O'REILLY®

Free Sampler

Learning Puppet 4

A GUIDE TO CONFIGURATION MANAGEMENT AND AUTOMATION

Jo Rhett

O'REILLY®

Learning Puppet 4

If you're a system administrator, developer, or site reliability engineer responsible for handling hundreds or even thousands of nodes in your network, the Puppet configuration management tool will make your job a whole lot easier. This practical guide shows you what Puppet does, how it works, and how it can provide significant value to your organization.

Through hands-on tutorials, DevOps engineer Jo Rhett demonstrates how Puppet manages complex and distributed components to ensure service availability. You'll learn how to secure configuration consistency across servers, clients, your router, and even that computer in your pocket by setting up your own testing environment.

- Learn exactly what Puppet is, why it was created, and what problems it solves
- Tailor Puppet to your infrastructure with environment layouts that meet your specific needs
- Write declarative Puppet policies to produce consistency in your systems
- Build, test, and publish your own Puppet modules
- Manage network devices such as routers and switches with *puppet device* and integrated Puppet agents
- Scale Puppet servers for high availability and performance
- Explore web dashboards and orchestration tools that supplement and complement Puppet

Jo Rhett is a DevOps engineer with 20 years of experience conceptualizing and delivering large-scale Internet services. He focuses on creating automation and infrastructure to accelerate deployment and minimize outages.

SYSTEM ADMINISTRATION

US $49.99 CAN $57.99
ISBN: 978-1-491-90766-5

Twitter: @oreillymedia
facebook.com/oreilly

Learning Puppet 4

A Guide to Configuration Management and Automation

Jo Rhett

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning Puppet 4
by Jo Rhett

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan
Proofreader: Jasmine Kwityn

Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2016: First Edition

Revision History for the First Edition
2016-03-22: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491907665* for release details.

978-1-491-90766-5

[LSI]

Table of Contents

Part IV. Integrating Puppet

CHAPTER 1

Thinking Declarative

If you have experience writing shell, Ruby, Python, or Perl scripts that make changes to a system, you've very likely been performing *imperative programming*. Imperative programming issues commands that change a target's state, much as the imperative grammatical mood in natural language expresses commands for people to act on.

You may be using *procedural programming* standards, where state changes are handled within procedures or subroutines to avoid duplication. This is a step toward declarative programming, but the main program still tends to define each operation, each procedure to be executed, and the order in which to execute them in an imperative manner.

While it can be useful to have a background in procedural programming, a common mistake is to attempt to use Puppet to make changes in an imperative fashion. The very best thing you can do is forget everything you know about imperative or procedural programming.

If you are new to programming, don't feel intimidated. People without a background in imperative or procedural programming can often learn good Puppet practices faster.

Writing good Puppet manifests requires declarative programming. When it comes to maintaining configuration on systems, you'll find declarative programming to be easier to create, easier to read, and easier to maintain. Let's show you why.

Handling Change

The reason that you need to cast aside imperative programming is to handle change better.

When you write code that performs a sequence of operations, that sequence will make the desired change the first time it is run. If you run the same code the second time in a row, the same operations will either fail or create a different state than desired. Here's an example:

```
$ sudo useradd -u 1001 -g 1001 -c "Joe User" -m joe
$ sudo useradd -u 1001 -g 1000 -c "Joe User" -m joe
useradd: user 'joe' already exists
```

So then you need to change the code to handle that situation:

```
# bash excerpt
getent passwd $USERNAME > /dev/null 2> /dev/null
if [ $? -ne 0 ]; then
    useradd -u $UID -g $GID -c "$COMMENT" -s $SHELL -m $USERNAME
else
    usermod -u $UID -g $GID -c "$COMMENT" -s $SHELL -m $USERNAME
fi
```

OK, that's six lines of code and all we've done is ensure that the username isn't already in use. What if we need to check to ensure the UID is unique, the GID is valid, and that the password expiration is set? You can see that this will be a very long script even before we adjust it to ensure it works properly on multiple operating systems.

This is why we say that imperative programming doesn't handle change very well. It takes a lot of code to cover every situation you need to test.

Using Idempotence

When managing computer systems, you want the operations applied to be *idempotent*, where the operation achieves the same results every time it executes. Idempotence allows you to apply and reapply (or converge) a configuration manifest and always achieve the desired state.

In order for imperative code to be idempotent, it needs to have instructions for how to compare, evaluate, and apply not just every resource, but also each attribute of the resource. As you saw in the previous section, even the simplest of operations will quickly become ponderous and difficult to maintain.

What is an Idempotent Operation?

In mathematics and computer science, idempotent operations are those that can be applied multiple times without changing the result beyond the initial application. The word literally means "[the quality of] having the same power," from the Latin roots *idem* + *potent* "same" + "power."[1] Here are some examples of idempotent and non-idempotent math and code:

`any number^1`	**Idempotent**	A number to the power of 1 is the same
`value = value * 2`	**Non-idempotent**	Will double every time
`value = value * 2 / 2`	**Idempotent**	Remains the same value
`echo "Good!" >> /some/file`	**Non-idempotent**	File will keep growing
`echo "Good!" > /some/file`	**Idempotent**	File will always have the same content

The simplistic final example avoids having to compare the state of the item by simply overwriting it every time. This only works in a limited set of situations. Most changes require evaluation to determine what changes are necessary.

Declaring Final State

As we mentioned in Introduction, for a configuration state to be achieved no matter the conditions, the configuration language must avoid describing the actions required to reach the desired state. Instead, the configuration language should describe the desired state itself, and leave the actions up to the interpreter. Language that declares the final state is called *declarative*.

Rather than writing extensive imperative code to handle every situation, it is much simpler to declare what you want the final state to be. In other words, instead of including dozens of lines of comparison, the code reflects only the desired final state of the resource (a user account, in this example). Here we will introduce you to your first bit of Puppet configuration language, a resource declaration for the same user we created earlier:

1 First seen in George Boole's book *The Mathematical Analysis of Logic*, originally published in 1847.

```
user { 'joe':
  ensure     => present,
  uid        => '1001',
  gid        => '1000',
  comment    => 'Joe User',
  managehome => true,
}
```

As you can see, the code is not much more than a simple text explanation of the desired state. A user named *Joe User* should be *present*, a home directory for the user should be created, and so on. It is very clear, very easy to read. Exactly how the user should be created is not within the code, nor are instructions for handling different operating systems.

Declarative language is much easier to read, and less prone to breakage due to environment differences. Puppet was designed to achieve consistent and repeatable results. You describe what the final state of the resource should be, and Puppet will evaluate the resource and apply any necessary changes to reach that state.

Reviewing Declarative Programming

Conventional programming languages create change by listing exact operations that should be performed. Code that defines each state change and the order of changes is known as *imperative programming*.

Good Puppet manifests are written with declarative programming. Instead of defining exactly how to make changes, in which you must write code to test and compare the system state before making that change, you instead declare how it should be. It is up to the Puppet agent to evaluate the current state and apply the necessary changes.

As this chapter has demonstrated, declarative programming is easier to create, easier to read, and easier to maintain.