Puppet Enterprise Deployment Guide

(Generated on July 01, 2013, from git revision 46784ac1656bd7b57fcfb51d0865ec7ff65533d9)

First Steps: Setting Up Your Environment

First Steps: Setting Up Your Environment

Once you've downloaded Puppet Enterprise, completed installation, and verified that all the parts are talking to each other, there are a few things you'll want to do to set up your working environment.

Puppet's manifests, as well as any modules, ENC's, etc. you are writing or modifying, should be treated like any other code under development, which is to say they should be version controlled, tested prior to release, and backed up frequently. Below we'll give you some suggestions for getting that done.

Version Control

Typically, a Puppet site consists of:

- A collection of modules, which contain classes that manage chunks of system functionality.
- A central manifest (typically, `site.pp`), which assigns classes from modules to the nodes at the site.
- The PE console, which also assigns classes from modules to nodes. This overlaps with `site.pp`; the preferences of your site's staff should guide you when choosing which to use.
- Hierarchical data in Hiera, which can be used by module classes when they're assigned to a node.

All of these (except the console) should be edited in version control and then deliberately deployed to the puppet master. Editing your manifests and data "live," directly on the puppet master, is a dangerous habit. You can lose functioning code or, worse, your half-written manifest could get published to your agent nodes prematurely. To avoid this, we strongly recommend you use a version control system such as Git or Subversion to control development of your manifests. Generally speaking, we recommend Git and GitHub, but really any version control system will work.

GIT RESOURCES

A course in using Git is well beyond the scope of this guide, but there are many online tutorials, guides and other resources. Some of our favorites are: Pro Git, Git+Immersion and the tutorials on GitHub. If you prefer video, the author of Pro Git, Scott Chacon, has an excellent presentation. There are also some applications that wrap Git in a GUI (e.g., GitX) if you feel more comfortable working that way. The Git site has links to GUI tools for various platforms as well as tons of information on how to set up and use the tool.

GIT WORKFLOWS AND YOUR INFRASTRUCTURE

While there are many ways to map Git workflows to infrastructure, one of the most popular patterns is to set up Git branches to map to puppet environments. For example, you can set up Git branches to correspond to the development, testing, and production environments favored by many

companies' workflows. You can even set things up to create environments dynamically as workflows change. For a detailed discussion and examples of this, refer to this blog post on git workflows and puppet environments.

If you are already familiar with Git and have some workflows and branch structures in place already, you can probably just bring your Puppet manifests into your usual controls. There's no need to reinvent the wheel. If you're setting up version control for the first time, and the above example using environments is too involved, there is a good overview of a fairly basic Puppet/Git workflow available here. Please note that the paths in this example refer to open source Puppet - modules in Puppet Enterprise are located in `/etc/puppetlabs/puppet/modules`.

Testing

There are several resources you can turn to in order to test your manifests and modules. At this stage of the game, we don't mean testing in the formal sense of unit tests, CI tests, etc. We simply mean making reality checks to make sure things are working in the way you want. So, first you'll want to pick out the nodes you're going to automate initially. Generally speaking, when you start to deploy PE, you'll want to begin with low-risk nodes that are not mission critical. There are few approaches you can take in selecting nodes.

- One approach is to select a sub-set of nodes in, say, your development pool (as opposed to production servers). Identify configurations and services those nodes have in common and automate those things one by one.
- Another approach is to spin up a new machine that is a generic version of some group of servers in your infrastructure (again, choose things at first that aren't mission critical). Gradually automate more and more of its configuration, checking as you go that it is working correctly. Gradually automate the configuration of services and applications, migrating data from existing servers as needed. Once you're satisfied everything is working correctly, you can cut over to the newly automated machine and go live with it. You can use some (or all) of the classes, manifests and modules you used for this machine as the groundwork for automating other categories of nodes.
- A third approach can actually let you get two things done. If you're like most sysadmins, you have a backlog of things people have asked you for that are not for essential purposes. For example, a team has asked you for a server to test some new tool on, or to try out some new database/application/file server. Build up this server using PE.

This guide will give you some basic ideas for simple smoke testing of manifests and modules. Below we'll briefly list a few tools and resources which can help you further test your PE implementation. For a more in-depth discussion, take a look at this blog post on syntax checking and automated testing.

However, all of that said, when you are first deploying Puppet, your best bet is to rely heavily on modules you can get from the Puppet Forge. Forge modules, especially those from frequent

contributors or “Modules of the Week”, are proven solutions for the most common things sysadmins need to automate and they can save you a ton of time.

NOOP

Don’t overlook the power of the `--noop` flag. Noop lets you do a dry run of Puppet without actually doing anything. Instead, noop will sniff out catalog compilation errors and give you log output that shows what Puppet would have done to what, when. To run in noop mode, add the `--noop` flag when running `puppet agent` or `puppet apply`. You will see the results of your run in the Console and in stdout.

PUPPET PARSER

Puppet parser is another simple tool that lets you check the syntax of Puppet code. You can run it manually to check a given file, for example, `puppet parser validate site.pp`. This will check the manifest file for correct syntax and will return errors showing which line(s), if any, contain syntax errors such as missing curly braces or other typos and mistakes. You can also integrate syntax checking into your text editor, as explained in the previously referenced blog post.

PUPPET-LINT

Puppet lint is an easy to use style checker that runs through your manifests to ensure that they conform to Puppet Labs’s style guide. This standard is meant for developing reusable public code and may be more strict than you need, but much of it is good sense everywhere, and checking against it can guard you from many common mistakes. It’s available as a ruby gem. Install it by running `$ gem install puppet-lint`. You can learn more about puppet-lint from this blog post.

PRE/POST COMMIT HOOKS

Tools like Git or Subversion have hooks that allow you to run scripts before or after a commit. At minimum, you can use pre/post commit hooks to run puppet parser or puppet-lint automatically, but you will surely think of many more useful tests and other things to do with them. The Git site has good information to help get you started.

`RSPEC-PUPPET`

Writing tests with rspec is probably beyond the scope of this guide and not something you’ll want to tackle when you are first deploying PE. However, we mention it here so that when you move on to writing more complex manifests and/or your own modules you’re aware of the tool and what it can do to help you write the best puppet code you can. If you’re curious now, we recommend this tutorial to get you started.

Backing Up

We really don’t need to explain how or why you should back up your manifests and other files, but we will gently remind you to back up your SSL certificates along with everything else. In PE, SSL certificates exist under the `/etc/puppetlabs/puppet/ssl` directory.

Organize Your Users

Access to modules and manifests is controlled by your version control system and code deployment

processes. Access to the PE console is controlled by its own system of users and permissions.

Figure out who at your institution should have access to PE and with what privileges. You can add and define users manually in the console, or you can access an existing directory via LDAP, etc. For more information, see the PE manual page on user management. Initially, you will probably want to limit access to a select group, possibly a group of one.

Ad Hoc Change Tools

You'll want to have some kind of fall-back plan and tools for worst-case scenarios where something prevents you from controlling a machine with Puppet. In most cases, this simply means ensuring that you do nothing that would prevent you from SSHing into a given node and manually fixing whatever went wrong (e.g. with a DNS entry, a certificate, a root password, etc.). In addition to SSH, tools you might consider for this role include vintage things like Expect, Rsync, or even Telnet/FTP. As your skills with PE grow, you'll be able to consign these tools to the ashbin of history, but for worst case scenarios, just keep them in mind.

- Next: Puppetize Your Infrastructure: Beginning Automation

Puppetize Your Infrastructure: Beginning Automation

How Can Puppet Enterprise Help You?

The particulars of every infrastructure will vary, but all sysadmins have some needs in common. They want to be more productive and agile, they want to spend less time fixing recurring problems, and they want keen, timely insight into their infrastructure. In this chapter, we identify some specific needs and provide examples of ways to meet them by using Puppet Enterprise (PE) to automate stuff. If you've been an admin for any period of time, these wants should be all too familiar:

1. I want to learn how to use a tool that will make me a more efficient sysadmin, so I can go home earlier (or get to the pub, or my kids' soccer game, or the weekly gaming session, dragon boat race, naked knitting club meeting, etc.). If you're reading this, it's safe to assume the tool you want to learn is PE.
2. I want to know beyond a shadow of a doubt that I can always securely access any machine in my infrastructure to fix things, perform maintenance, or reconfigure something. I don't want to spend much time on this. I just want it to work.
3. I don't want to get repeated requests at all hours to fix simple mistakes. I want my infrastructure to care for itself, or at least be highly resistant to stupid.
4. I want fast, accurate reporting so that I know what's going on and can recover from issues quickly, with minimal downtime. I want my post-mortems to read: "I saw what was broken, I ran PE to restore it, the node was down for two minutes."

5. I want to be able to implement changes quickly and repeatably. Pushing out a new website (shopping cart app, WordPress template, customer database) should be trivial, fast, and reliable (see above re: getting to the pub, etc.).

Below, we'll run through some examples that will help a hypothetical admin meet all these needs by automating things in her infrastructure using PE (and with the assistance of pre-built modules from the Puppet Forge). We've tried to choose things that are low-risk but high-reward. That way you can try them out and can not only build your confidence working with PE, you can also actually start to get your infrastructure into a more automated, less on-fire condition relatively soon. Even if the specific services we discuss aren't directly applicable to your particular infrastructure, hopefully these examples will help you see methods you can apply to your specific situation and help you understand how and why PE can make you a better, less-stressed admin.

A (not-so) Fictional Scenario

Judy Argyle[1] is a sysadmin at a mid-sized flying car manufacturing concern, the National Car Company (NCC). The company is undergoing yet another periodic reorganization, and Judy has been tasked with administering the marketing department's servers. Previously, the department had been maintaining their own machines and so each server has been provisioned and configured in idiosyncratic ways. This has left an infrastructure that is time-consuming to manage, hard to survey, and very brittle. While you might think that flying cars sell themselves, the fact is, as management is constantly reminding Judy, the marketing department is very important and needs to be up and running 24/7/365.

This new responsibility is not exactly welcome. Judy already has 14 things that consistently set her hair on fire. In addition, NCC is rolling out a new model, the "Enterprise 1701," which means lots of people are making heavy demands on the IT infrastructure. She needs help. That help is Puppet Enterprise which, luckily, her manager has just approved implementing on a trial basis (since PE is free for up to 10 nodes, getting the approval wasn't all that hard). If PE can work for the marketing department, Judy will get the go-ahead to deploy it across the enterprise.

To start with, Judy downloads PE and, following the first two sections of this guide and the additional documentation it references, she gets a master, console and database support installed on a spare RHEL box and agents installed on the four servers that constitute the marketing department's infrastructure. After signing the various certificates and kicking off a puppet run on each agent, she can look at the console to confirm that all the agents are talking to the master. All of this feels pretty familiar to Judy since she took the Puppet Fundamentals Course a few weeks back.

Enough preamble, let's start automating infrastructure.

Three Things You Can Configuration Manage First

Poor Judy. As we know, the infrastructure of the marketing department is an oddball collection of

hardware that was set up ad hoc by a series of interns. Specifically, it consists of the following four servers:

1. An Ubuntu box named “web01” that runs Apache to serve up NCC’s main website
2. An old RHEL 5 FTP server named “ftp01” that provides NCC’s employees with access to marketing materials
3. A Debian server, “crm01”, running customer relationship software, SugarCRM
4. A Windows database server, “sql01”, running an SQL db containing customer info.

In addition, there is also the puppet master server, a RHEL box named “puppet.”

Fortunately for Judy, the Puppet Forge has modules that will run three of the four machines. For the fourth, the SugarCRM box (crm01.marketing), Judy decides she will write her own module.

> Note: In case you’re not familiar with modules, they are self-contained bundles of code and data that use Puppet to express a model for a given piece of infrastructure and interact with the puppet master to build your desired configuration state. A module consists of simple structures of folders and files that deliver manifests and extensions, like custom types or custom facts.) To learn more about the Puppet Forge and the module download and installation process, visit the installing modules page. Important: As of PE 2.8, the Puppet module tool is not compatible with Windows nodes, although modules themselves can be readily applied to Windows systems. In a typical Puppet environment, it is unlikely that a module would need to be installed directly on a Windows agent. However, if you encounter a corner case where installing a module directly on a Windows agent is unavoidable, you can do so by downloading tarballs. You may also need to go the tarball route to get Forge modules if local firewall rules prevent access to the Forge.

THING ONE: NTP

The first thing Judy wants to do is get reoriented and comfortable with PE (her Fundamentals class was several weeks and several beers ago). She needs to do something that will show her PE can quickly automate something, anything, across the marketing department’s infrastructure. She needs a real-world task that will prove the concept of automation, preferably something that even her boss can see and understand. She decides to start by managing NTP on all the marketing servers.

Managing NTP is useful in a Puppet Enterprise deployment because it reduces the possibility of certificate expiration. In order to provide the most robust security possible, Puppet’s certificates have short expiration times. This helps guard against things like replay attacks. Syncing time across all nodes reduces the possibility of certificates expiring unintentionally. In addition, having time synced across nodes will help ensure logs are correct and accurate, with events logged in the right order.

Okay, then. Judy starts by ssh’ing into her master and getting her selected module from the Forge by entering `sudo /opt/puppet/bin/puppet module install saz-ntp`. The module downloads and

installs automatically, as seen on the command line:

```
[jargyle@puppet ~]$ sudo /opt/puppet/bin/puppet module install saz-ntp
[sudo] password for jargyle:
Preparing to install into /etc/puppetlabs/puppet/modules ...
Downloading from http://forge.puppetlabs.com ...
Installing -- do not interrupt ...
/etc/puppetlabs/puppet/modules
└── saz-ntp (v2.0.3)
```

Next, Judy opens up the PE console in a supported browser and, using the "Class" tool in the sidebar at the lower left, she clicks the "Add class" button. She names the class "ntp" and clicks the "Create" button.

Now Judy needs to add the new class to all of her *nix-based nodes. Because there is a Windows machine present in the marketing department infrastructure, Judy can't use the default group, she needs to create a new group for all *nix nodes that excludes the Windows box. In the console, she clicks the "Add group" button in the sidebar's "Group" tool, she names the new group "nixes" and assigns the "ntp" class to it. Then, in the "Nodes" box, she starts typing the name of the Ubuntu webserver, "web01", PE autocompletes her entry and adds it to the list. She repeats this for each *nix node, including the puppet master. Then she clicks the "Create" button to add the new group.

> Note: Every node with a puppet agent will automatically become a member of the default group after it has completed its first puppet run. In a mixed Windows/*nix/OSX environment, the default group is mainly useful for Live Management tasks and across-the-infrastructure reporting.

PE creates the new "nixes" group, assigns the "ntp" class to all the nodes, and displays the results in the Groups view.

Now Judy wants to get the new class applied to the configuration of the servers in the group. She hops back to her terminal where she is still connected to her master and types `sudo /opt/puppet/bin/puppet agent -t` in order to kick off a puppet run manually (she could also just wait the default 30 minutes for the next scheduled puppet run). Puppet runs and the master compiles the updated manifest and then applies the resulting catalog, including the new NTP class, to the "puppet" RHEL box. She repeats the `puppet agent -t` command on the other boxes to get them updated as well. (If you like, you can learn more about this workflow to see all the steps in detail.)

To verify NTP is running, Judy ssh's into "web01" and runs `sudo /etc/init.d/ntpd status`; the

resulting status message confirms it's up.

She can also use the console to show her what happened. By clicking on the Nodes tab and then selecting, say, web01, she sees that there are two reports showing the most recents runs. Note that the color of the checkmarks by each node uses the same color key as other UI elements in reports, so blue indicates a successfully changed node, green indicates a successful run with no changes, etc. By clicking on a report, she can view the events in the run and see that the NTP class was successfully applied. She can also see other useful metrics such as how long Puppet runs take (which will be useful if she needs to performance tune her deployment down the road). Drilling down into the reports provides all kinds of details about what exactly PE did and when.

THING TWO: MOTD

Next, Judy decides to move on to managing MOTD on the marketing servers.

This is a good idea on Judy's part. While MOTD is not (usually) a mission-critical service, it can be a useful tool for communicating with users. MOTD is simple and low-risk, but automating it will keep building her experience and skills to automate similar resources. It's a very small step to go from automating MOTD to managing other text-like sorts of files like config files, boilerplate, HTML templates, etc.

Of course, that's not to say that MOTD in some modern systems (e.g., Ubuntu) can't be complex and powerful. Puppet can automate those kinds of implementations too. But for now, Judy needs to keep things simple. Further, it doesn't hurt that MOTD management is something she can do quickly and easily to show her boss that she's taking the right approach and getting tangible results.

As before, she starts by ssh'ing into her master and downloading her selected module by entering `sudo /opt/puppet/bin/puppet module install rcoleman-motd`. The module downloads and installs automatically.

The procedure continues just as it did with the NTP module: add the new class in the console and then apply it to the "nixes" group. After the next run, puppet compiles and applies the newly updated catalog, including the new MOTD class. (Note that if desired you can change the interval between automatic puppet runs.)

In this case, rather than going back to the command line, Judy decides to use Live Management to kick off the Puppet run. She does this by clicking Live Management on the main nav bar in the console and, making sure the "nixes" nodes are chosen in the Node selector on the left, she then clicks Control Puppet in the secondary nav bar. Then she clicks the "Run" button to kick off the run.

To make sure the new class works as expected, she logs out of her master and logs back in again to see if the new default MOTD message pops up. When she does that, she sees a new message containing a bunch of facter facts describing the machine and the classes puppet assigned to it on the last run. Cool.

```
------------------------------------------------
Welcome to the host named puppet
CentOS 6.4 x86_64
------------------------------------------------

FQDN: puppet.ps.eng.puppetlabs.net
IP:   10.16.133.138

Processor: Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
Memory:    1.83 GB

Puppet: 2.7.21 (Puppet Enterprise 2.8.1)
Facter: 1.6.17

The following classes have been declared on this node
during the most recent Puppet run against the Puppet
Master located at puppet.ps.eng.puppetlabs.net.
- pe_mcollective::role::console
- ntp
- pe_compliance
- motd
- pe_accounts
- pe_mcollective
- pe_mcollective::role::master
- settings
- default
- pe_mcollective::params
- pe_mcollective::role::console
- pe_mcollective::client::puppet_dashboard
- pe_mcollective::shared_key_files
- ntp::params
- ntp
- pe_compliance
- pe_compliance::agent
- motd
------------------------------------------------
```

Now she wants to tweak the MOTD message to identify the marketing department's machines. In the terminal, she navigates to the default module directory (`etc/puppetlabs/puppet/modules`) and locates the "motd" directory. In that directory, she opens up the `templates/motd.erb` file in vi and adds a line below line three that reads "Welcome to the NCC Marketing Department". She kicks off another Puppet run and sure enough, when she logs in to the master again, the new line has been added to the MOTD.

(Note that editing the motd.erb file directly means that Judy will have to be careful if and when

updates to the module are published. Running an upgrade could over-write her modified motd.erb and cause her to lose her changes. To avoid future problems, she will either have to create a fork of the module, make a custom wrapper module or, at minimum, make sure to create a duplicate of her custom motd.erb file before upgrading. Forking and wrapping will be discussed in the next chapter.)

In the interest of thoroughness, she ssh's into the Apache server where she sees that the default MOTD message has been added, but her new welcome message isn't there yet. Ah, right, a new catalog won't be compiled until the next puppet run. Judy kicks off a run manually and sure enough, she sees a notice in the run output telling her the new line has been added. A quick double-check by logging back into the Apache server confirms the new message is there.

Easy enough. Now Judy has something to show her boss to prove that Puppet automation works, and she's confident that PE is correctly controlling and communicating with the infrastructure. She also knows that she can successfully edit text-based files and use PE to deliver those files to chosen elements of the infrastructure.

On to something a little more challenging: using PE to configure and manage Apache.

THING THREE: APACHE

This webserver is a consistent thorn in Judy's side. A lot of people touch it, marketing people. They are constantly tweaking pages, updating forms processing, adding the latest whiz-bang metrics tools, and who knows what else. They are all very nice people, but none of them has the slightest idea how an Apache server works. The most recent problem is that someone has been mucking around in `/etc` and has deleted the Virtual Hosts config file. As a result, the site is down and only displaying the Apache default page.

It's obviously time to get the webserver under automation. Back to the Forge again, where Judy soon finds the Puppet Labs Apache module. Another quick `sudo /opt/puppet/bin/puppet module install puppetlabs-apache` on the master node and the module is installed.

Out of the box, the module doesn't quite match the Apache configuration Judy needs. Since you can't pass parameters directly via the console, she needs to create a new class that describes how NCC's Apache configuration is supposed to work. To do that, she needs to build a custom module where she can define the class. On the master, Judy navigates to `/etc/puppetlabs/puppet/modules` and she creates the framework for the module with a simple `sudo mkdir -p ncc/manifests`. This is the basic directory structure Puppet expects for class definitions in modules.

Next, Judy uses a text editor (vi, nano, whatever) to create a `web.pp` file in the new `ncc/manifests` directory. Then, along the lines of the example in the module documentation, she writes the following Puppet code to create a new class called `ncc::web`:

```
class ncc::web {
  class { 'apache': }
  class { 'apache::php': }
```

```
  apache::vhost { 'www.ncc.com':
    priority   => '10',
    vhost_name => '10.16.1.2',
    port       => '80',
    docroot    => '/var/www/html',
  }
}
```

Note: if this puppet code is opaque and unfamiliar to you, you may wish to spend some time learning about writing manifests. To learn about a specific type, check out the reference guide.

Tip: Don't edit manifests or templates in Notepad. It will introduce CLRF-style line breaks that can break things, especially in the case of templates used for config files whose applications care about whitespace.

To make sure her code is clean and correct, she writes out the new `web.pp` file and then runs it through the puppet syntax checker with `puppet parser validate web.pp`. The command returns nothing, indicating the code is clean. (To be extra sure, she could do a quick `echo $?` which should return `0`; anything non-zero would indicate an error such as a non-existent file.)

Now Judy goes back to the console where the new class can be added in the usual way, by clicking the "Add class" button and entering `ncc::web`, which PE automatically discovers and adds.

Judy obviously doesn't need to assign the new Apache class to every machine in the default or nixes groups, she only needs it on "web01." However, knowing that as she moves automation beyond the marketing department she will have more webservers to manage, Judy decides to create a new group specifically for Apache nodes.

In the console, she clicks the "Add group" button in the Group view, she names the new group "webservers" and assigns the "ncc::web" class to it. Then, in the Nodes box she starts typing the name of the Ubuntu webserver, "web01", to add it. Then she clicks the "Create" button to create the new group. Lastly, she uses Live Management to kick off a puppet run on "web01" to get it configured according to the new class. Looking at the node in the console she can see that the new group has been created, with "web01" as a member and with the class `ncc::web` included. The blue checkmark indicates new changes were applied in the last run (specifically, the webservers group was created with "web01" as a member and containing the `ncc::web` class).

If the code is working correctly, the new class should have configured the site's vhost correctly and pointed it at the correct root document directory. A quick look at the output from the puppet run shows that config file has been properly created. Sure enough, when Judy points her browser at NCC's home page, the site comes up. Looking at the glamor shots of the new model 1701 on the

home page, Judy smiles and indulges herself with a mental high-five; thanks to PE, she knows the server is up and staying up no matter much stupid gets thrown at it.

Next: Puppetize Your Infrastructure: More Advanced Automating (coming soon)

1. Judy Argyle is a fictitious character. Any resemblance to any person living or dead is purely coincidental.