# Puppet Workshop

## Configuration Management Made Easy

**Jeroen van Meeuwen, RHCE**

**Stefan Hartsuiker, RHCE**

# Puppet Workshop
# Configuration Management Made Easy

Author Jeroen van Meeuwen, RHCE *j.van.meeuwen@ogd.nl*
Author Stefan Hartsuiker, RHCE *s.hartsuiker@ogd.nl*


This book is a configuration management workshop wrapped around puppet, the next-generation configuration management utility that has proven to be simple, straightforward, flexible, scalable, stable, fast, extensible and most importantly, truely Free.

# Preface

This is the Configuration Management Workshop reader as provided to you by the Operator Groep Delft. This reader is composed form both an introduction to Configuration Management with Puppet as well as a reference for later use.

## 1. About the Contributors

### Author

*Jeroen van Meeuwen* (RHCE, LPIC-2, MCP, CCNA) is currently a Senior System Engineer, specialized in Linux systems and Systems Architecture, working for Operator Groep Delft in The Netherlands. His experience with computers goes back to the early '90s, with a Philips P2000T being over a decade old, little tapes containing programs but most importantly games, and 16K memory cartridges. Since 1998, he has been involved with Red Hat Linux (5.2 at that time), and was an early adopter of Fedora Core Linux in November 2003, until his first real contributions to Free and Open Source Software were made in 2005.

As a contributor to Free and Open Source Software within the Fedora community, amongst other programs, Jeroen has developed Revisor, a Python framework to build distributions with. With regards to Configuration Management, Jeroen currently maintains or co-maintains -amongst other packages- the entire stack of packages related to Puppet

### Contributors

*Stefan Hartsuiker* (RHCE, LPIC-2, MCP) is currently a System Engineer, specialized in maintaining Linux Systems, also working for Operator Groep Delft in the Netherlands and a colleague of Jeroen van Meeuwen. His experience with computers started in the late 80's with a Acorn Electron, a smaller version of the BBC Electron (no it had nothing to do with the British Broadcasting Company), for which he had to type in several pages worth of lines of code just to play a game. His first introduction to Linux came in the early 90's with the slackware distribution on about 32 floppies. Since then he has used and maintained Suse, Debian, RedHat and Fedora systems.

Currently Stefan is a contributor to Revisor and working on packaging Free and Open Source Software in the Fedora Community. He is also a fan of Puppet and is working with Jeroen on designing an infrastructure for Puppetmasters.

## 2. About this Document



## 3. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

## 3.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

**`Mono-spaced Bold`**

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**-**`Alt`**-**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**-**`Alt`**-**`F7`** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`Mono-spaced Bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

**Proportional Bold**

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find…** from the

[1] https://fedorahosted.org/liberation-fonts/

> **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

***Mono-spaced Bold Italic*** or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

## 3.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
```

```
books         Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests   Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }

}
```

## 3.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

**Note**

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

**Important**

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

**Warning**

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

# 4. Feedback

Should you find any discrepancies or additional information for this documentation, we would appreciate to hear from you.

Our mailing lists are:

- ***http://lists.fedorahosted.org/mailman/listinfo/courses-users/***
  Our "users" mailing list where anyone can comment on the course materials offered, provide other means of feedback and ask questions when things appear to not be as clear as they intend to be.
- ***http://lists.fedorahosted.org/mailman/listinfo/courses-devel/***
  Our development mailing list for anyone seeking to get involved in the project.
- ***http://lists.fedorahosted.org/mailman/listinfo/courses-commits/***
  This mailing list is used to send any changes made to any of the documents to anyone subscribed.

# Introduction

Welcome to the Puppet Workshop (or Configuration Management workshop). Today's workshop is comprised of the following topics, in order of appearance:

### Topics in this workshop

- *Introduction to Configuration Management*

  A short introduction to what part of configuration management it is that Puppet is able to assist with.

- *Introduction to Puppet*

  An introduction to how Puppet resolves many of the issues that exist without configuration management.

- *Puppet Terminology*
- *How Puppet Works*

  A birds view of Puppet's operations

- *Puppet Features*
- *Troubleshooting Puppet*
- *Setting up Puppet*

  Set up Puppet so that it keeps on going

- *How to use Puppet*

  Tips & tricks on using Puppet

- *Other Things To Do With Puppet*

  Advanced stuff Puppet can also do

- *Best Practices*

  More Tips & Tricks on Puppet

## 1.1. Target Audience

The primary audience for this book is, of course, Linux system administrators and engineers seeking to implement, further enhance or extend their knowledge about configuration management in general and by using the next-generation configuration management utility Puppet in particular.

# Introduction to Configuration Management

## 2.1. What is Configuration Management?

Within virtually every organization, there's probably a number of systems running Linux, Solaris, Mac OS X and/or HP-UX. These systems need to be configured appropriately to be able to function properly. Some will need special drivers, and all of them will need correct DNS settings, certain packages installed and certain other packages removed, users created, and SSH host keys exchanged. The more systems, the more these diverge in the configuration they need, diverge in the way this configuration needs to be applied, and the more these configurations will show discrepancies arising over time.

More specifically, an organization may have a couple of webservers, fileservers, a DNS and a DHCP server, a number of desktop PCs, and a number of laptops. The laptops may need slightly different system configuration (no LDAP authentication, and with a VPN client installed, for example), and the desktop PCs may need different applications installed then the servers, and so forth. Yet, between, say, a hundred desktop PCs, you would want the configuration to be as similar as possible. You may want to diverge between a software developer's desktop PC and a desktop PC in Human Resources, but in essence these are desktop profiles diverging on the application level, applied upon a stable system configuration which remains the same, or similar at least.

By the time the organization grows, replaces the hardware, upgrades to another version of the operating system, or applies changes, the challenge to making everything work yet maintain a similar configuration between all nodes becomes bigger. While every attempt made to control the situation can be called a form of configuration management, the solution without a configuration management framework is often comprised of:

1. a number of scripts (with or without revision control), to move around files, install packages, perform daily check-ups,
2. NFS mounts with programs pre-installed, so that nodes can mount these NFS shares and the software needs to be provided once, in one location, for all to share,
3. file server shares with pre-compiled drivers, or driver sources being compiled on the nodes by scripts running on the nodes,
4. terminal servers or desktop servers like with FreeNX, so that configuration concentrates on a smaller number of boxes

This means that workarounds for actual (user) problems maybe require an additional `if-then-else` in one or the other script, and updates to programs installed require manual compilation and installation. The success rate of these solutions never reaches 100%, and as it turns out the longer such a implemented solution runs, the more exotic problems become and the more machines will fail to remain up-to-date regardless of any attempt made to fix the issue; simply because it becomes to diverse and unmaintainable.

### 2.1.1. Configuration Management

Generally speaking, with configuration management, it's about managing the configuration of one or more organizational resources in order to have it be in a state in which it can perform the operations

required by, and possibly critical to, the organization's operations. In addition to that, configuration management often concerns administrative tasks as to what systems provide a service and what *SLA* or *OLA* is applicable to that service, as well as the purchase date, location of the system, responsible party, etcetera.

In this workshop though, we are not going to explore configuration management of a coffee machine. Instead we look at the computers in a network running any platform but the one from a prominent proprietary North America-based vendor. We are talking automation and further enhancement of Computer Systems Administration.

When managing the operating system and software running on mainframes, servers, desktop PCs and laptops, you may find yourself looking for answers to questions such as:

- How do I manage what packages are installed on a given system?
  - How do I manage the configuration of those packages (this software)?
  - How do I make sure these packages are updated?
- How do I make sure the services that every machine needs to run are actually running?
- How do I manage monitoring the services or a machine's state?
- A job needs to run periodically (maybe via **crontab**), but how do I make sure it is run, and how can I change or remove the job later?
- Given different operating systems and operating system versions, how do I make sure I apply the correct routine for adding a user, starting a service, install/update/remove a package?

## 2.1.2. Configuration Management Requirements

This section is about what you would want Configuration Management to do for you, as a system administrator for the systems within your organization:

- **Maintain consistency across systems**

  Consistency across systems is key in understanding where a problem might come from and assessing where problems may be first introduced. If each and every system is unique, you may end up searching for unique aspects of the system's configuration in order to determine the cause of a problem, while if systems are mostly consistent and the exceptions to the rule are easily determined, you may have found the problem even before your users experience the consequences.

  **Consistency !== Equality**

  Of course keeping systems consistent in their configuration doesn't say all your systems should be entirely equal, because that would not be feasible for many organizations and defeat the purpose of configuration management. Needless to say though, having all systems be entirely unique defeats part of the purpose of configuration management as well.

- **Categorize systems**

  Grouping systems into categories like (for example) *desktop*, *server* and/or *laptop*, helps in applying changes to one category, such as installing **GNOME** or keeping systems up-to-date according to a schedule that may (servers) or may not (desktops, laptops) need a service or maintenance window.

### Different profiles

More generally speaking, different profiles for each of these categories may be defined as well. A developer's desktop most likely has different requirements then a publicly accessible information booth at the reception desk.

- **Version Control**

  Version control lets you keep track of changes applied to the overall configuration management framework, which is important because if you are managing different aspects of a (large) number of systems, and something goes wrong, the changes applied to the configuration Puppet uses will most likely be the first clue as to what caused the new problem and lets you recover relatively fast. Additionally, version control adds a layer that also gives you the chance to perform access control, to have notifications of changes applied sent to interested people, and to branch off.

- **Overview of systems' tasks and services**

  Being able to quickly tell what a system does exactly, and how it differs from another system not only aids in performing risk assessments (impact of a given change), but may also help in determining the impact of a change beforehand, as well as determine the impact of an unexpected system or service interruption. Providing an example to the latter; if you update httpd across systems (whether tested or untested), but the new software version doesn't work as expected, a configuration management framework should be able to quickly give you an overview of impacted systems and services.

- **Updating systems**

  Some systems can be updated irregularly, such as desktop PCs, but need to be kept up-to-date nonetheless. Other systems have service and/or maintenance windows, such as servers, and thus need a very regular and strict update schema, compliant with the update policies in place.

## 2.2. Problems without Configuration Management

There's a number of challenges an organization might encounter when not implementing some form of configuration management, such as:

1. **Different operating systems**

   If you have a diverse organization in terms of the operating systems your systems run, applying the same or similar configuration to a set of different operating systems is challenging in terms of adding a user or setting a password on one operating system is not the same as adding a user or setting a password on another operating system. The same applies to installing, updating or removing a package, and so forth. Additionally the more different operating systems you have, the harder managing any given *system resource* becomes. Some commands for day-to-day administrative tasks may be equal, or similar, but most of them are and/or behave different.

2. **Different distributions**

   Although an organization may not have different distributions running right now, sooner or later, an organization will migrate from one distribution to another; That is practically inevitable. If an organization does have different distributions running, practical problems such as the location of certain files become evident, as well as different interfaces to resource-management (like adding a user with **useradd** or **adduser**).

3. **Different versions of distributions**

   Different versions of distributions, or, more accurately, the different versions of the applications shipped with each distribution version, as well as the configuration settings for updated programs that come with the distributions, can form a challenge when or if the organization does not have a proper configuration management framework in place. Note that even though an organization may not have different versions of a distribution right now, at some point the organization will need to upgrade to the next available release.

4. **Different tasks to perform**

   Each different system in an organization is performing one or more tasks that may be unique to the system or may be shared between a group of systems, but with many different tasks being performed throughout the organization's infrastructure, keeping track of what system performs which task, keeping these systems up to date and configuring them to have the required packages installed for each of the tasks they perform, tackling the problem becomes harder.

5. **Different ways to perform a task**

   Within an organization that has multiple servers performing the same task, keeping a similar state or perform a task in a similar manner is challenging in the sense that without configuration management, you are most likely to find more and more ways to purging old files from `/tmp/` and `/var/tmp/`, for example. The same differentiation may apply to how webservers' VirtualHosts are configured, or how a NFS share is mounted (mount options in particular).

6. **Different nodes**

   This one goes to hardware-specific needs and configuration. When the systems in an organization are not all of the same brand, make and model, or each system has different harddisk layouts, or needs different videocard drivers, you are basically keeping lists and making choices based on those lists.

7. **Different services**

   Different services of course are configured differently, as far as configuration file locations and syntax are concerned. However, figuring out the best way to apply certain configuration to a system for each service is less efficient without configuration management. You might adjust a script or two and/or adjust the source repository from which you pull updates to each system, but the changes may turn out to only apply to that system that needed the exception to the rule instead of focussing on a more general solution to the problem once, and apply that solution multiple times, over and over again. Overview is key here, even if your standard environment exists of numerous different profiles.

8. **Interfaces to a system resource**

   This is probably the hardest one if you are not using any configuration management framework. Given different operating systems, distributions and/or distribution versions, in which case any combination of the three only makes the problem harder to solve, you are most likely to encounter so many different ways to manage a given system resource, that a simple script or routine cannot cover all of them -and remain comprehensible and maintainable. One example is adding a user to the system, and making the user a group member of several groups. You may find routines ranging from using **useradd** or **adduser** depending on the distribution used, to writing out ldifs from a template and using **ldapadd** or **ldapmodify** depending on whether the user already exists or not, and whether the user is a member of the group already.

9. **Fast pace changes**

   Changes that need to be applied sooner rather then later are often only applied by the time a crontab job polls for new configuration, or when a system reboots. This does not work in cases where changes need to be applied quickly, such as when a package installed on some or all systems exposes vulnerabilities that make the system remotely exploitable.

## 2.3. Not So Technical Aspects

In addition to the problems you may encounter with or without configuration management, there's a number of problems or challenges that are not so technical, but you may want to see resolved by a configuration management utility;

1. **Applying changes**

   Applying changes to multiple machines at once may become a problem depending on the size of the organization or the amount of remote and direct control that you have over your systems. There was a time when changing the DNS servers for a set of systems required one to log on to the console of each system and edit **`/etc/resolv.conf`** manually. You can see the problem become bigger if the organization does not have 20 systems, but 1200.

2. **Keeping track of changes**

   Another challenge is keeping track of the changes applied to each system. Even with configuration management, errors can be made and systems may behave unexpectedly, in which case you will want to know what changed on these systems, and how to recover to an operational state. Keeping track of changes without a configuration management framework however is a little harder, but with configuration management, you have reports (changes applied to a system in a nice overview), and most advisebly you have the configuration for Puppet stored in a Source Control Management system, or SCM system, like CVS, SVN, Mercurial, or GIT.

3. **Staging changes**

   Staging changes is a huge must-have in case changes are radical or might destroy a normal system's operation (even if temporary). For such changes, you would want to test the changes first, and with Puppet, you get this in the form of *environments*. Additionally, in case any critical component needs to change, proper Change Management then requires you to Build & Test the solution prior to implementation. This is often not a very bad idea to relieve stress in case the implemented solution does not work, especially if the change is time-constrained such as with service windows.

4. **Exceptions to the rule**

   As important as keeping systems consistent is being able to name and point out the exceptions to the rules that you set. Of course, every organization has exceptions and until the point where high-availability, high-performance, load-balancing or virtualization clusters are deployed (or storage pools for that matter), no two systems are alike. Trying to keep a consistent configuration amongst all your systems doesn't change that. Note though, being able to reproduce the reasoning behind the exceptions to the rule is important in recovering from (unexpected) system or service interruptions.

5. **Restoring a system**

   Restoring a system to it's normal operations often requires you to have a backup of the most recent configuration files and transactional data for the machine. Although configuration management with puppet does not facilitate the backup of transactional data, it does facilitate the

backup of configuration files, taking away the rather boring task of having to manually select which items need to be backed up on a regular basis, and restored when recovering the system.

# Introduction To Puppet

Puppet is a solution to many of the problems set forth in *Section 2.2, "Problems without Configuration Management"*, and thus perfect for a workshop on Configuration Management.

Another solution may be *CFEngine*. We have chosen not to use CFEngine for several reasons:

- Puppet has an open development model, whereas CFEngine has not. This means that the changes and bugfixes, and more importantly innovation and development is in the hands of you and me.
- The level of abstraction of system resources that Puppet enables you to use allows you to concentrate on the bigger picture, rather then needing to figure out again and again, and then specify again and again, how a certain task is to be performed on a given operating system, distribution and/or specific distribution version. CFEngine however is a very low-level utility, perfect for keeping 800 identical machines in shape, but becomes worse with any desirable discrepancy between systems because of that low-level management.

For a more detailed CFEngine vs. Puppet poem, visit *http://reductivelabs.com/trac/puppet/wiki/CfengineVsPuppet*.

## 3.1. What Does Puppet Do?

Puppet offers a high-level abstraction of system resources like you would encounter on any given system, such as users, services and packages. Seeing as how different operating systems and different distributions each have different interfaces (*providers* in puppet terms), to these system resources, managing a package to be installed, updated, removed or be of a certain version includes a lot of `if-then-else` statements in a script you would write to manage that particular system resource; one package.

On Debian, Ubuntu and derivative distributions for example, the package provider may be **apt**, **dpkg**, **smart**, **alien**, **PackageKit**, while on Fedora, Red Hat and it's derivatives, the package provider may be **rpm**, **yum**, **PackageKit**, **apt** or **smart**. Although some of these package managers can be combined, while others can not, and systems usually stick to their natively integrated package manager, figuring out such while actually trying to manage the result of what a package manager does could be seen as a lot of work for little gain.

Another difference between distributions is how services can be started, or configured to start up when the machine boots. A **service** script may be available, or **`/etc/init.d/`** may contain scripts to start and stop a service. Also, some of these service providers may have `status`, `reload` and `restart` command parameters, whereas others may not have. Additionally, using **chkconfig** to configure the runlevels the service should be enabled or disabled in may not be available on all systems. The list of examples here is of course, exhaustive.

By abstracting these system resources into *types*, Puppet takes on the headaches for most operating system and distribution specific interfaces to managing these system resources. It knows, or figures out all by itself, what provider to use given a *type*.

### Abstraction of system resources

Abstraction of the system resources into so-called *types* causes the administrator to only need to configure a type, such as *package*, *user*, *cron*, and so forth, directly addressing the system resource on the managed system without the exact specifics of the managed system. The configuration management utility itself will figure out what package manager backend to use, whether it's apt, yum, rpm, dpkg, smart or PackageKit.

This is a Puppet example to ensure user *sysadmin* exists on a system:

```
user { "sysadmin":
ensure => present
}
```

### YP Client Example

Puppet example to ensure the *ypbind* package is installed and the most recent version, *ypbind* is correctly configured, and the *ypbind* service is running:

```
package { "ypbind":
    ensure => latest
}

file { "/etc/yp.conf":
    source => "puppet:///files/yp.conf",
    notify => Service["ypbind"],
    require => Package["ypbind"]
}

service { "ypbind":
    enable => true,
    ensure => running,
    require => [
        File["/etc/yp.conf"],
        Package["ypbind"]
    ]
}
```

The above example is called a *manifest*, built out of *types* (package, file, service), which, once defined in a manifest, are referred to as *resources*. See also *Appendix A, Puppet Terminology*

The configuration file for this YP client is pulled from `puppet:///files/yp.conf`, the puppetmaster's fileserver, which is where you put the file, possibly from within a Version Control System.

## 3.2. Ordering

As you can see in the *YP Client Example*, Puppet does not just manage resources and hopes for the best, but allows one resource to require another resource (`File["/etc/yp.conf"] { require => Package["ypbind"] }`), and for one resource to notify another resource on changes (`File["/etc/yp.conf"] { notify => Service["ypbind"] }`).

This capability ensures that files are placed in the right location after the appropriate package is installed, and before the service starts. This also allows changes to configuration files to notify the service so that it can reload or restart.

## 3.3. Write Once, Apply Many Times

Returning to the *YP Client Example*, if many systems need to become YP clients, of course you can apply the snippet to all nodes:

```
node 'node1.example.com' {
    package { "ypbind":
        ensure => latest
    }

    file { "/etc/yp.conf":
        source => "puppet:///files/yp.conf",
        notify => Service["ypbind"],
        require => Package["ypbind"]
    }

    service { "ypbind":
        enable => true,
        ensure => running,
        require => [
            File["/etc/yp.conf"],
            Package["ypbind"]
        ]
    }
}

node 'node2.example.com' {
    package { "ypbind":
        ensure => latest
    }

    file { "/etc/yp.conf":
        source => "puppet:///files/yp.conf",
        notify => Service["ypbind"],
        require => Package["ypbind"]
    }

    service { "ypbind":
        enable => true,
        ensure => running,
        require => [
            File["/etc/yp.conf"],
            Package["ypbind"]
        ]
    }
}
```

This of course is not very efficient; One change would need to be applied many times still, and errors are easily made. Besides, it doesn't scale well. Group the YP Client relevant resources into a *class* instead, and then include the class with each node:

```
# Example /etc/puppet/manifests/classes/yp.pp

class yp::client {
    package { "ypbind":
        ensure => latest
    }

    file { "/etc/yp.conf":
        source => "puppet:///files/yp.conf",
        notify => Service["ypbind"],
        require => Package["ypbind"]
    }

    service { "ypbind":
        enable => true,
        ensure => running,
        require => [
            File["/etc/yp.conf"],
            Package["ypbind"]
        ]
    }
}
```

```
# Example /etc/puppet/manifests/site.pp

import "classes/*.pp"

node 'node1.example.com' {
    include yp::client
}

node 'node2.example.com' {
    include yp::client
}

node 'node3.example.com' {
    include yp::client
}
```

Although this looks much more scalable and long-term sustainable, we're not there yet. If the `yp::client` is not the only one class of what makes up, say, a desktop, and other classes to be included are named `foo1` through `foo1000` (meaning, a lot of different ones), then here's what happens next;

```
# Example /etc/puppet/manifests/site.pp

import "classes/*.pp"

node 'node1.example.com' {
    include yp::client
```

```
    include foo1
    include foo2
    include fooX
    [...skip 996 lines for abbrevity...]
    include foo1000
}

node 'node2.example.com' {
    include yp::client
    include foo1
    include foo2
    include fooX
    [...skip 996 lines for abbrevity...]
    include foo1000
}

node 'node3.example.com' {
    include yp::client
    include foo1
    include foo2
    include fooX
    [...skip 996 lines for abbrevity...]
    include foo1000
}
```

While instead, since these are all using the same "profile", *desktop*, we could create a **`groups/desktop.pp`** that says:

```
class desktop {
    include yp::client
    include foo1
    include foo2
    include fooX
    [...skip 996 lines for abbrevity...]
    include foo1000
}
```

and have **`site.pp`** say:

```
# Example /etc/puppet/manifests/site.pp

import "classes/*.pp"

node 'node1.example.com' {
    include desktop
}

node 'node2.example.com' {
    include desktop
}
```

```
node 'node3.example.com' {
    include desktop
}
```

Note that nodes, like classes, allow a simple form of inheritance, but can only inherit from one other node. As such, you can also use:

```
node desktop {
    include yp::client
}

node 'node1.example.com' inherits desktop {
}

node 'node1.example.com' inherits desktop {
}

node 'node1.example.com' inherits desktop {
}
```

# Puppet Terminology

First, some clarification on the terminology used in this documentation. See also *Appendix A, Puppet Terminology*

- **class**
  A class is a collection of resources applied to a node with a single `include` statement. It groups together a comprehensible set of resources. A class *yp::client* would manage the `File["/etc/nsswitch.conf"]`, `File["/etc/yp.conf"]`, `Package["ypbind"]`, and `Service["ypbind"]` resources.

- **fileserver**
  The fileserver is where the puppet pulls files from. It is normally integrated with the puppetmaster, but it can be an entirely different server, too.

  The fileserver serves files to puppets that request them, but it also serves *templates*, which are parsed on the fileserver (puppetmaster), and passed on to the client as a whole new file.

- **manifest**
  The collection of classes, modules and resources that the *puppetmaster* uses to distribute the appropriate configuration to a *puppet*.

- **module**
  A module is a placeholder for files, manifests, plugins and templates. Creating a module has numerous advantages such as separate version control, separate staging from development through testing to production, and so forth.

  *See also*: *Section 9.1, “Using Modules”*, *Section 9.4, “Using Plugins”*

- **node**
  The client, a node, is an operating system instance running the puppet client application. This can be a regular operating system running directly on top of actual hardware, a virtual guest as well as a virtual host.

- **puppet**
  The client, a node, runs the **puppetd** daemon or service, and is referred to as the *puppet*

- **puppetmaster**
  The puppetmaster is the node that runs the server-side application to a puppet setup.

- **resource**
  A resource is an instantiated *type*. It has been defined and it cannot be undefined. The puppetmaster sends all applicable resources the a puppet, which then applies them. Resources are fundamentally built from a *type*, a *title*, and a list of *attributes*, with each resource type having a specific list of supported attributes.

- **system resource**
  A system resource is a resource available on the node whether it is managed by puppet or not. Unlike what is otherwise understood by system resources, the puppet definition of system resources throughout this documentation does not so much refer to hardware resources like CPU or memory, but rather to manageable aspects of the operating system, like users, packages, services, files, cronjobs, and so forth.

- **type**
  Puppet uses *types* to abstract system resources. Types have parameters such as `ensure => present|absent` in case of a user, or `ensure => installed|absent|latest|1.0-1.el5`, indicating in which state the system resource should be. Each type has a title, which must be unique throughout the manifest, and a list of supported attributes. E.g., there is no `mode => 644` to the package type.

More definition of terms are in *Appendix A, Puppet Terminology*

# How Puppet Works

This is an overview of how puppet works -in a working setup.

1. **The puppet starts for the first time**
   It generates a certificate using the node's FQDN.

   **Note**
   Although not required, it is strongly recommended to have the client use a FQDN that is registered in DNS (forward as well as reverse).

2. **The puppet submits the certificate to the puppetmaster**
   The puppetmaster, also the Certificate Authority, or *puppetca*, needs to sign the certificate before the client can be considered authenticated.

3. **The puppet waits 300 seconds for a signed certificate**
   It this configurable timeout of 300 seconds[1] has passed, the puppet quits.

4. **The puppetmaster signs the certificate**
   To do so, you can either configure the puppetmaster to automatically sign certificates or sign manually. Automatically signing certificates is generally a very bad idea. To manually sign a certificate, use:

   ```
   # puppetca --sign <Certificate-CN>
   ```

   To refuse a certificate, or clean a signed certificate, use:

   ```
   # puppetca --clean <Certificate-CN>
   ```

5. **The puppet receives the signed certificate**
   Immediately thereafter, the puppet starts a configuration run.

   **Warning**
   The time on both the puppetmaster and the puppet cannot vary more then 5 minutes as the certificate generated by the puppet and signed by the puppetca has a validity period. The start time of this validity period is of importance as the validity period has to have at least started by the time the puppet uses the certificate --or else the signed certificate is considered invalid. If the difference in time of these two nodes is more then 5 minutes, you will get a "Certificates not trusted" type of error.

6. **The puppet generates all the facts**
   Most configurations rely on client information to make decisions. When the Puppet client starts, it loads the Facter Ruby library, collects all the facts it can, and passes those facts to the interpreter. When you use Puppet over a network, these facts are passed over the network to the server and the server uses them to compile the client's configuration.

7. **The puppetmaster parses it's manifests**
   The puppetmaster parses through all it's manifests, including the manifests not applicable to the puppet that is polling. It only sends out the manifest applicable to the puppet polling, however.

8. **The puppet receives the manifests**
   When the puppet receives the manifests, it may still contain variables such as `$hostname`, `$operatingsystem` and others, which the puppet fills out with the appropriate values.

9. **The puppet applies the manifest**
   While the puppet applies the manifest, it pulls files from the puppetmaster's *fileserver* after checking the local checksum against the remote checksum. When running with debug output, this will show as:

```
debug: Calling fileserver.list
debug: //Node[node1.example.com]/File[/tmp/foo]/checksum: Initializing
 checksum hash
debug: //Node[node1.example.com]/File[/tmp/foo]: Creating checksum
 {md5}85e53dc9439253a1ec9ca87aeffd9b0b
debug: Calling fileserver.describe
```

10. **Files that are replaced are backed up**
    The puppet sends a copy of the files it replaces back to the puppetmaster.

11. **The puppet reports to the puppetmaster**
    A detailed report of what the puppet has done with the manifests is sent back to the puppetmaster.

12. **The puppet waits for 30 minutes**
    The next run the puppet performs/polls for is after a configurable timeperiod, which defaults to 30 minutes.

A puppet setup is comprised out of the following parts:

### The Puppetmaster

The puppetmaster of course is the core element in a puppet setup. Not only is it responsible for collecting the facts from, and the handing over the appropriate manifest to the puppets, it also takes care of serving the files needed by the manifest, performs the certificate authority task, offers a clientbucket for backing up changed files, maybe stores the configurations in a database, maybe does reporting, and maybe does some graphing.

### The Puppet

Another pretty essential player in this game, as without puppets, there are no strings to pull.

### Good Hostnames

The puppet determines what certificate to use or generate based on it's current hostname. Once a hostname is used to request the certificate exchange with, that is the valid certificate and the node name for the puppet, which the puppetmaster uses to determine the applicable manifests with. An ever changing hostname doesn't help.

# Puppet Features

A few of the Puppet features

## 6.1. Puppet Speaks

Puppet speaks the language or dialect used on the local, managed system, fluently.

## 6.2. Secure Communication

Your files, templates, and potentially passwords, are passed to the puppet using secure communication.

## 6.3. Free and Open Source Software

Puppet has a free development model, and has proven to participate in the Free and Open Source Software world with the true Free spirit.

## 6.4. Facts, not Fiction

When Puppet becomes your main configuration management tool, you want to use facts, not fiction. Puppet takes the most reliable sources for it's facts; the client does not set every single fact.

## 6.5. Repeat, Repeat, Repeat

Write something once. Then use it. And again. And then improve it. And then use it again. And again.

# Setting Up Puppet

In this section, we are going to set up a puppetmaster, and a puppet client. The puppetmaster is going to run the *mongrel* server-type, for setting up a puppetmaster for larger environments.

## 7.1. Installation

The default server type for the puppetmaster is called *webrick*, a single-threaded webserver. The webserver handles the puppets' requests for manifests, certificate exchanges, as well requests for files and templates. Being single-threaded, the webrick webserver can only handle one client at a time. While the puppets poll the puppetmaster with a default interval of 30 minutes, and configuration runs can take longer then 60 seconds, putting more then 25 clients in front of a puppetmaster with a webrick webserver is a very, very bad idea.

There is a multi-threaded webserver in Ruby, called *mongrel*. This is a simple, multi-threaded, but not very feature-rich webserver. For one, it does not perform SSL. For scalability purposes though, the mongrel server type is an absolute must, and can better be chosen as the webserver to handle the puppets' requests, right from the beginning. This however requires a frontend that performs the SSL part of the communications between the puppetmaster and the puppets. We choose Apache's HTTPd for it's excellent performance, flexible configuration, excellent configuration syntax, and because it can be set up as a reverse proxy load balancer, allowing more then one puppetmaster behind the scenes if necessary.

Install the required packages for the puppetmaster:

### Smaller organizations (< ~25 clients)

- The puppetmaster.

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

- (optional) A database server (one of MySQL, SQLite3 or Postgresql), and the appropriate Ruby library. During this workshop, we use MySQL.

```
# yum install mysql-server ruby-mysql
```

- (optional) The Ruby RRDtool library.

```
# yum install ruby-RRDtool
```

### Larger organizations (> ~25 clients)

- A webserver capable of performing as a frontend SSL reverse proxy load balancer, such as the Apache HTTPd webserver.

```
# yum install httpd mod_ssl
```

- The Ruby mongrel library, for better scalability.

```
# yum install rubygem-mongrel
```

- Follow the recipe for < ~25 clients for the other packages that are required (see above).

## 7.2. Configuration

In this section, we walk you through the initial configuration of a puppetmaster with the mongrel server type.

### 7.2.1. Configuring the Puppetmaster

The configuration file for puppet and puppetmaster is **`/etc/puppet/puppet.conf`**. It is a file in INI-like format with sections, keys and values. There's 4 sections of interest,

- **[main]**
  Primarily file locations, directory settings and other globals applicable to both the puppet as well as the puppetmaster.
- **[puppetca]**
  Puppet Certificate Authority (puppetca) settings.
- **[puppetd]**
  Puppet client daemon settings.
- **[puppetmasterd]**
  Puppetmaster daemon settings.

#### 7.2.1.1. Relevant Settings

##### Relevant Settings For The First Run

For the first run of the puppetmaster, the following settings require configuration:

- **[main]**
  The locations where puppet seeks it's configuration and puts it's transitional data. The most important setting is **`vardir`**, which should be set to **`/var/lib/puppet/`**. Further settings include:
  - `logdir = /var/log/puppet/`
  - `rundir = /var/run/puppet/`
  - `ssldir = $vardir/ssl/`

**Note**

If you used a package to install puppet, the defaults should work, but may not comply with your backup strategy. It is the upstream puppet package that cannot cater to each and every distribution or operating system it is available for, and therefore has a set of defaults that will work, but will need to be changed on most platforms.

- **[puppetmasterd]**
  - **certname**
    The puppetmaster certificate's Common Name (CN), for which by default the system's hostname is used. The fully qualified domain name of the system is a pretty reasonable value.

    ```
    $ hostname
    ```

  - **certdnsnames**
    A colon (`:`) seperated list of DNS names resolving to the puppetmaster. Include here:

    1. The short hostname of the system, using the output of:

       ```
       # hostname -s
       ```

    2. **`puppet`**
    3. **`puppet`**, followed by the DNS domain name of the system, using the output of

       ```
       # dnsdomainname
       ```

    4. Any other hostname or fully qualified domain name you want to use for the puppetmaster.

- Another setting to check is whether or not this puppetmaster is going to be the Certificate Authority

  ```
  [puppetmasterd]
      ca = true
  ```

  The default is often set to `true`.

- Whether or not to use autosigning of certificates, using

  ```
  [puppetca]
      autosign = false
  ```

  The default is to *not* use autosigning. Only applicable if `puppetca` is set to `true`.

### Other Relevant Settings

The following settings require review before the puppetmaster is going in production.

- A list of environments using a comma seperated list, in

  ```
  [puppetmasterd]
      environments = development,testing,production
  ```

  *See also*: *Section 9.2, "Environments"*

- Whether or not to use reporting, and what reporting to use (tagmail, store, rrdgraph). To configure the types or reports that should be used by the puppetmaster, use a comma separated list without spaces, in:

```
[puppetmasterd]
    reports = tagmail,store,rrdgraph
```

*See also*: Section 11.2, "Tweaking Reporting"

- The location of tagmail.conf, in order to map tags you give to resources to email addresses the reports should be sent to;

```
[main]
    tagmap = /path/to/tagmail.conf
```

for reporting changes applied to puppets, via email.

*See also*: Section 11.2, "Tweaking Reporting"

#### 7.2.1.2. Configuring the fileserver

Configuring the fileserver is more accurately described in Section 9.6.1, "Fileserver Operations".

#### 7.2.1.3. Minimal site.pp

Create a minimal `site.pp` in **`/etc/puppet/manifests/site.pp`** for the puppetmaster to parse on it's initial startup. Below is an example.

```
#
# Example site.pp
#

# The default node

node default {
}
```

#### 7.2.1.4. Service Configuration

On Red Hat based systems, use **`/etc/sysconfig/puppetmaster`** to configure the service. It has three variables set, of which `PUPPETMASTER_MANIFEST` needs to point to the default manifest to use. Change the default only if you are not going to use environment specific

### 7.2.2. Configuring the SSL Frontend Reverse Proxy Load Balancer

A webserver needs to be configured to handle the SSL XML-RPC requests from the puppets, because the mongrel server type is not capable of performing SSL.

The webserver is going to listen on port 8140, the default port for the puppetmaster to listen for clients. It is going to forward traffic (after being decrypted) to the puppetmaster on 127.0.0.1:8141.

Setting up the webserver requires you install **httpd** and **mod_ssl**. If these are not installed already, use:

```
# yum install httpd mod_ssl
```

Refer to *Appendix B, Example SSL Frontend Reverse Proxy Load Balancer Configuration* for more an example VirtualHost configuration for an SSL frontend reverse proxy load balancer.

# Language Tutorial

The language puppet uses has a strong focus on making the specification of types as easy as possible within a heterogeneous set of resources shared over all the systems you are managing.

## 8.1. Resources

Resource are built by using a type and specifying a list of attributes to that type. Each type has a specific list of supported attributes. You can find the complete list of native types and their attributes on *http://reductivelabs.com/trac/puppet/wiki/TypeReference*

A simple example resource is:

```
file { "/etc/passwd":
    owner => "root",
    group => "root",
    mode => 644
}
```

Any system on which this resource is managed will use it to verify that the **`/etc/passwd`** is owned by the root user and group, and has permissions 644. The field before the colon, in this case "/etc/passwd", is the resource's title, which can be used to refer to the resource later in the manifest.

For simple resources that don't vary much between systems, a single title to refer to is sufficient. Many system resources though vary from system to system. The OpenSSH Client package for example is called **openssh-client** on Debian systems, while it is **openssh-clients** on Fedora systems. To create the package resource to get the OpenSSH Client package installed through a manifest, Puppet allows you to specify:

```
package { "openssh-clients":
    ensure => installed,
    name => $operatingsystem ? {
        "Debian" => "openssh-client",
        default => "openssh-clients"
    }
}
```

Puppet on a Debian system will now direct the package manager to install "openssh-client", while on other systems, the package manager is told to install "openssh-clients".

**Important**

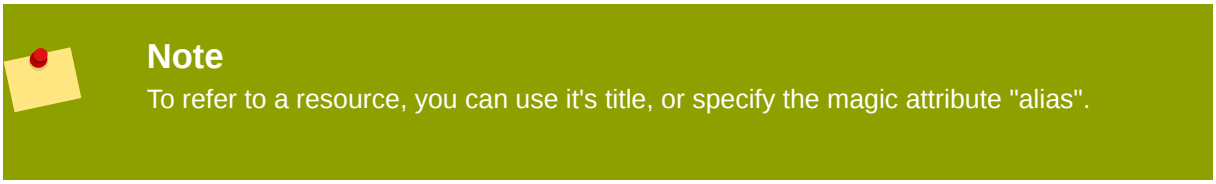
Note that the name of this resource is now conditional, and thus virtually unpredictable from within the rest of the manifests, but still if you wanted to require the OpenSSH Client package you do not need to conditionally require the resource, but instead can use the title of the resource, "openssh-clients".

Even more complex resources are file resources that are located in different paths on different operating systems, such as the configuration file for the OpenSSH Client package.

```
file { "/etc/ssh/ssh_config":
    path => $operatingsystem ? {
        "Solaris" => "/usr/local/etc/ssh/ssh_config",
        "Darwin"  => "/etc/ssh_config",
        "Debian"  => "/etc/openssh/ssh_config",
        default   => "/etc/ssh/ssh_config"
    },
    require => Package["openssh-clients"]
}
```

The *title* for this resource is "/etc/ssh/ssh_config", while the path of the file managed is conditional to the value of the `$operatingsystem` variable. The `$operatingsystem` variable in this case is a fact describing the operating system name. Note that the title of the resource is used as a path if the path parameter had not been seperately configured.

Also note that it is the title of the `Package["openssh-clients"]` used in the reference to the OpenSSH Client package resource.




## 8.1.1. Resource Defaults

You can specify resource defaults so that the default parameter is always used when you make a resource by capitalizing the resource and not specifying a title. With the exec type for example, the command either must use a fully qualified path, or the path parameter to the exec type must be included.

```
Exec {
    path => [
        "/bin/",
        "/usr/bin/",
        "/usr/local/bin/",
        "/sbin/",
        "/usr/sbin/",
        "/usr/local/sbin/"
    ]
}
```

This makes the following work

```
exec { "echo this works": }
```

While normally you would have to specify (in each exec resource):

```
exec { "/bin/echo this works" }
```

Needless to say, **echo** does not exist in **`/bin/`** on all operating sytems you may want to manage.

This type of defaults you would normally store in a file named after the type you specify the defaults for, in the directory **`utils/`** beneath the **`manifests/`** directory.

#### 8.1.1.1. Conditional Resource Defaults

To facilitate the use of different defaults per operating system for example, you can set the resource defaults conditionally, using either `if-then-else` statements, or using `case` statements.

Case statements let you set the resource defaults as follows:

```
case $operatingsystem {
    "Darwin": {
        Exec {
            path => [
                "/foo/bin/",
                "/bar/sbin/",
                (...etc...)
            ]
        }
    }
    "Solaris": {
        Exec {
            path => [
                "/baz/bin/",
                "/baz/sbin/",
                (...etc...)
            ]
        }
    }
    default: {
        Exec {
            path => [
                "/bin/",
                "/usr/bin/",
                (...etc...)
            ]
        }
    }
}
```

**Important**

Resource defaults are not global, but apply to every resource in the current scope. If you define the resource defaults within a class, then the resources within that class and all sub-classes will have defaults set. The only way you can currently specify global defaults is to define them outside of any classes.

## 8.1.2. Resource Collections

There are two ways you can combine multiple resources: Classes and definitions. Classes are only interpreted once per node. Definitions on the other hand can be used as custom types and are meant to be interpreted more then once, each time with different parameters.

### 8.1.2.1. Classes

Classes are introduced by using the `class` keyword, and support a simple form of inheritence. Subclasses are allowed to override resources defined in parent classes.

```
class unix {
    file { [
            "/etc/passwd",
            "/etc/group"
        ]:
        owner => root,
        group => root,
        mode => 644
    }
}

class freebsd inherits unix {
    File["/etc/passwd"] {
        group => "wheel"
    }
    File["/etc/group"] {
        group => "wheel"
    }
}
```

You can use the `undef` keyword when overriding a resource to make the child class act as if the value had never been set in the parent:

```
class freebsd inherits unix {
    File["/etc/passwd"] {
        group => undef
    }
}
```

In this example, nodes which include the unix class will have the password file forced to group root, while nodes including freebsd would have the password file group ownership left unmodified.

It is also possible to add additional values to resource parameters using the **+>** operator:

```
class apache {
    service { "apache":
        require => Package["httpd"]
    }
}
```

```
class apache-ssl inherits apache {
    # host certificate is required for SSL to function
    Service["apache"] {
        require +> File["apache.pem"]
    }
}
```

The above effectively makes the `require` parameter for the `Service["apache"]` resource in the `apache-ssl` class equal to `[ Package["httpd"], File["apache.pem"] ]`.

You can add multiple values by separating each value with commas:

```
class apache {
    service { "apache":
        require => Package["httpd"]
    }
}

class apache-ssl inherits apache {
    Service["apache"] {
        require +> [
            File["apache.pem"],
            File["/etc/httpd/conf/httpd.conf"]
        ]
    }
}
```

The above would make the require parameter in the apache-ssl class equal to `[ Package["httpd"], File["apache.pem"], File["/etc/httpd/conf/httpd.conf"] ]`.

Like resources, you can also require a class, like so:

```
class apache {
    service { "apache":
        require => Class["squid"]
    }
}
```

In this case, the `Class["squid"]` will need to be applied successfully before the `Service["apache"]` resource is applied on the puppet.

### Namespacing

Classes such as the `apache` class in one of the forementioned examples, sub-classed by the `apache-ssl` class, can also be defined within another class:

```
class apache {
    service { "apache":
        require => Package["httpd"]
    }
```

```
    class ssl inherits apache {
        Service["apache"] {
            require +> [
                File["apache.pem"],
                File["/etc/httpd/conf/httpd.conf"]
            ]
        }
    }
}
```

In this case, using the ssl subclass of apache would become **`include apache::ssl`**.

The main difference (operational wise) between the two styles is that if you make a module out of the first above mentioned classes, both classes would have to be in their own file, so as to facilitate automatic loading by puppet. If you do not do this, puppet cannot find the class(es) that have a different name from the name of the module. On its own there is nothing wrong with having each class in its own file, however with large modules this tends to generate a lot of files, something one may wish to avoid.

### 8.1.2.2. Definitions

Definitions are very similar to classes, but are introduced with the `define` keyword, and do not allow inheritance. Definitions also take parameters, which classes do not.

```
class yum {
    define repository($enable = true) {
        file { "/etc/yum.repos.d/$name.repo":
            mode => 644,
            owner => "root",
            group => "root",
            backup => false,
            links => follow,
            source => $enable ? {
                true => [
                    "puppet:///yum/$os/$osver/repos/$name.repo"
                ],
                default => [
                    "puppet:///yum/$os/$osver/repos/$name.repo.disabled"
                ]
            }
        }
    }
}
```

This definition can be used as follows:

```
node 'node1.example.com' {
    yum::repository { "custom":
        enable => true
    }
```

```
}
```

Now, `node1.example.com` gets a file **`/etc/yum.repos.d/custom.repo`** from **`puppet:///yum/$os/$osver/repos/custom.repo`**.

**Note**

The above example makes use of `$os` and `$osver` variables you have to set first. See *Section 12.1, "Setting $os and $osver"* for more details.

# How To Use Puppet

Now that we've set up Puppet, let's see how we can use puppet.

## 9.1. Using Modules

Modules are collections of manifests, files, templates, plugins and possibly documentation on how to use the module, and have a special position in Puppet that allows them to be loaded automatically. Modules live in **`$confdir/modules/`** by default, so you will need to change that using the `modulepath` setting in **`/etc/puppet/puppet.conf`**:

```
[puppetmasterd]
    modulepath = /var/lib/puppet/modules/
```

Of course, for this workshop we are using Linux, so this is the preferred path for us, but this may vary, depending on the Operating System the puppetmaster is run on.

If you are to have multiple environments (development, testing and production for example), you would move modules to **`/var/lib/puppet/modules/environment/`**. For more information on configuring environments see *Section 9.2.1, "Setting Up Environments"*. Modules are comprised of the following:

### *module_name*/manifests/init.pp

This file is mandatory and defines the manifest for the module. The ***module_name*/manifests/init.pp** should at least define 1 class with the same name as the module to allow automatically loading the module.

### *module_name*/files/

The tree of files this module may distribute to clients. Note that this tree should only contain the modules default, which would represent the best default for all of the systems in the organization. If the manifest uses files from the module, address the files as follows from the manifest:

```
source => "puppet:///module_name/path/to/file"
```

So, for example, if the file to distribute is ***module_name*/files/foo**, the `source` value would be:

```
source => "puppet:///module_name/foo
```

For more information on using multiple sources, see *Section 12.2, "Using Multiple Sources"*. For more information on fileserver operations, see *Section 9.6, "Fileserver"*.

### *module_name*/templates/

Templates are basically files with conditional content, or little programs spitting out what the actual file should look like given a set of variables, conditions and logical statements. Templates should go into the ***module_name*/templates/** directory, and can therefrom used as follows:

```
content => template('module_name/template_filename')
```

Example with **`webserver/templates/virtualhost.conf.erb`**:

```
content => template('webserver/virtualhost.conf.erb')
```

#### *`module_name`*`/plugins/puppet/type/`

The location to put custom types to be distributed with this module. Requires `pluginsync` to be set to `true` in the puppet configuration.

#### *`module_name`*`/plugins/puppet/provider/`

The location to put custom providers to be distributed with this module. Requires `pluginsync` to be set to `true` in the puppet configuration.

#### *`module_name`*`/plugins/facter/`

The location to put custom facts to be distributed with this module. Requires `pluginsync` to be set to `true` in the puppet configuration.

#### *`module_name`*`/documentation/`

Just an optional placeholder for documentation you may want to distribute along with the module, if you are to share the module with the community.

## 9.2. Environments

Environments aid in staging developments and not breaking the production. There are three environments you may want to consider using:

### development

The development environment is where you stage developments in your manifests, modules and plugins. You would apply the development environment to a small number of non-critical systems, that maybe are dedicated to the sole purpose of puppet development. These development systems would not necessarily be managed extensively with puppet, and may just apply only the classes you are making changes to. There is no use in applying the `foo` class to a development system if you are making changes to the `bar` module or class.

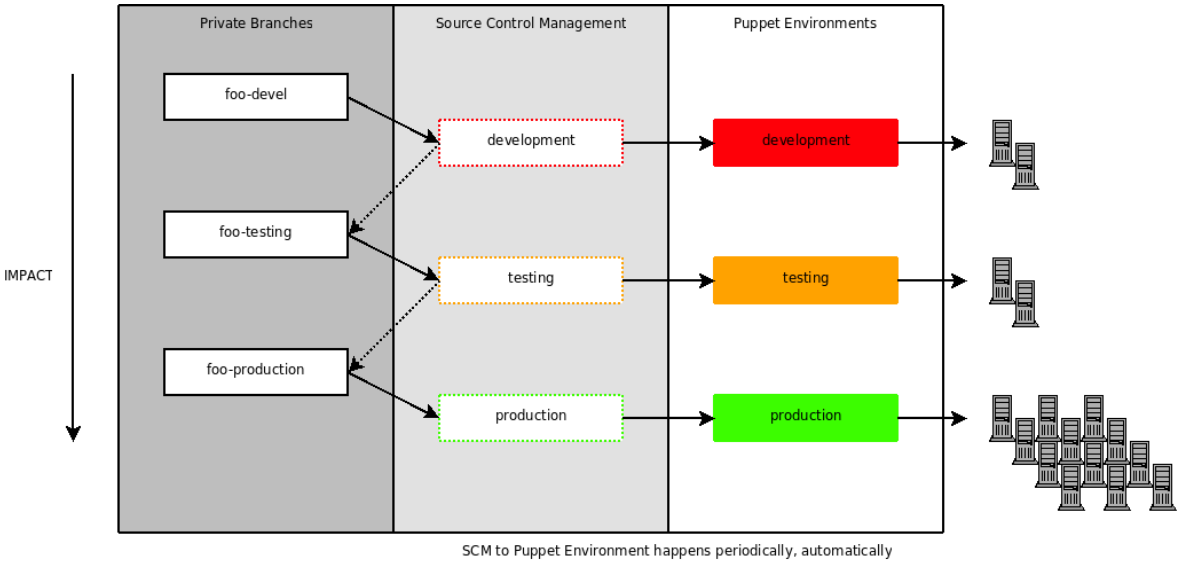
### testing

The testing stage would be the stage where your basic development is finished, and you test the changes in a broader scope. Again, these systems should be non-critical systems, but these systems may also not be dedicated to puppet testing. In the testing stage, you would want to make sure all puppet classes and modules are applied correctly when used in combination with one another. For example, conflicts between modules and/or classes could be caused by duplicate resource definitions, or incorrect inter-module or inter-class dependencies (`require =>`, `notify =>`, `before =>` and `subscribe =>` statements). Although most of the dependencies are used in the development stage as well, using the testing stage makes sure N+X modules cooperate before you destroy your production.

Obviously, the testing stage is entirely optional. One could decide to not use the testing stage the way it is described here, but instead assign a couple of (power-)user desktops and have these users provide feedback on the changes applied, or not use the testing stage at all.

### production

The production stage is the most important stage in the entire process of staging changes. The production environment is where most of the systems within your orgnization are managed in, and as such, changes applied to the production environment need to be proven stable (or system and services may be interrupted).




## 9.2.1. Setting Up Environments

Setting up the environments requires you to configure the available environments in **`/etc/puppet/puppet.conf`**. The relevant settings are:

```
[puppetmasterd]
    environments = development,testing,production
```

Per environment, create a section similar to the following:

```
[development]
    manifest = /var/lib/puppet/manifests/development/site.pp
    modulepath = /var/lib/puppet/modules/development

[testing]
    manifest = /var/lib/puppet/manifests/testing/site.pp
    modulepath = /var/lib/puppet/modules/testing

[production]
    manifest = /var/lib/puppet/manifests/production/site.pp
    modulepath = /var/lib/puppet/modules/production
```

Make sure you put the files and directories in place before restarting the puppetmaster service.

### 9.2.1.1. Client Configuration

Setting the environment the puppet gets it's configuration from is three-fold.

1. The environment you want to run the puppet in needs to be a valid environment, and besides default valid environments *development* and *production*, additional environments need to be configured using the **`environments`** setting in the `[puppetd]` section of **`/etc/puppet/puppet.conf`**:

   ```
   [puppetd]
       environments = development,testing,production
   ```

2. You can set the environment the puppet normally runs against with the **`environment`** setting in the `[puppetd]` section:

   ```
   [puppetd]
       environment = production
   ```

   The default environment for a puppet is *production*

3. The environment can be specified using the `--environment` command-line option to **puppetd**.

## 9.3. Virtual Resources

Using virtual resources is a way to avoid duplicate definitions in your manifests if you define a virtual resource in parent classes, inherit the parent classes in sub-classes, and realize() the virtual resource in the sub-class.

An example where you can use virtual resources is a webserver class:

```
class webserver {
    package { "httpd":
        ensure => installed
    }

    class ssl {
        package { [
                "httpd",
                "mod_ssl"
            ]:
            ensure => installed
        }
    }
}
```

Now, if you were to `include webserver` on a system, everything is well. However, some other class may require the `webserver::ssl` class to be applied to the system as well:

```
class mail {
    class server {
        # Include webserver::ssl for secure webmail capacity
        include webserver::ssl

        webserver::virtualhost { "mail.$domain":
```

```
            enable => true,
            certificate => true
        }

        package { "squirrelmail":
            ensure => installed
        }
    }
}
```

This would result in a duplicate definition of the `Package["httpd"]` resource. Such is easily prevented by making sure the `Package["httpd"]` resource is only defined once:

```
class webserver {
    @package { [
            "httpd",
            "mod_ssl"
        ]:
        ensure => installed
    }

    realize(Package["httpd"])

    class ssl inherits webserver {
        realize(    Package["httpd"],
                    Package["mod_ssl"])
    }
}
```

To declare a package to be a virtual resource, one would place a `@` in front of the keyword package. Then, in the class where the virtual resource is realized, one would call the `realize()` function on the resource, as above in the example.

A more complex but also more appropriate example can be found at *http://git.puppetmanaged.org/?p=puppet;a=blob;f=manifests/init.pp*. Note that in the puppet module linked from puppetmanaged.org, the definition and use of virtual resources is a necessity.

## 9.4. Using Plugins

Using plugins (e.g. custom types, facts, functions and providers) gives you even more control over the behaviour of puppet and extend what it can do. An example custom fact that exposes the version of Python installed is:

```
# From http://reductivelabs.com/trac/puppet/wiki/Recipes/PythonVersion
require 'facter'

pythonversion = nil
if FileTest.exists?("/usr/bin/python")
    pythonversion = %x{python -V 2>&1}.split(" ")[1]
end
```

```
Facter.add("pythonversion") do
    setcode do
        pythonversion
    end
end

Facter.add("pythonmmversion") do
    pythonmmversion = nil
    if pythonversion != nil
        pythonversionsplit = pythonversion.split(".")
        pythonmmversion = pythonversionsplit[0] + "." +
 pythonversionsplit[1]
    end
    setcode do
        pythonmmversion
    end
end
```

Using this new custom fact you can create conditional statements in manifests or templates using `$pythonverion` or `$pythonmmversion`.

## 9.5. Using Manifests from a SCM

Using manifests from a Source Control Management system.

### 9.5.1. Using a Single Tree

Using a single tree in case you want to use environment staging and module support is considered a bad idea, but can still be done. The layout of such a repository can be one of the following:

#### 9.5.1.1. Using SVN

Using SVN allows Windows users to contribute to the repository and/or make edits to files in the repository, which of course has it's advantages.

The layout of an SVN repository would be as follows:

```
/path/to/repository/
    `- trunk/
    `- branches/
        `- testing/
        `- production/
```

**`trunk/`** would represent the development branch in this case. Continuing with **`trunk/`**, the repository layout could look as follows:

```
trunk/
    `- conf/
    `- files/[1]
```

[1]

```
    `- modules/
        `- module1/
            `- files/
            `- manifests/
            `- plugins/
            `- templates/
        `- module2/
            `- (etc)
    `- manifests/
        `- site.pp
        `- classes/
        `- nodes/
        `- utils/
        `- (etc)
```

Checking out the SVN repository (sub-directories) in each various location on the puppetmaster would look like

```
# svn co http://server/svn/puppet/trunk/conf/ /etc/puppet/[2]
# svn co http://server/svn/puppet/trunk/modules/ \
        /var/lib/puppet/modules/development/
# svn co http://server/svn/puppet/trunk/manifests/ \
        /var/lib/puppet/manifests/development/
# svn co http://server/svn/puppet/branches/testing/modules/ \
        /var/lib/puppet/modules/testing/
# svn co http://server/svn/puppet/branches/testing/manifests/ \
        /var/lib/puppet/manifests/testing/
# svn co http://server/svn/puppet/branches/production/modules/ \
        /var/lib/puppet/modules/production/
# svn co http://server/svn/puppet/branches/production/manifests/ \
        /var/lib/puppet/manifests/production/
```

See *Appendix E, SVN Commit Hooks* for how to make the puppetmaster update it's sources when you commit changes to the SVN repository.

#### 9.5.1.2. Using GIT

It is currently not possible to use a single GIT tree to configure all of your puppet environment without requiring manual (or automatic) copying of files and directories from within a copy of the source repository into the location where puppet expects them to be.

### 9.5.2. Multiple Trees

Multiple trees is more efficient because you can merge changes from one stage to another on a per-module basis, rather then merging entire branches with all configuration, manifests, modules and so

Used for files distributed without them being integrated into a module

[2] Note that checking out the puppet configuration from `trunk/` like this almost certainly results in you having to manage the puppetmaster with puppet; See *http://puppetmanaged.org/documentation/puppet-module/* for more information.

forth. Additionally, using multiple trees allows you to upstream modules integrated into your own set of modules.

Using one SCM tree for each module allows you to set access control accordingly, and stage changes per module instead of globally.

### 9.5.3. Modules From Upstream

You can use one or more modules from an upstream provider such as:

- *http://puppetmanaged.org/*
- *http://git.black.co.at/*
- *http://reductivelabs.com/trac/puppet/wiki/Recipes*

Using either a single tree or multiple trees as described above; the environment staging though does not apply to most of the modules available from upstream providers. If the upstream provider does provide branches for each of your environments, you would still only want to use the production or equivalent branch.

If you want to use a module from an upstream provider without the use of the branches for staging, you can specify a colon-seperated list to the `modulepath` setting for your environments to have puppet search through the list of paths. Such would look like:

```
[puppetmasterd]
    environments =  development,testing,production

[development]
    manifest = /var/lib/puppet/manifests/development/site.pp
    modulepath = /var/lib/puppet/modules/upstream/:/var/lib/puppet/modules/
development/

[testing]
    manifest = /var/lib/puppet/manifests/testing/site.pp
    modulepath = /var/lib/puppet/modules/upstream/:/var/lib/puppet/modules/
testing/

[production]
    manifest = /var/lib/puppet/manifests/production/site.pp
    modulepath = /var/lib/puppet/modules/upstream/:/var/lib/puppet/modules/
production/
```

In the example configuration, you would clone, checkout and pull the modules you use from an upstream provider to **`/var/lib/puppet/modules/upstream/module_name/`**.

## 9.6. Fileserver

Puppet's fileserver, again a core component in being able to apply configuration management to your systems, needs to be configured appropriately to allow files, templates, plugins and facts to be distributed to your clients.

## 9.6.1. Fileserver Operations

Using **`/etc/puppet/fileserver.conf`**, you can define fileserver *mounts*.

Each fileserver mount has a `path` attribute and `allow` and/or `deny` statements. The `path` attribute describes where the files are on the local filesystem, and `allow` and `deny` statements allow you to apply access control to these fileserver mounts. Note that `deny` always has precedence over `allow`, and that the order of `allow` and `deny` does therefor not matter.

Also note, that the wildcard matches used in `allow` and `deny` match DNS, or IP addresses. You can also use CIDR notations for the latter.

Additionally, the special `[modules]` mount does not use the `path` attribute but instead figures out the filesystem path based on the `modulepath` setting in **`/etc/puppet/puppet.conf`**, taking the environment used into account as well.

```
# This file consists of arbitrarily named sections/modules
# defining where files are served from and to whom

# Define a section 'files'
# Adapt the allow/deny settings to your needs. Order
# for allow/deny does not matter, allow always takes precedence
# over deny

[facts]
    path /var/lib/puppet/facts
    allow *

[files]
    path /var/lib/puppet/files
    allow *.example.com
    deny *.evil.example.com

[modules]
    allow *
```

The `source =>` parameter in resources now let's you use the following sources:

- **`puppet:///files/`*`path/to/file`***
  Resulting in the file being looked for in **`/var/lib/puppet/files/`*`path/to/file`***.
- **`puppet:///`*`module_name`*`/path/to/file`**
  Resulting in the file being looked for in the path for module *`module_name`*, subdirectory **`files/`**, ***`path/to/file`***.

  So, **`puppet:///sudo/sudoers`** would result in **`/var/lib/puppet/modules/`*`$environment`*`/sudo/files/sudoers`**.

# Troubleshooting Puppet

This section is about troubleshooting the manifests, the puppetmaster and the puppet.

## 10.1. Manifests

One commonly made error is of course a syntax error. Other errors include:

- Not realizing a *virtual resource*

  For more on virtual resources, see also: *Section 9.3, "Virtual Resources"*

- Depending on resources not defined
- Wrong or missing parameters to a *defined type*
- Circular or missing dependencies or notifications

### Manifest syntax error

Should you have a syntax error in a manifest, and have it go all the way through to the puppetmaster, the puppetmaster will not automatically pick up the new manifest, unless the puppetmaster is restarted -in which case the first puppet run will make it complain about the inability to parse all the manifests. Symptoms include your changes not being applied to the puppets, and error messages in **`/var/log/messages`**

### Checking for syntax errors

Checking for syntax errors is a good habit. Puppet has a utility called **puppet** that can help you find out whether your manifests have syntax errors. In a directory tree full of **`.pp`** files, you could run:

```
$ puppet --noop --parseonly `find -name "*.pp"`
```

### Not realizing virtual resources

Virtual resources like normal resources can only be defined once, and need to be realized before they are included in the manifest sent to the client. You can realize a virtual resource multiple times. Make sure the virtual resources that you use are realized in all the appropriate places.

### Depending on resources not defined

Like in the *YP Client Example*, resources can depend on other resources, so that one of the resources, in this case `Package["ypbind"]`, is applied first, `File["/etc/yp.conf"]` is applied second and `Service["ypbind"]` is applied last. Note that the depending resources are only applied if the dependent resource is applied succesfully.

### Wrong or missing parameters to a defined type

A *defined type* is a custom type or maybe an aggregation of types, you define yourself from within a manifest, taking a few parameters and creating resources from *native type*. You can take a look at an *Section C.1, "Example Defined Type"* to see how this works. A common error is changing the defined type with regards to the parameters it takes, but not updating the use of these defined types.

### Circular or missing dependencies or notifications

If `File["/etc/yp.conf"]` requires `Package["ypbind"]`, and `Package["ypbind"]` requires `File["/etc/yp.conf"]`, neither of both are going to succeed waiting for the other resource to be applied succesfully first.

The other way around is when `Service["ypbind"]` does not depend on `File["/etc/yp.conf"]`, the **ypbind** service may be started before the correct settings are applied.

If `File["/etc/yp.conf"]` changes, the **ypbind** service needs to be reloaded or restarted to get the new settings applied to the service operation. If the `notify => Service["ypbind"]` is missing in `File["/etc/yp.conf"]` though, this will not just happen.

## 10.2. The Puppetmaster

The puppetmaster service is configured using **`/etc/sysconfig/puppetmaster`**, which is also where you can specify what the log destination is, and whether the puppetmaster service should run in verbose or debug mode. Both would be using the `PUPPETMASTER_EXTRAOPTS` variable.

### 10.2.1. Debugging The Puppetmaster

Running the puppetmaster in debug mode generates a lot of output. You may want to redirect the output to a file and read the file to prevent the available scrollback buffer from filling up too fast. To run the puppetmaster in debug mode, stop the puppetmaster service and type:

```
# puppetmasterd --debug --no-daemonize [--servertype mongrel --masterport
 8141]
```

Or run the service in verbose or debug mode by editing **`/etc/sysconfig/puppetmaster`** and adding `--verbose` or `--debug` to `PUPPETMASTER_EXTRAOPTS`.

## 10.3. The Puppet

Debugging the puppet can be tricky, because it is the puppetmaster that parses the manifests first, and so if the puppetmaster fails parsing the manifests, the puppet client won't get any changes to it's manifests. If files or templates have changed on the fileserver, these will obviously be changed on the puppet as well, but if a template uses a new variable from the manifest, the template will fail to parse and the error message will occur on the puppetmaster.

### 10.3.1. Debugging The Puppet

To get an incling about what went wrong, the output of

```
# puppet --verbose --debug --noop
```

This will run puppet without actually doing the work. Look for things like a 503 error (HTTP permission denied) of a (fatal) error like "class not found" or a warning like "duplicate resource".

# Other Things To Do With Puppet

There's way more you can do with puppet. Actually there's so much you can do with puppet it does not fit in a workshop type of course.

## 11.1. Store Configurations In A Database

A database can be configured to store the configuration distributed by the puppetmaster, and applied by the puppet. This is optional, and creates some overhead to the original purpose of configuring the puppets, but provides the opportunity to create overviews of applied classes to nodes, and a complete inventory of *facts* for all nodes. similarities / exemptions.

Regardless of which of the following methods is chosen, they have one thing in common and that is that only the database has to be created. Puppet will create the tables with the correct schema's for you. Only the postgresql example is shown, for the other databases, please consult the respective manual on how to create a database.

### 11.1.1. SQLite3

SQLite(3) is a file based, light SQL database which is suitable for small databases. Depending on the size of the organization or priority you give to storing configs, generally speaking using SQLite(3) is not very suitable. In addition, SQLite3 isn't easily queried either manually or automatically. To setup SQLite3, provide the following settings in **`/etc/puppet/puppet.conf`**:

```
[puppetmasterd]
    reports = store[,tagmail,rrdgraph]
    storeconfigs = true
    dbadapter = sqlite3
    dblocation = /var/lib/puppet/storeconfigs.sqlite
```

### 11.1.2. MySQL

MySQL of course is much more scalable, and you can query it manually or automatically. A simple query can give you all webservers in the organization:

```
$ mysql -p puppet -e 'SELECT hosts.name
    FROM resources INNER JOIN hosts
        ON resources.host_id = hosts.id
    WHERE resources.title = "webserver"
    GROUP BY name;'

+--------------------------+
| name                     |
+--------------------------+
| app1.genomicscenter.nl   |
| elwood.kanarip.com       |
| master.puppetmanaged.org |
| open.the-cave-of-steef.nl |
| pinky.kanarip.com        |
| server.ogd.nl            |
```

```
| vito.kanarip.com          |
+---------------------------+
7 rows in set (0.02 sec)
```

To configure storing the configurations, use the following settings in **/etc/puppet/puppet.conf**:

```
[puppetmasterd]
    storeconfigs = true
    dbadapter = mysql
    dbserver = 127.0.0.1
    dbuser = puppetuser
    dbpassword = password
    [dbsocket = /var/lib/mysql/mysql.sock]
```

And create the database:

```
# mysql -p -e "CREATE DATABASE puppetdb;"
# mysql -p -e "GRANT ALL PRIVILEGES on puppetdb.* to puppetuser@localhost
 identified by 'password';"
```

### 11.1.3. PostgreSQL

PostgreSQL is even more scalable then MySQL as it's multi-master, but most importantly you may already have PostgreSQL running in your organization, and may just want to add puppet's configuration store to that infrastructure.

To add the configuration store to PostgreSQL, use the following settings in **/etc/puppet/puppet.conf**:

```
[puppetmasterd]
    storeconfigs = true
    dbadapter = postgresql
    dbuser = puppetuser
    dbpassword = password
    dbserver = localhost
    dbname = puppetdb
```

And create the database:

```
# su - postgres
$ psql
postgres=# create database puppetdb;
CREATE DATABASE
postgres=# create user puppetuser with unencrypted password 'password';
CREATE ROLE
postgres=# grant create on database puppetdb to puppetuser;
```

## 11.2. Tweaking Reporting

Reports can be sent out to various email addresses as a notification on errors or changes applied to the systems. This not only helps in keeping track of changes being applied, but can also help you keep your manifests clean, attending to each error that may otherwise have passed by unnoticed.

To enable reporting, use the following settings in **`/etc/puppet/puppet.conf`**:

```
[main]
    tagmap = /etc/puppet/tagmail.conf

[puppetmasterd]
    reports = tagmail

[puppet]
    reportfrom = puppet-reports@domain.tld
```

This will cause the puppetmaster to look at **`/etc/puppet/tagmail.conf`**, which maps tags to email addresses that need to be notified on changes applied to systems tagged with the appropriate keywords.

### Using tags in your manifests

You can use tags in your manifests like so:

```
class ssh {
    tag("security")
    file { "...":
        (...)
    }
}
```

To have notifications on changes to systems including the ssh class sent to security-admins@domain.tld, use the following tagmap entry:

```
ssh: ssh-admins@domain.tld
security: security-admins@domain.tld
```

A catch-all can be specified to have notifications on all changes sent to an emailaddress or multiple email addresses:

```
all: system-admins@domain.tld
```

## 11.3. Writing Custom Facts

Writing custom facts, like writing custom types, functions or providers, is done in Ruby, using the interfaces and objects offered by Puppet.

## 11.4. Writing Custom Types

*http://reductivelabs.com/trac/puppet/wiki/CreatingCustomTypes*

## 11.5. Writing Custom Functions

*http://reductivelabs.com/trac/puppet/wiki/WritingYourOwnFunctions*

## 11.6. Writing Custom Providers

*http://reductivelabs.com/trac/puppet/wiki/ProviderDevelopment*

## 11.7. Storeconfigs, Reporting and Puppetview

Using Puppet's storeconfig capability and standard reporting, in combination with rrdgraphing, you can use applications such as **Puppetview** to create nice overviews of results of your configuration management as applicable to your systems.

# Best Practices

Using Puppet starting in a small(er) scale, scaling up to more and more systems in your organization to become managed with puppet, you might encounter some challenges and to prevent you ending up with a giant mess of manifests, files, templates, needing to search (with `grep` and `find` for example) where it is exactly you need to configure the next change, or having to use configuration items that need to be changed as soon as your organization introduces the next generation operating systems or distribution version, here's a set of *Best Practices*, or actually Tips & Tricks.

## 12.1. Setting $os and $osver

The `$operatingsystem` and `$operatingsystemrelease`, `$operatingsystemversion` or `$lsbdistrelease` variables relate directly to specific operating system resources, configuration file locations, package names and most importantly, settings you can or cannot use in the configuration files for specific applications[1]. Using the `$operatingsystem`, `$operatingsystemrelease`, `$operatingsystemversion` and/or `$lsbdistrelease` variables (set from Facter) help you determine the Operating System (Distribution) and Version.

However, some distributions have Facter set `$operatingsystemrelease`, while others set `$lsbdistrelease`. If these variables need to be used in manifests, you really do not want to use these both because in your manifests all you are interested in is the Operating System Version. Using the following snippet of code in **`site.pp`** enables you to use `$os` and `$osver` throughout your manifests.

```
# Get facts and give them a good, good name
$os = $operatingsystem

case $os {
    "Fedora", "CentOS", "RedHat": {
        $osver = $lsbdistrelease
    }
    "Debian", "SuSE', "OpenSuSE": {
        $osver = $operatingsystemrelease
    }
    "Darwin": {
        $osver = $operatingsystemrelease
    }
}
```

## 12.2. Using Multiple Sources

Because no system is exactly alike, it is advisable to formalize a method to allow exceptions, for instance:

```
#Allow exceptions
file { "/etc/httpd/conf/httpd.conf":
```

[1] Enterprise Linux 3 distributions for example have OpenSSH Server versions that do not use, and are incompatible with, the `UsePAM` setting in **`/etc/ssh/sshd_config`**. In these cases, you would want the configuration file for EL-3 to come from a different location then for the EL-4 and EL-5 distributions that do take `UsePAM` because of the OpenSSH Server package versions they ship.

```
    source => [
        "puppet://$server/webserver/$hostname/httpd.conf",
        "puppet://$server/webserver/httpd.conf.$hostname",
        "puppet://$server/webserver/httpd.conf"
    ],
    require => Package["httpd"],
    notify => Service["httpd"]
}
```

This allow three ways for a webserver to get the resource `/etc/httpd/conf/httpd.conf`.

## 12.3. Group Profiles

Grouping *profiles* that define certain resources to be applied to the systems you manage can help you in giving and keeping structure in the ever-growing configuration tree you use.

We know of organizations that run the following deterministic set of profiles:

- **Classes**
  Organization specific classes not eligible to become a module. Note that these are mostly very simple classes, managing a single resource, or merely defining a set of variables per-class to be used throughout the manifests.

  Using classes to manage multiple resources is strongly discouraged as these are commonly eligible to become modules. However, when using a module from upstream changing that module to fit your needs introduces the risk of conflicts when trying to update the module from upstream. Therefor, classes can be used to override and/or extend classes from modules.

  Classes are stored in **`classes/*.pp`**, where the name of the actual file represents the class defined in the file.

- **Domains**
  Domains are most commonly used within organizations that manage branch offices, or merge with other organizations, or Puppet setups that manage more then one organization at a time.

  puppetmanaged.org runs multiple organizations with each organization maintaining their own domain specific configuration tree and manifests.

  CustomerA has 3 branch offices and uses domains to define branch office specific settings.

  Defined domains should go in **`domains/*.pp`** where the actual filename is the most descriptive common denominator between all systems in a domain.

- **Groups**
  Grouping systems allow you to control a lot more with a lot less. Less is more, sort of. Groups can include *servers*, *desktops*, *laptops*.

  Given a group of servers you can imagine SELinux needs to be enabled on all of these, and the firewall needs to be up and running. Create a group servers in **`groups/server.pp`** and include the class defined or inherit the node used.

- **Modules**
  Modules that need settings or files that need to be pulled (updated) from upstream. For older versions of Puppet (< 0.24.4), you can state "imports" here as well.

  Modules allow you to group in one subdirectory all the files needed for the module. This means that upstream can still be the repository that all other files come from, or a completely different repository (perhaps from an opensource contributer)

- **Nodes**
  The nodes you manage, one per file. These should be be actual nodes, not abstraction levels where you inherit from.

  This allows a per host configuration. This allow for exceptions/additions to the more general groups a node can be part of. Of course this also allows for definition of a node that is not part of any group.

- **Services**
  This is kind of on a par with grouping, but the design concept behind it is different. Instead of grouping together the same kind of hardwareclass, group together the common function these nodes have.

  Given a cluster of webservers and another cluster of mailservers, it should be obvious what the benefit is. You can change a website instantly on all websever or add an additional maildomain all the nodes of a mailcluster, all in one change.

- **Utils**
  para

  explanation

- **Users**
  para

  explanation

# Part I. Appendices

# Appendix A. Puppet Terminology

- **class**
  A class is a collection of resources applied to a node with a single include statement. It groups together a comprehensible set of resources. A class *ypclient* would manage the `File["/etc/nsswitch.conf"]`, `File["/etc/yp.conf"]`, `Package["ypbind"]`, and `Service["ypbind"]` resources.

- **defined type**
  defined type

- **fact**
  A client-side generated aspect of the node the puppet client runs on. Example facts are the amount of available memory, the hostname, the fully qualified domain name, the operating system (version).

- **manifest**
  The collection of classes, modules and resources that the *puppetmaster* uses to distribute the appropriate configuration to a *puppet*.

- **module**
  module

- **native type**
  A type provided by puppet

- **node**
  The client, a node, is an operating system instance running the puppet client application. This can be a regular operating system running directly on top of actual hardware, a virtual guest as well as a virtual host.

- **puppet**
  The client, a node, runs the **puppetd** daemon or service, and is referred to as the *puppet*

- **puppetmaster**
  The puppetmaster is the node that runs the server-side application to a puppet setup.

- **resource**
  A resource is an instantiated *type*

- **system resource**
  A system resource is a resource available on the node whether it is managed by puppet or not. Unlike what is otherwise understood by system resources, the puppet definition of system resources does not so much refer to resources like CPU or memory, but rather to whether or not a package is installed or what version of said package, or the $osversion, and so on and so forth.

- **virtual resource**
  A virtual resource is like a normal *resource*, but it is not sent to the puppet (e.g. not part of the manifest sent to the client) until it is realized. Virtual resources may be realized multiple times, and are sent to the puppet only once.

  *See Also*: *Section 9.3, “Virtual Resources”*

- **type**
  definition

# Appendix B. Example SSL Frontend Reverse Proxy Load Balancer Configuration

```
<ifModule !mod_proxy.c>
    LoadModule proxy_module modules/mod_proxy.so
</IfModule>

<IfModule !mod_proxy_http.c>
    LoadModule proxy_http_module modules/mod_proxy_http.so
</IfModule>

<IfModule !mod_proxy_balancer.c>
    LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
</IfModule>

<IfModule !mod_headers.c>
    LoadModule headers_module modules/mod_headers.so
</IfModule>

<IfModule !mod_ssl.c>
    LoadModule ssl_module modules/mod_ssl.so
</IfModule>

<IfModule !mod_authz_host.c>
    LoadModule authz_host_module modules/mod_authz_host.so
</IfModule>

<IfModule !mod_log_config.c>
    LoadModule log_config_module modules/mod_log_config.so
</IfModule>

<Directory />
    Options FollowSymLinks
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>

Listen 8140
NameVirtualHost *:8140

<Proxy balancer://master.puppetmanaged.org>
  BalancerMember http://127.0.0.1:8141 keepalive=on retry=30
</Proxy>
```

```
<VirtualHost *:8140>
    ServerName master.puppetmanaged.org
    SSLEngine on
    SSLCipherSuite SSLv2:-LOW:-EXPORT:RC4+RSA
    SSLCertificateFile      /var/lib/puppet/ssl/
certs/master.puppetmanaged.org.pem
    SSLCertificateKeyFile   /var/lib/puppet/ssl/
private_keys/master.puppetmanaged.org.pem
    SSLCertificateChainFile /var/lib/puppet/ssl/ca/ca_crt.pem
    SSLCACertificateFile    /var/lib/puppet/ssl/ca/ca_crt.pem
    SSLVerifyClient optional
    SSLVerifyDepth  1
    SSLOptions +StdEnvVars

    # The following client headers allow the same configuration to work
 with Pound.
    RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

    <Location />
        SetHandler balancer-manager
        Order allow,deny
        Allow from all
    </Location>

    ProxyPass / balancer://master.puppetmanaged.org:8140/ timeout=180
    ProxyPassReverse / balancer://master.puppetmanaged.org:8140/
    ProxyPreserveHost on
    SetEnv force-proxy-request-1.0 1
    SetEnv proxy-nokeepalive 1

    ErrorLog  logs/master.puppetmanaged.org-balancer-error_log
    CustomLog logs/master.puppetmanaged.org-balancer-access_log combined
    CustomLog logs/master.puppetmanaged.org-balancer-ssl_request_log "%t %h
 \
                                    %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\"
 %b"
</VirtualHost>
```

# Appendix C. Examples

para

## C.1. Example Defined Type

A defined type is a custom type you can define yourself. This can be a shortcut to a collection of resources given a set of parameters, or just a single resource, as shown in the following examples.

### Defined type for a single resource

This is an example defined type that creates a single resource, with just one parameter controlling the source of where the file comes from.

```
define yum::repository ($enable = true) {
    file { "/etc/yum/repos.d/$name.repo":
        mode => 644,
        owner => "root",
        group => "root",
        backup => false,
        links => follow,
        source => $enable ? {
            true => "puppet:///yum/$os/$osver/repos/$name.repo",
            default => "puppet:///yum/$os/$osver/repos/$name.repo.disabled"
        }
    }
}
```

You can use this defined type to manage the files in **`/etc/yum/repos.d/`**, and thus the repositories available to the YUM package manager, by calling (for example):

```
yum::repository { "custom":
    enable => true
}
```

This will create a `File["/etc/yum/repos.d/custom.repo"]` and grab if from a location on the puppetmaster's fileserver where it has `enabled=1` in the repository configuration file. Should `$enable` have been set to false in the `Yum::Repository["custom"]` resource, then the file would have been grabbed from the location on the puppetmaster's fileserver where the contents would have `enabled=0`.

### Multiple resources in a defined type

This is an example

```
define yum::plugin($enable = false) {
    file { "/etc/yum/pluginconf.d/$name.conf":
        ensure => $enable ? {
            true => file,
            default => absent
```

```
        },
        owner => "root",
        group => "root",
        mode => 644,
        source => "puppet:///yum/plugins/$name.conf"
    }

    package { "yum-$name":
        ensure => $enable ? {
            true => installed,
            default => absent
        }
    }

    case $name {
        "versionlock": {
            file { "/etc/yum/pluginconf.d/versionlock.list":
                source => "puppet:///yum/plugins/versionlock.list"
            }
        }
    }
}
```

# Appendix D. GIT Commit Hooks

The following GIT commit hooks allow you to check the manifests you commit before they are actually committed (pre-commit), and sync the puppetmasster after the changes are applied to the repo (post-commit), including posting the changes or diff to a mailing list.

### pre-commit

There is no valid pre-commit hook for GIT repositories yet.

### post-commit

```
#!/bin/sh
#
# An hook script to mail out commit update information.
# Called by git-receive-pack with arguments: refname sha1-old sha1-new
#
# To enable this hook:
# (1) change the recipient e-mail address
# (2) make this file executable by "chmod +x update".
#

project=$(cat $GIT_DIR/description)
[ -f $GIT_DIR/commit-list ] && \
        recipients=$(cat $GIT_DIR/commit-list)
[ -n "$recipients" ] || exit 0

ref_type=$(git cat-file -t "$3")

# Only allow annotated tags in a shared repo
# Remove this code to treat dumb tags the same as everything else
# case "$1","$ref_type" in
# refs/tags/*,commit)
#   echo "*** Un-annotated tags are not allowed in this repo" >&2
#   echo "*** Use 'git tag [ -a | -s ]' for tags you want to propagate."
#   exit 1;;
# refs/tags/*,tag)
#   echo "### Pushing version '${1##refs/tags/}' to the masses" >&2
#   # recipients="release-announce@somwehere.com
#   announce@somewhereelse.com"
#   ;;
# esac

# set this  to 'cat' to get a very detailed listing.
# short only kicks in when an annotated tag is added
short='git shortlog'

# see 'date --help' for info on how to write this
# The default is a human-readable iso8601-like format with minute
# precision ('2006-01-25 15:58 +0100' for example)
```

```
date_format="%F %R %z"

# Set to the number of pathname components you want in the subject line
# to indicate which components of a project changed.
num_path_components=2

# Set subject
(if expr "$2" : '0*$' >/dev/null ; then
    subject="Changes to '${1##refs/heads/}'"
    echo "Subject: $subject"
else
    base=$(git-merge-base "$2" "$3")
    subject=$(git-diff-tree -r --name-only "$base" "$3" |
              cut -d/ -f-$num_path_components |
              sort -u |
              xargs echo -n)

    commits=$(git-rev-list "$3" "^$base" | wc -l)

    if [ "$commits" -ne 1 ] ; then
        subject="$commits commits - $subject"
    fi

    branch="${1##refs/heads/}"

    if [ "$branch" != "master" ] ; then
        subject="Branch '$branch' - $subject"
    fi

    echo "Subject: $subject"
fi

echo "To: $recipients"
echo "X-Project: $project"

module=$(basename `readlink -f $GIT_DIR`)

echo "X-Git-Module: $module"
echo ""

if expr "$2" : '0*$' >/dev/null
then
    # new ref
    case "$1" in
        refs/tags/*)
            # a pushed and annotated tag (usually) means a new version
            tag="${1##refs/tags/}"
            if [ "$ref_type" = tag ]; then
                eval $(git cat-file tag $3 | \
                    sed -n '4s/tagger \([^>]*>\)[^0-9]*\([0-9]*\).*/
tagger="\1" ts="\2"/p')
```

```
                date=$(date --date="1970-01-01 00:00:00 $ts seconds"
 +"$date_format")
                echo "Tag '$tag' created by $tagger at $date"
                git cat-file tag $3 | sed -n '5,$p'
                echo
            fi

            prev=$(git describe "$3^" | sed 's/-g.*//')
            # the first tag in a repo will yield no $prev
            if [ -z "$prev" ]; then
                echo "Changes since the dawn of time:"
                git rev-list --pretty $3 | $short
            else
                echo "Changes since $prev:"
                git rev-list --pretty $prev..$3 | $short
                echo ---
                git diff $prev..$3 | diffstat -p1
                echo ---
            fi
        ;;

        refs/heads/*)
            branch="${1##refs/heads/}"
            echo "New branch '$branch' available with the following
 commits:"
            git-rev-list --pretty "$3" $(git-rev-parse --not --all)
        ;;
    esac
else
    case "$base" in
    "$2")
        git diff "$3" "^$base" | diffstat -p1
        echo
        echo "New commits:"
        ;;
    *)
        echo "Rebased ref, commits from common ancestor:"
        ;;
    esac
    git-rev-list "$3" "^$base" | while read rev; do git-show $rev; echo "";
 echo "";  done
fi) | /usr/local/bin/send-unicode-email.py $recipients
exit 0
```

**Note**

The post-commit GIT hook requires **/usr/local/bin/send-unicode-email.py**

# Appendix E. SVN Commit Hooks

The following SVN commit hooks allow you to check the manifests you commit before they are actually committed (pre-commit), and sync the puppetmaster after the changes are applied to the repo (post-commit), including posting the changes or diff to a mailing list.

### pre-commit

```
#!/bin/sh
# SVN pre-commit hook to check Puppet syntax for .pp files
# Modified from http://mail.madstop.com/pipermail/puppet-users/2007-
March/002034.html
REPOS="$1"
TXN="$2"
tmpfile=`mktemp`
export HOME=/tmp
SVNLOOK=/usr/bin/svnlook
$SVNLOOK changed -t "$TXN" "$REPOS" | awk '{print $2}' | grep '\.pp$' |
 while read line
do
    $SVNLOOK cat -t "$TXN" "$REPOS" "$line" > $tmpfile
    if [ $? -ne 0 ]; then
        echo "Warning: Failed to checkout $line" >&2
    fi
    puppet --color=false --confdir=/tmp --vardir=/tmp --parseonly --
ignoreimport $tmpfile >&2
    if [ $? -ne 0 ]; then
        echo "Puppet syntax error in $line." >&2
        exit 2
    fi
done
res=$?
rm -f $tmpfile
if [ $res -ne 0 ]; then
    exit $res
fi
```

### post-commit

```
#!/bin/sh
REPOS="$1"
REV="$2"
PATH=/usr/bin:/bin
PROJECT=puppet
TO=to@domain.tld
FROM=from@domain.tld

svn up /etc/puppet/
svn up /var/lib/puppet/manifests/development/
```

```
svn up /var/lib/puppet/manifests/testing/
svn up /var/lib/puppet/manifests/production/
svn up /var/lib/puppet/modules/development/
svn up /var/lib/puppet/modules/testing/
svn up /var/lib/puppet/modules/production/

# Note that --from is optional and if omitted the from address
# will be $COMMITTER@$hostname
#
# If no --from is used, the committer may need to be subscribed
# to the mailing list used in order for this message to be
# accepted to the mailing list
#

svnnotify --repos-path "$REPOS" \
          --revision "$REV" \
          --smtp localhost \
          --svnlook /usr/bin/svnlook \
          --to $TO \
          --from $FROM \
          --subject-prefix "Puppet SVN Commit: " \
          --with-diff --subject-cx --handler HTML::ColorDiff
```

# Appendix F. Module Conventions

This section lists a number of conventions used by all modules on *http://puppetmanaged.org*, that you can use (f desireable) and modify as needed.

## F.1. Code Layout

A consistent layout of the code is important within a collaborative environment. This section describes a number of conventions used in the modules on puppetmanaged.org.

### F.1.1. Indentation

- **4 spaces to a tab**

  No tabs should be used, and instead 4 space characters should be inserted to increase the indentation level. This applies to the manifests, custom facts, the configuration files (if compatible) as well as the DocBook documentation.

- **Aggregated resource definitions**

  Aggregated resource definitions, such as `file { [ "foo", "bar" ]: ensure => absent }` are indented like this:

```
file { [
        "foo",
        "bar"
    ]:
    ensure => absent
}
```

## F.2. Sources

The location a file is sourced from is subject to the following conditions:

1. A file should first be sourced from a domain-specific configuration tree as to allow domain administrators to maintain their own files. The exact location again is subject to the following conditions:
   1.a. If the file is part of a directory structure that is going to be pulled from the source recursively, and the configuration files in that directory structure is host specific, files for such a host go into a sub-directory `$hostname`.

   1.b. If the file is is a single host-specific file to be pulled from the source, the file goes into the same directoryas the original file appended with `.$hostname`.

2. A file should then be sourced from the common, shared **`files/`** file server share and is organization specific, but not necessarily domain-specific. The **`files/`** file server share allows an organization to maintain the configuration in a single source controlled tree. Again, the following conditions apply to a file location:
   2.a. If the file is part of a directory structure that is going to be pulled from the source recursively, and the configuration files in that directory structure is host specific, files for such a host go into a sub-directory `$hostname`.

2.b. If the file is is a single host-specific file to be pulled from the source, the file goes into the same directoryas the original file appended with `.$hostname`.

3. The source is the default configuration available with the puppet module.

Such a set of conditions can maybe be shown best in an example configuration, in this case a single file:

```
class sudo {
    file { "/etc/sudoers":
        source => [
            # Organization-wide, host specific, single file
            "puppet:///files/sudo/sudoers.$hostname",

            # Organization-wide
            "puppet:///files/sudo/sudoers",

            # Module default
            "puppet:///sudo/sudoers"
        ]
    }
}
```

If, however, a host is configured to recursively pull a directory from source, this is a better approach:

```
class foo {
    file { "/path/to/foo/":
        source => [
            # Organization-wide, host specific, directory tree
            "puppet:///files/foo/$hostname/",

            # Organization-wide
            "puppet:///files/foo/common/",

            # Module default
            "puppet:///foo/common/"
        ],
        recurse => true
    }
}
```

For common files shared between the common and host specific tree you use symbolic links from the host specific tree to the common tree. If, however, there is a million files in the common directory tree with thousands of changes every day, which makes using symbolic links not particularly sustainable, you can also specify an additional File resource in the manifest applied to the host and manage the host-specific file separate from the recursively managed directory. An example:

```
file { "/path/to/foo/":
    source => "puppet:///foo/common/",
    recurse => true
```

```
}

file { "/path/to/foo/bar":
    source => "puppet:///foo/$hostname/bar"
}
```

For host specific files and directory trees shared amongst a group of hosts, you use a category specific file name or directory name appendix (such as `.admins-only`) and use symbolic links between the host specific file or directory and the category specific file or directory.

### F.2.1. Scalability Issues

Suppose you are a large ISP (example-isp.com) running a million websites which may or may not be managed by puppet. Putting all virtualhost specific configuration files in your domain specific tree under **`webserver/sites/`** doesn't scale very well because of the large amount of files in one single directory which could be inconvenient when trying to edit those files. We've set up a domain-specific tree working around that problem by adding a **`webserver/sites.d/`** tree dividing the files into a hierarchy with makes tab-completion and finding the files needing to be edited somewhat easier.

The **`webserver/`** tree in GIT => *http://git.puppetmanaged.org/?p=domains/example-isp.com/.git;a=tree;f=webserver*

## F.3. Module Tree Layout

Like described on *Puppet's wiki*[1], a module tree looks as follows:

```
MODULE_PATH
    `- module_name/
        `- README
        `- depends/
        `- files/
        `- manifests/
        `- plugins/
            `- facter/
            `- puppet/type/
        `- templates/
```

See also: *http://reductivelabs.com/trac/puppet/wiki/ModuleOrganisation*

This is a segmentedlist:

Module Layout

| Directory | Description |
|---|---|
| `MODULE_PATH` | The path that holds the modules (see `modulepath` in **`/etc/puppet/puppet.conf`**) |
| **`` `- module_name/depends/ ``** | Dependencies for the module |
| **`` `- module_name/files/ ``** | Files distributed with the module. Files in this directory can be sourced by using the `puppet:///module_name/` URI |

[1] http://reductivelabs.com/trac/puppet/wiki

| Directory | Description |
| --- | --- |
| **`- module_name/manifests/** | The manifests directory for this module. Should at least hold a file called **init.pp** |
| **`- module_name/plugins/facter/** | Holds custom facts for this module |
| **`- module_name/plugins/puppet/type/** | Holds custom puppet types for this module |

Additionally, puppetmanaged.org modules define a tree layout for some of the files needed by these modules. For example, Red Hat Enterprise Linux 3 cannot have the `UsePAM` configuration directive in **/etc/ssh/sshd_config**. For the SSH module, the **files/** directory has the following layout:

```
ssh/files
    `- CentOS -> RedHat
    `- RedHat/
        `- 3/
            `- sshd_config
    `- sshd_config
```

whereas the module sources the **/etc/ssh/sshd_config** as follows:

```
file { "/etc/ssh/sshd_config":
    owner => "root",
    group => "root",
    mode => 600,
    replace => true,
    source => [
        # Organization specific, OS version specific, host specific
        "puppet:///files/ssh/$os/$osver/sshd_config.$hostname",
        # Organization specific, OS version specific
        "puppet:///files/ssh/$os/$osver/sshd_config",

        # Organization specific, OS specific, host specific
        "puppet:///files/ssh/$os/sshd_config.$hostname",
        # Organization specific, OS specific
        "puppet:///files/ssh/$os/sshd_config",

        # Organization specific, host specific
        "puppet:///files/ssh/sshd_config.$hostname",
        # Organization specific
        "puppet:///files/ssh/sshd_config",

        # Module default, OS version specific
        "puppet:///ssh/$os/$osver/etc/ssh/sshd_config",
        # Module default, OS specific
        "puppet:///ssh/$os/etc/ssh/sshd_config",
        # Module default
        "puppet:///ssh/etc/ssh/sshd_config"
    ],
    notify => Service["sshd"],
```

```
    require => Package["openssh-server"]
}
```

## F.4. File And Directory Paths

To be able to clearly distinct files from directories (and vice-versa of course), the modules use the following naming convention:

- Directory names end with a forward slash.
- Filenames do not (however obvious).

And here's why it's important. There's a few utilities that do not work if a trailing slash to a fully qualified directory name is omitted. One of these situations is shown as an example (and is not the actual code in the git module):

```
git::pull { [
        "foo",
        "bar"
    ]:
    base_source => "/srv/git",
    localtree => "/var/lib/puppet/modules/development"
}

define git::pull($localtree, $base_source, $branch = 'master') {
    exec { "git_pull_$name":
        command => "git pull ${base_source}${name} $branch",
        cwd => "${localtree}${name}"
    }
}
```

where the cwd has become **`/var/lib/puppet/modules/developmentfoo`** instead of the intended **`/var/lib/puppet/modules/development/foo/`**. Also, the `$base_source` in this example is, unintentionally, **`/srv/gitfoo`**.

If the modules do not use the standard directory naming schema as set forth in this section, some modules might start omitting the forward slash (between `${localtree}` and `$name`, or between `${base_source}` and `${name}`), and others might not, which in the end leads to confusion and malfunctioning modules.

# Appendix G. Revision History

Revision 1.0