called sudo. You can include multiple classes by using multiple include directives or separating each class with commas.

```
include sudo,sshd
```

Let's add this class to our Puppet configuration. We're going to create a file called sudo.pp in the /etc/puppet/manifests/classes directory. You can see its contents in Listing 19-11.

Listing 19-11. *The sudo Class*

```
class sudo {
    package { sudo:
         ensure => present,
    }

    file { "/etc/sudoers":
        source => "puppet://$puppetserver/files/etc/sudoers",
        owner => "root",
        group => "root",
        mode => 0440,
    }
}
```

VERSION CONTROL

As your configuration gets more complicated, you should consider adding it to a version control system such as Subversion. A version control system allows you to record and track changes to files, and is commonly used by software developers. For configuration management, version control allows you to track changes to your configuration. This is highly useful if you need to revert to a previously known state or make changes without impacting your running configuration.

You can find information about how to use Subversion at `http://svnbook.red-bean.com/` and some specific ideas about how to use it with Puppet at `http://reductivelabs.com/trac/puppet/wiki/VersionControlPuppet`.

You can see we've added a `class` directive and called it `sudo`. The contents of our class are specified between the curly braces.

We've specified two resources inside our class, a `package` resource and a `file` resource. The package resource, `Package["sudo"]`, specifies that the package `sudo` must be installed using the attribute `ensure` and setting its value to `present`. To remove the package, we would set the `ensure` attribute to `absent`. If we wanted to ensure that the `sudo` package was always up to date, we would specify a value of `latest` for the `ensure` attribute like so:

```
package { sudo:
     ensure => latest,
}
```

On every Puppet run, the client will now check that the currently installed version of the `sudo` package is the latest. If it is the latest version, then Puppet will do nothing; if a later version is available, then Puppet will install it.

To manage your packages, Puppet uses the default package manager. For example, on Red Hat it will use `yum` and on Debian it will use `aptitude` to install, remove, or update your package. This is one of the more convenient features of Puppet—you specify the `package` resource, and Puppet detects the appropriate package manager to use and installs the required package. You don't need to do anything else or even understand how the package manager works.

■ **Note** We've discovered that Puppet calls the various items it can configure *types*, for example, the `package` type. The code that interacts with a particular operating system (e.g., the code that interacts with the Yum package manager) is called a *provider*. Each type may have multiple providers. For example, the `package` type has providers for Yum, Aptitude, up2date, Ruby Gems, ports, portage, rug, and OSX DMG files, among many other package managers. The `package` providers allow Puppet to configure packages on a wide variety of Unix operating systems and Linux distributions.

Next, we've specified a file resource, `File["/etc/sudoers"]`. You've seen some of the attributes of this resource before: the `owner`, `group`, and `mode` attributes. The `source` attribute allows Puppet to retrieve a file from the Puppet file server and deliver it to the client. The value of this attribute is the name of the Puppet file server and the location and name of the file to retrieve.

```
puppet://$puppetserver/files/etc/sudoers
```

Let's break down this value. The `puppet://` part specifies that Puppet will use the Puppet file server protocol to retrieve the file.

■ **Note** Currently this is the only protocol available. In future versions of Puppet, the file server will support other protocols, such as HTTP or rsync. This support is expected in versions after 0.25.0.

The `$puppetserver` variable contains the hostname of our Puppet server. We created this variable and placed it in our `site.pp` file earlier. Instead of the variable, you can specify the hostname of the file server here.

```
puppet://puppet.example.com/files/etc/sudoers
```

The next portion of our `source` value specifies the file share and the specific file to serve. Here the share is `files`, which we created earlier in our `fileserver.conf` file, and the specific file to load is `/etc/sudoers`. This assumes the file `sudoers` is in the directory `/etc/puppet/manifests/files/etc/sudoers`. Let's copy a `sudoers` file there now. We'll use the default `sudoers` file on our host.

```
puppet$ mkdir -p /etc/puppet/manifests/files/etc/
puppet$ cp /etc/sudoers /etc/puppet/manifests/files/etc/sudoers
```

CREATING A PUPPET CONFIGURATION

The best way to convert your existing configuration to Puppet is to start small. Choose a function or application, such as `sudo` or the SSH daemon, and convert its configuration management from manual to managed with Puppet. When these functions are stable, add additional components to your Puppet configuration. A good way to approach this task is to classify your hosts by their functions. For example, our gateway.example.com host runs a number of services such as Apache, Postfix, and OpenVPN, so a logical first step would be to configure these services and then slowly add the additional functions also supported on this host.

Applying Our First Configuration

We've created our first configuration and we're going to apply it on our client. Back on the gateway.example.com host, we run the Puppet client again, as shown in Listing 19-12.

Listing 19-12. *Applying Our First Configuration*

```
gateway$ puppetd --server=puppet.example.com --no-daemonize --verbose
notice: Starting Puppet client version 0.24.7
info: Caching catalog at /var/lib/puppet/localconfig.yaml
notice: Starting catalog run
info: Filebucket[/var/lib/puppet/clientbucket]: ➥
```

```
Adding /etc/sudoers(a8ae43fcf346af54d473b13b17d6d037)
notice: //Node[gateway.example.com]/sudo/File[/etc/sudoers]: Filebucketed to  with ➥
sum a8ae43fcf346af54d473b13b17d6d037
notice: //Node[gateway.example.com]/sudo/File[/etc/sudoers]/source: replacing from ➥
source puppet://puppet.example.com/files/sudoers with contents ➥
{md5}7255bc94cd66fc3416f991aed81ab447
notice: //Node[gateway.example.com]/sudo/File[/etc/sudoers]/mode: mode changed '640' ➥
to '440'
notice: Finished catalog run in 3.52 seconds
```

■ **Tip** Puppet logs to the `/var/log/messages` file on Red Hat and the `/var/log/daemon.log` file on Ubuntu.

In Listing 19-12, we've run the Puppet client, `puppetd`, and connected to the master. We can see a catalog run commence on our client. In Puppet, the combined configuration to be applied to a host is a catalog and the process of applying it is called a *run*.

■ **Tip** You can find a glossary of Puppet terminology at `http://reductivelabs.com/trac/puppet/wiki/GlossaryOfTerms`.

In the first step of our run, we see a line describing a `filebucket`. The `filebucket` is a special type used to back up files, and you'll note we didn't specify this type. Puppet automatically backs up files that are going to be changed or replaced (in this case, our host already has an `/etc/sudoers` file and we're going to replace it with our new file from the Puppet master). Here Puppet will copy the file to a directory on the client, usually underneath `/var/lib/puppet/clientbucket`. This means if we want to get this file back, we can manually retrieve it.

■ **Tip** Puppet can also back up the file to our master using the `filebucket` type. See `http://reductivelabs.com/trac/puppet/wiki/TypeReference#filebucket`.

After backing up the file, Puppet copies the new `/etc/sudoers` file from the master.

■ **Tip** Puppet also has a testing mode called noop. In this mode, Puppet doesn't update your configuration but merely tells you what it would have done. This is very useful for testing your configuration prior to applying it. You can run the Puppet client in noop mode by using the `--noop` option with the `puppetd` command.

Lastly, Puppet has changed the permissions of the new file to 0440. But why didn't Puppet change the owner and group of the file, too? Well, in this case, the file is already owned by the root user and belongs to the root group, so Puppet changes nothing. Puppet will make changes on the client only if something needs to be changed. If your current configuration is correct, then Puppet will not do anything.

So that's it. Puppet has configured our client. If the Puppet client was now running as a daemon, it would wait 30 minutes (by default) and then connect to the master again to check if the configuration has changed on our client or if a new configuration is available from the master. We can adjust this run interval using the runinterval option in the /etc/puppet/puppet.conf configuration file.

```
[puppetd]
runinterval=3600
```

Here we've adjusted the run interval to 3600 seconds, or 60 minutes.

PUPPET BEST PRACTICES

Puppet configuration can get quite complex. One of Puppet's users, Stanford University, has written a best practices guide that offers some advice about how to configure Puppet. The Puppet Best Practices guide is available at http://reductivelabs.com/trac/puppet/wiki/PuppetBestPractice. Remember, this document contains only guidelines, and the information within may not completely suit your environment.

Specifying Configuration for Multiple Hosts

We've barely scratched the surface of Puppet's configuration capabilities, so let's look at extending our current configuration to multiple clients or nodes. We'll demonstrate how to differentiate configuration on two clients and apply slightly different configuration to each.

To implement this differentiation, we're going to use Puppet's partner tool, Facter. Facter is a system inventory tool that returns facts about your hosts. We can run Facter from the command line using the facter binary to see what it knows about our gateway.example.com client.

```
gateway$ sudo facter
architecture => i386
domain => example.com
facterversion => 1.5.2
fqdn => gateway.example.com
hardwareisa => i686
hardwaremodel => i686
hostname => gateway
id => root
interfaces => eth0,eth1
ipaddress => 192.168.0.254
ipaddress_eth0 => 192.168.0.254
ipaddress_eth1 => 10.0.2.155
kernel => Linux
kernelrelease => 2.6.18-92.el5
```

```
kernelversion => 2.6.18
operatingsystem => RedHat
operatingsystemrelease => 5
...
```

We've shown you a small selection of the facts available in Facter, but you can see that it knows a lot about our host, including its name, network information, operating system, and even the release of the operating system.

So how is this useful to Puppet? Well, each of these facts is available to Puppet as a variable. Puppet runs Facter prior to applying any configuration, collects the client's facts, and then sends them to the Puppet master for use in configuring the client. For example, the `hostname` fact is available in our Puppet configuration as the variable `$hostname`. Let's look at an example in Listing 19-13.

MORE ABOUT FACTER

Facter supports adding facts via environment variables. Any environment variable on your client that is prefixed with `FACTER` (e.g., `FACTER_LOCATION`) will be available as the variable `$location` in your Puppet configuration. You can read more about this at `http://reductivelabs.com/trac/puppet/wiki/FrequentlyAskedQuestions#can-i-access-environmental-variables-with-facter`.

Facter is also highly extensible. With a small amount of Ruby code, you can add your own facts, for example, information customized to your environment. You can read about how to add these custom facts at `http://reductivelabs.com/trac/puppet/wiki/AddingFacts`.

Listing 19-13. *Using Facts*

```
class sudo {
    package { sudo:
         ensure => present,
    }

    file { "/etc/sudoers":
        source => "puppet://$puppetserver/files/$hostname/etc/sudoers",
        owner => "root",
        group => "root",
        mode => 0440,
    }
}
```

You can see the `sudo` class we previously defined with one small change in the `source` attribute of the `File["/etc/sudoers"]` resource.

```
puppet://$puppetserver/files/$hostname/etc/sudoers
```

We've added the `$hostname` variable to the `source` attribute's value. Now instead of looking for the file in the `/etc/puppet/manifests/files/etc/` directory, it will look in the `/etc/puppet/manifests/files/$hostname/etc` directory. When the client connects, the `$hostname` variable

will be replaced with the hostname of the client connecting—for example, if the gateway host connected, then the source attribute would become /etc/puppet/manifests/files/gateway/etc. We can now have a different sudoers file for particular clients; for instance, we could have the following:

```
/etc/puppet/manifests/files/gateway/etc/sudoers
/etc/puppet/manifests/files/headoffice/etc/sudoers
```

Depending on which client connected, they would get a file appropriate to them. But this isn't the only use for facts. We can also use facts to determine how to configure a particular node, as shown in Listing 19-14.

Listing 19-14. *A Fact in a Case Statement*

```
node default {

    case $operatingsystem
        redhat:  { include redhat  } # include the redhat class
        ubuntu  { include ubuntu } # include the ubuntu class
        default: { include generic } # include the generic class
    }
}
```

Here we created our default node definition, which is the node configuration used for all nodes that don't explicitly have a node defined. Inside this node definition, we used a feature of the Puppet language, a case statement. The case statement, a concept common to many programming languages, specifies a result based on the value of a variable—in this case, the $operatingsystem fact, which contains the name of the operating system running on the client (e.g., redhat for Red Hat or ubuntu for Ubuntu).

■ **Tip** Puppet has two other types of conditionals: selectors and if/else clauses. You can read about these at http://reductivelabs.com/trac/puppet/wiki/LanguageTutorial#conditionals.

In Listing 19-14, if the value of the $operatingsystem is redhat, then the redhat class is included on this client. If the value is ubuntu, then the ubuntu class is included. The last value, default, is the behavior if the value does not match either redhat or ubuntu. In this case, the generic class is applied to the client.

In a case statement, we can also specify multiple values by separating each with a comma like so:

```
    case $operatingsystem
        redhat,centos:  { include redhat  } # include the redhat class
        ubuntu,debian  { include ubuntu } # include the ubuntu class
        default: { include generic } # include the generic class
    }
```

Now if the value is redhat or centos, then the redhat class would be included, and if the value is ubuntu or debian, then the ubuntu class would be included.

We've used another Puppet conditional, a selector, in Listing 19-15.

Listing 19-15. *A Selector*

```
service { "sshdaemon":
    name => $operatingsystem ? {
        redhat => "sshd",
        ubuntu => "ssh",
        default => "ssh",
    }
    ensure => running,
```

In Listing 19-15, we introduced a new type, service, that manages services on hosts. We've titled our service resource sshdaemon, but we've used another attribute called name to specify the name that will be used to start or stop the service on the client. We've used a Puppet language construct called a *selector*, combined with the $operatingsystem fact, to specify the name attribute. This is because on each operating system we've specified, the SSH daemon is called something different. For example, on Red Hat the SSH daemon's init script is called sshd, while on Ubuntu it is called ssh.

The name attribute uses the value of the $operatingsystem fact to specify what the daemon will be called on each distribution. Puppet, in turn, uses this to determine what service to start or stop. So if the value of the $operatingsystem fact is redhat, then the service resource will use the name sshd to manage the SSH daemon. The default value is used when the value of the $operatingsystem is neither redhat nor ubuntu.

Lastly, the ensure attribute has been set to running to ensure the service will be started. We could set the ensure attribute to stopped to ensure it is not started.

■ **Note** The Puppet language has a lot of useful features. You can find a full tutorial of the language at http://reductivelabs.com/trac/puppet/wiki/LanguageTutorial.

Relating Resources

Resources in Puppet also have the concept of *relationships*. For example, a service resource can be connected to the package that installs it. Using this, we could trigger a restart of the service when a new version of the package is installed. This allows us to do some useful things. Consider the simple example in Listing 19-16.

Listing 19-16. *Requiring Resources*

```
class ssh {
    service { "sshdaemon":
        name => $operatingsystem ? {
            redhat => "sshd",
            ubuntu => "ssh",
```

```
            default => "ssh",
        },
        ensure => running,
        require => File["/etc/ssh/sshd_config"],
    }

    file { "/etc/ssh/sshd_config":
        path    => "/etc/ssh/sshd_config",
        owner   => root,
        group   => root,
        mode    => 644,
        source => "puppet://$puppetserver/files/etc/ssh/sshd_config",
        notify  => Service[sshdaemon],
    }
}
```

Listing 19-16 shows a new class called `ssh`, which contains the `service` resource we created in Listing 19-15. We have created a `file` resource to manage the `/etc/ssh/sshd_config` file. You've seen almost all the attributes in these resources except `require` in the `service` resource and `notify` in the `file` resource. These are not, however, normal attributes—they are called *metaparameters*. Let's look at each metaparameter and see what it does.

The `require` metaparameter allows you to build a relationship to one or more resources. Any resource you specify in the `require` metaparameter will be configured *before* this resource, hence Puppet will process and configure the `File["/etc/ssh/sshd_config"]` resource before the `Service["sshdaemon"]` resource. This approach ensures that the appropriate configuration file is installed prior to starting the SSH daemon service. You could do a similar thing with a `package` resource.

```
class httpd {
    package { "httpd":
        ensure => present,
    }

    service { "httpd":
        ensure => running,
        enabled => true,
        require => Package["httpd"],
    }
}
```

Here the `package` resource, `Package["httpd"]`, must be installed before the `Service["httpd"]` service can be started.

■ **Tip** We've also added the `enabled` attribute to the `Service["http"]` resource. When set to `true`, this attribute ensures our service starts when the host boots (similar to using the `chkconfig` or `update-rc.d` command).

We've also specified another metaparameter, this one called `notify`, in Listing 19-16. This metaparameter has been added to the `File["/etc/ssh/sshd_config"]` resource. The `notify` metaparameter tells other resources about changes and updates to a resource. In this case, if the `File["/etc/ssh/sshd_config"]` resource is changed (e.g., if the configuration file is updated), then Puppet will notify the `Service["sshdaemon"]` resource, causing it to be run and thus restarting the SSH daemon service.

■ **Tip** Two other relationships you can construct are subscribe and before. You can see both of these at `http://reductivelabs.com/trac/puppet/wiki/TypeReference#metaparameters` and also read about other available metaparameters you may find useful.

Using Templates

In addition to retrieving files from the Puppet file server, you can also make use of a template function to apply specific values inside those files to configure a service or application. Puppet templates use a Ruby template language called ERB (see `http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/`). It's very simple to use, as you can see in Listing 19-17.

Listing 19-17. *Using Templates*

```
file { "/etc/ssh/sshd_config":
            path    => "/etc/ssh/sshd_config",
            owner   => root,
            group   => root,
            mode    => 644,
            content => template("/etc/ssh/sshd_config.erb"),
            notify  => Service[sshdaemon],
    }
```

In Listing 19-17, we used the same `File["/etc/ssh/sshd_config"]` resource we created earlier, but we exchanged the `source` attribute for the `content` attribute. With the `content` attribute, rather than a file being retrieved from the Puppet file server, the contents of the file are populated from this attribute. The contents of the file can be specified in a string like so:

```
content => "this is the content of a file",
```

Or, as Listing 19-17 shows, we can use a special Puppet function called `template`. To use the template function, we specify a template file, and Puppet populates any ERB code inside the template with appropriate values. Listing 19-18 shows a very simple template.

Listing 19-18. *sshd_config Template*

```
Port 22
Protocol 2
ListenAddress <%= ipaddress_eth0 %>
```

```
SyslogFacility AUTHPRIV
PermitRootLogin no
PasswordAuthentication no
ChallengeResponseAuthentication no
GSSAPIAuthentication yes
GSSAPICleanupCredentials yes
UsePAM yes
X11Forwarding yes
Banner /etc/motd
```

We've only used one piece of ERB in Listing 19-18, to specify the `ListenAddress` of our SSH daemon, `<%= ipaddress_eth0 %>`. The `<%= value %>` syntax is how you specify variables in a template. Here we specified that Puppet should set the `ListenAddress` to the value of the `$ipaddress_eth0` variable. This variable is, in turn, the value of the `ipaddress_eth0` fact, which contains the IP address of the eth0 interface.

When we now connect a client that applies the `File["/etc/ssh/sshd_config"]` resource, the value of the `ipaddress_eth0` fact on the client will be added to the template and then applied on the client in the `/etc/ssh/sshd_config` file.

You can perform a wide variety of functions in an ERB template—more than just specifying variables, including basic Ruby expressions. You can read about how to use templates in more detail at `http://reductivelabs.com/trac/puppet/wiki/PuppetTemplating`, and you can see another example of a typical template at `http://reductivelabs.com/trac/puppet/wiki/Recipes/ResolvConf`.

Puppet looks for templates in a directory specified by the `templatedir` configuration option. This option usually defaults to `/var/lib/puppet/templates`. We're going to override this to put our templates with the rest of our manifests and configuration. In the `puppet.conf` configuration file on the Puppet master, we add the following:

```
[puppetmasterd]
templatedir=/etc/puppet/manifests/templates
```

The template specified in Listing 19-17 is now located at `/etc/puppet/manifests/templates/etc/ssh/sshd_config`.

PUPPET AND PROVISIONING

You can also combine Puppet with your provisioning environment and boot servers. You can find instructions on how to combine Puppet with both Cobbler and Ubuntu Preseed at `http://reductivelabs.com/trac/puppet/wiki/BootstrappingWithPuppet`.

Definitions

You've already seen one type of Puppet collection: a class. There is another type of collection: the definition or `define` directive. *Definitions* are used for a configuration that has multiple instances on a client. The best way to think about a definition is as a reusable snippet of configuration that you can call with arguments.

This reuse is also the key difference between classes and definitions. Classes contain single instances of resources—for example, a class could contain a package resource that defined the `httpd` package. This package will exist only once on a node and hence is installed, removed, or managed using a class. But some configurations exist multiple times on your clients—for example, the `httpd` server may have multiple virtual hosts defined. You could then create a definition to configure virtual hosts and pass in appropriate arguments to configure each. As long as each set of arguments is different, Puppet will configure the new virtual host every time the definition is evaluated.

A definition is created by using the `define` directive, specifying a title for the definition and then listing any arguments in brackets. The definition itself is specified next and is enclosed in curly braces. Listing 19-19 contains a definition that runs a script to configure a new virtual host.

Listing 19-19. *Definition*

```
define new_vhost ( $ipaddress, $domain ) {
       exec { "/usr/sbin/create_vhost --vhost $title --ip $ipaddress --domain ➥
$domain":
       }
}

new_vhost { vhost1:
       ip => "192.0.2.155",
       domainname => "vhost1.example.com"
}
```

In Listing 19-19, we created a definition called `new_vhost` that has arguments of the variables `$ipaddress` and `$domain`. Inside the definition, we used the `exec` resource type (this is another type; it executes an external script). We specified three variables, `$title` and the previously mentioned `$ipaddress` and `$domain` variables, in the script defined in the `exec` resource type.

■ **Note** The `$title` variable is available in all resources and contains the title of the resource.

On the next lines, we have actually called the `new_vhost` definition. We call it much like we define a resource type. We specify the name of the definition being called, the title, which in this case is `vhost1` (which is also the value of the `$title` variable). We then specify the remaining variables to be passed to the definition in the same format as we would specify attributes in a resource.

If we use this definition, we'll see a log message on the client much like the following:

```
notice://new_vhost[vhost1]/Exec[/usr/sbin/create_vhost --vhost vhost1 --ip➥
192.0.2.155 --domain vhost1.example.com]/returns: executed successfully
```

■ **Tip** You also saw a definition being used in a recipe, `http://reductivelabs.com/trac/puppet/wiki/Recipes/ResolvConf`, that we linked to earlier.

More Puppet

We've barely touched on Puppet in this chapter—there's a lot more to see. In the sections that follow, we'll describe some of the topics we haven't covered that you can explore further to make the best use of Puppet.

Modules

You've already seen two collections of resources, classes and definitions, but Puppet has another, more complex type of collection called a *module*. You can combine collections of classes, definitions, templates, files, and resources into modules. Modules are portable collections of configuration; for example, a module might contain all the resources required to configure Postfix or Apache.

You can read about how to use modules at `http://reductivelabs.com/trac/puppet/wiki/PuppetModules`. Also on this page are links to a huge number of user-contributed modules. Someone else has almost certainly written a module to configure a service or application you may want, and in many cases you can just download and reuse these modules to save having to write ones yourself.

You can read about how to create your own modules and how they are structured at `http://reductivelabs.com/trac/puppet/wiki/ModuleOrganisation`.

Functions

Puppet also has a collection of functions. *Functions* are useful commands that can be run on the Puppet master to perform actions. You've already seen two functions: `template`, which we used to create a template configuration file, and `include`, which we used to specify the classes for our nodes. There are a number of other functions, including the `generate` function that calls external commands and returns the result, and the `notice` function that logs messages on the master and is useful for testing a configuration.

You can see a full list of functions at `http://reductivelabs.com/trac/puppet/wiki/FunctionReference` and find some documentation on how to write your own functions at `http://reductivelabs.com/trac/puppet/wiki/WritingYourOwnFunctions`.

Reports

Puppet has the ability to report on events that have occurred on your nodes or clients. Reporting is pretty basic right now; you can see the current reports at `http://reductivelabs.com/trac/puppet/wiki/ReportReference`. You can also find some examples of how to use reports and build your own custom reports at `http://reductivelabs.com/trac/puppet/wiki/ReportsAndReporting`.

External Nodes

As you might imagine, when you begin to have a lot of nodes your configuration can become quite complex. If it becomes cumbersome to define all your nodes and their configuration in manifests, then you can use a feature known as *external nodes* to better scale this. External nodes allow you to store your nodes and their configuration in an external source. For example, you can store node information in a database or an LDAP directory. You can read more about external and LDAP nodes at `http://reductivelabs.com/trac/puppet/wiki/ExternalNodes` and `http://reductivelabs.com/trac/puppet/wiki/LDAPNodes`, respectively.

Environments

One of Puppet's most useful features is support for the concept of environments. *Environments* allow you to specify configuration for particular environments—for example, you might have development, test, and production environments. Puppet allows you to maintain parallel sets of configuration for each environment and apply them to different clients.

This is a powerful mechanism for catering for a variety of scenarios—for example, creating a development ➤ testing ➤ production life cycle for managing custom-designed infrastructure and applications. You can also use environments to maintain separate sets of configuration for sites or security zones—for example, separate configuration for Demilitarized Zones (DMZs) and the internal network.

You can read about environments at `http://reductivelabs.com/trac/puppet/wiki/UsingMultipleEnvironments`.

Documenting Your Configuration

A bane of many system administrators is documentation, both needing to write it and needing to keep it up to date. Puppet has some useful built-in tools that allow you to document your configuration manifests and modules. By running the `puppetdoc` binary, you can have Puppet scan your manifests and configuration and generate documentation in HTML, among other formats. You can read about manifest documentation at `http://reductivelabs.com/trac/puppet/wiki/PuppetManifestDocumentation`.

SCALING PUPPET

The default Puppet master uses an internal web server called Webrick. Generally this web server supports only a small number of clients, usually 30 to 50. To scale Puppet beyond this number of clients, you need to make use of the alternative web server, Mongrel. You can read about Puppet scalability at `http://reductivelabs.com/trac/puppet/wiki/PuppetScalability` and `http://reductivelabs.com/trac/puppet/wiki/UsingMongrel`.

Troubleshooting Puppet

Puppet has a big and helpful community as well as extensive documentation. In addition, one of the authors of this book, James Turnbull, has written a book specifically about Puppet called *Pulling Strings with Puppet* (Apress, 2008). In addition to the book, you can see Puppet's wiki

at `http://reductivelabs.com/trac/puppet/wiki`. It includes a lot of useful resources such as the following reference pages:

- *Configuration Reference*: `http://reductivelabs.com/trac/puppet/wiki/TypeReference`
- *Type Reference*: `http://reductivelabs.com/trac/puppet/wiki/TypeReference`
- *Report Reference*: `http://reductivelabs.com/trac/puppet/wiki/ReportReference`
- *Function Reference*: `http://reductivelabs.com/trac/puppet/wiki/FunctionReference`

Also helpful on the wiki are the following resources:

- *Language Tutorial*: `http://reductivelabs.com/trac/puppet/wiki/LanguageTutorial`
- *Getting Started guide*: `http://reductivelabs.com/trac/puppet/wiki/GettingStarted`
- *FAQ*: `http://reductivelabs.com/trac/puppet/wiki/FrequentlyAskedQuestions`

In addition, you can find details of Puppet's mailing lists, the #puppet IRC channel, and a variety of other resources, including the ticketing system at `http://reductivelabs.com/trac/puppet/wiki/GettingHelp`.

Summary

In this chapter, we've introduced you to some simple provisioning tools that make the process of building and installing your hosts quick and easy. You've learned how to do the following:

- Configure a network boot infrastructure.
- Automatically boot a host with a chosen operating system.
- Install a chosen operating system and automatically answer the installation questions.

We've also introduced a configuration management tool, Puppet, that will help you consistently and accurately manage your environment. You've learned how to do the following:

- Install Puppet.
- Configure Puppet.
- Use Puppet to manage the configuration of your hosts.
- Use the more advanced features of Puppet.

In the next chapter, we'll demonstrate how you make can use of virtualization and virtual servers to deploy your infrastructure cheaply and efficiently.