O'REILLY®

Early Release

RAW & UNEDITED

Puppet Best Practices

DESIGN PATTERNS FOR MAINTAINABLE CODE

Chris Barbour

Puppet Best Practices

Chris Barbour

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Puppet Best Practices

by Chris Barbour

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: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2015-03-13: First early release

2015-06-04: Early release revision 2

See *http://oreilly.com/catalog/errata.csp?isbn=0636920038528* for release details.

ISBN: 063-6-920-03852-8

[?]

Table of Contents

CHAPTER 1

Forward

First of all, I'd like to thank you for taking the time to read this early preview of Puppet Best Practices. The reception to the book has been amazing, and we've received great feedback about our initial update.

What's new?

This months update includes two new chapters. Chapter 4 focusing on coding best practices and the features of the Puppet DSL. This chapter explores iterators, variables, and conditional logic among other language features. It also discusses coding principles as they apply to Puppet.

Chapter 5 focuses on the development of Puppet modules. In this chapter we explore module structure, scoping, the use of defined types, and module testing.

Puppet 4.0 was officially launched following our previous pre-release. We had already been discussing practices relating to Puppet 4. This update significantly extends our Puppet 4 coverage.

Puppet 4 is our target platform for this book, however we expect that a lot of sites will continue to use Puppet 3 for the foreseeable future. We attempt to address practices for both major releases of Puppet, directly exploring differences between the two releases where applicable. This should both help you handle your current requirements, and plan your approach for the future.

The introduction of the future parser and the release of Hiera 2 are both fairly significant changes for Puppet 4. We are excited about the introduction of iterators to the Puppet DSL, as well as module data sources, and the improvements for data handling. We have however, found that most best practices apply to both releases of Puppet.

Best practices will become even more important to this release; the new features of Puppet 4 are powerful, but carry a lot of potential for mis-use. This book should prove to be invaluable as you upgrade, grow, and improve your environment.

About best practices

Best practices books are the result of years of professional experience. Unlike purely technical works, there often isn't always objectively correct best practices answer for every conceivable situation. By exposing this early release to your criticism, I hope to identify weaknesses in this book; sections that can be improved upon. If you find anything in this book confusing or incomplete, I encourage you to reach out for clarification. My goal is to produce the best work I possibly can. Your feedback is invaluable to this process, and very much appreciated.

My objective for this book is to share professional experience, help other IT professionals solve problems, and to improve the overall quality of code and infrastructure deployed in the real world.

About this pre-release

This work has not yet been professionally edited, and I hope you will forgive the grammar, syntax, and formatting errors that are present in this initial draft. Our goal is to document the core concepts of this book first, and then correct the flaws iteratively. What you currently see here is the product of passion and many sleepless nights fueled only by caffeine and determination.

While all feedback is sincerely appreciated, the most valuable feedback you can provide are your thoughts regarding the concepts and recommendations provided by this book. Do you find the discussion of programming principles useful? Do you disagree with any of the recommendations, and if so why? Is the material easy to understand, or confusing?

If you have any feedback, I request that you please reach out. Please feel free to email me at *puppet.best.practices@gmail.com*

Sincerely,

Chris Barbour, Author, Puppet Best Practices

Preface

This book is a work in progress – new chapters will be added as they are written. We welcome feedback – if you spot any errors or would like to suggest improvements, please email the author at *puppet.best.practices@gmail.com*.

In this book, we discuss on how to build and deploy highly maintainable Puppet code, with a focus on avoiding and eliminating technical debt.

Puppet is by far the most popular configuration management and automation platform available today, and is quickly being adopted by companies of all sizes, from single platform startups to established enterprises that may be using it to manage half a dozen different operating systems.

This book does not attempt to explain the basics of using Puppet; instead we will be looking at how to use Puppet most effectively. This will include discussions about the design decisions behind Puppet, an exploration of how to organize code and data, a look at many common features of Puppet, and discussions about common pitfalls and traps when deploying Puppet infrastructure.

Who Should Read This Book

This book is intended for readers who have some basic familiarity with Puppet and are interested in improving their understanding of Puppet and the surrounding ecosystem of tools. The concepts described in this book are appropirate for Puppet novices and experts alike, however readers who are new to Puppet should consider using this as a suppliment to Puppet Labs training or other introductory work that can provide a foundational understanding of Puppet.

This book is appropriate for all professionals working with Puppet, regardless of whether you are responsible for architecting Puppet infrastructure or responsible for writing a module or two to support a single application.

The information contained in this book will be invaluable to both green and brown field deployments of Puppet. If you are building a new environment, we will discuss how to lay a solid foundation that provides flexibility to grow and change. If you already have an exiting environment or are inheriting an environment, we will explore useful strategies to eliminate many pain points in your code base and improve upon what you have.

Why I Wrote This Book

This book draws heavily from experiences as a Puppet consultant. The folks I've met while deploying Puppet have been very bright and talented individuals who were quick to learn the features of Puppet, and could often find very innovative ways of using those features to solve immediate problems, and to produce code that is impossible to understand.

Almost universlly, sites deploying Puppet have grown both quickly and organically. The consequences of design decisions made early in development often had reprocussions that were not obvious when they were originally made. As code became established and moved to production, it became harder and harder to correct those problems without risking the stability of the site that code manages.

None of these problems are specific to Puppet. As with any CFM solution, the amount of code being maintained will grow with the size of the site. It will often dwarf any other single codebase maintained by your operations team.

This book highlights design patterns, both good and bad, that can be used when building Puppet environments and discusses the impact of each decision. The coding patterns contained in this book will help you design code that can be extended, maintained, and supported not only by yourself, but by the people who may inherit your work down the road.

A Word on Puppet Today

Puppet Best Practices have changed significantly since Pupet's early releases. In some ways, Puppet is much easier to work with and support. For example, parameterized modules and automatic data bindings have made it much simpler to re-use 3rd party modules. On the other hand, Puppet has added many new features and design patterns since then, and the demarcation points between various systems aren't always clear.

Best practices will continue to evolve. Puppet 4 will introduce new syntax and features. Common complaints will be resolved. New issues will be discovered.

The goal of this book is not just to help you solve the problems you may be experiencing today, but also to help prepare you for the future.

Navigating This Book

This book is organized roughly as follows:

- Chapters 1 and 2 discuss design concepts that drive the recommendations made throughout this book.
- Chapters 3 through 8 explore major features of Puppet such as Hiera in depth and provide many concrete recommendations for each component.
- Chapter 9 and 10 discusses release management practices and development tools.
- Chapter 11 and 13 discuss advanced topics relating to extending Puppet and other infrastructure management tools that may be useful to your site.

This book is organized so that it can be read front to back, however most of the chapters in this book are fairly self contained and will provide references to other topics where appropriate.

I strongly encourage you to start with chapters 2 and 3. From there, feel free to skip around if you're the impatient type, or continue reading through to gain a wholistic understanding of the Puppet infrastructure. Once you've read through this book, it is my sincere hope that you will return to individual sections when needed to address a difficult problem, to refresh your knowledge, or to pickup strategies that may have been missed the first time through.

Online Resources

- *https://docs.puppetlabs.com/*
- *https://ask.puppetlabs.com*

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

Constant width italic
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This book would not have been possible without the patience and support of my wife and son.

CHAPTER 2

The Puppet Design Philosophy

Before we begin to explore practical best practices with Puppet, it's valuable to understand the reasoning behind these recommendations.

Puppet can be somewhat alien to technologists who have a background in automation scripting. Where most of our scripts scripts are procedural, Puppet is *declarative*. While a declarative language has many major advantages for configuration management, it does impose some interesting restrictions on the approaches we use to solve common problems.

Although Puppet's design philosophy may not be the most exciting topic to begin this book, it drives many of the practices in the coming chapters. Understanding that philosophy will help contextualize many of the recommendations covered in this book.

Declarative Code

The Puppet Domain Specific Language (DSL) is a declarative language, as opposed to the imperative or procedural languages that system administrators tend to be most comfortable and familiar with.

In an imperative language, we describe how to accomplish a task. In a declarative language, we describe what we want to accomplish. Imperative languages focus on actions to reach a result, and declarative languages focus on the result we wish to achieve. We will see examples of the difference below.

Puppet's language is (mostly) verb-less. Understanding and internalizing this paradigm is critical when working with Puppet; attempting to force Puppet to use a procedural or imperative process can quickly become an exercise in frustration and will tend to produce fragile code.

In theory, a declarative language ideal for configuration base-lining tasks. With the Puppet DSL, we describe the desired state of our systems, and Puppet handles all responsibility for making sure the system conforms to this desired state. Unfortunately, most of us are used to a procedural approach to system administration. The vast majority of the bad Puppet code I've seen has been the result of trying to write procedural code in Puppet, rather than adapting existing procedures to Puppet's declarative model.

Procedural Code with Puppet

In some cases writing procedural code in Puppet is unavoidable. However, such code is rarely elegant, often creates unexpected bugs, and can be difficult to maintain. We will see practical examples and best practices for writing procedural code when we look at the exec resource type in Chapter 6

Of course, it's easy to simply say "be declarative." In the real world, we are often tasked to deploy software that isn't designed for a declarative installation process. A large part of this book will attempt to address how to handle many uncommon tasks in a declarative way. As a general rule, if your infrastructure is based around packaged open source software, writing declarative Puppet code will be relatively straight forward. Puppet's built in types and providers will provide a declarative way to handle most of your operational tasks. If you're infrastructure includes Windows clients and a lot of Enterprise software writing declarative Puppet code may be significantly more challenging.

Another major challenge system administrators face when working within the constraints of a declarative model is that we tend to operate using an imperative work-flow. How often have you manipulated files using regular expression substitution? How often do we massage data using a series of temp files and piped commands? While Puppet offers many ways to accomplish the same tasks, most of our procedural approaches do not map well into Puppet's declarative language. We will explore some examples of this common problem, and discuss alternative approaches to solving it.

What is declarative code anyway?

As mentioned earlier, declarative code tends not to have verbs. We don't create users and we don't remove them; we ensure that the users are present or absent. We don't install or remove software; we ensure that software is present or absent. Where create and install are verbs, present and absent are adjectives. The difference seems trivial at first, but proves to be very important in practice.

A real world example:

Imagine that I'm giving you directions to the Palace of Fine Arts in San Francisco.

Procedural instructions:

- Head North on 19th Avenue
- Get on US-101S
- Take Marina Blvd. to Palace Dr.
- Park at the Palace of Fine Arts Theater

These instructions make a few major assumptions:

- You aren't already at the Palace of Fine Arts
- You are driving a car
- You are currently in San Francisco
- You are currently on 19th avenue or know how to get there.
- You are heading North on 19th avenue.
- There are no road closures or other traffic disruptions that would force you to a different route.

Compare this to the declarative instructions:

- Be at 3301 Lyon Street, San Francisco, CA 94123 at 7:00PM

The declarative approach has a few major advantages in this case:

- It makes no assumptions about your mode of transportation. These instructions are still valid if your plans involve public transit or parachuting into the city.
- The directions are valid even if you're currently at the Palace of Fine Arts
- These instructions empower you to route around road closures and traffic

The declarative approach allows you to chose the best way to reach the destination based on your current situation, and it relies on your ability to find your way to the destination given.

Declarative languages aren't magic. Much like an address relies on you understanding how to read a map or use a GPS device, Puppet's declarative model relies on it's own procedural code to turn your declarative request into a set of instructions that can achieve the declared state. Puppet's model uses a *Resource type* to model an object, and a *provider* implementing procedural code to produce the state the model describes.

The major limitation imposed by Puppet's declarative model might be somewhat obvious. If a native resource type doesn't exist for the resource you're trying to model, you can't manage that resource in a declarative way. Declaring that I want a red two story house with 4 bedrooms might empower you to build the house out of straw or wood or

brick, but it probably won't actually accomplish anything if you don't happen to be a general contractor.

There is some good news on this front, however. Puppet already includes native types and providers for most common objects, the Puppet community has supplied additional native models, and if you absolutely have to accomplish something procedurally you can almost always fall back to the `exec` resource type.

A practical example

Let's examine a practical example of procedural code for user management. We will discuss how to make the code can be made robust, it's declarative equivalent in Puppet, and the benefits of using Puppet rather than a shell script for this task.

Imperitive / Procedural Code

Here's an example of an imperative process using BASH. In this case, we are going to create a user with a home directory and an authorized SSH key on a CentOS 6 Host.

Example 2-1. Imperative user creation with BASH

```
groupadd examplegroup
useradd -g examplegroup alice
mkdir ~alice/.ssh/
chown alice.examplegroup ~alice/.ssh
echo "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAm3TAgMF/2RY+r7KIeUoNbQb1TP6ApOtgJPNV\
0TY6teCjbxm7fjzBxDrHXBS1vr+fe6xa67G5ef4sRLl0kkTZisnIguXqXOaeQTJ4Idy4LZEVVbngkd\
2R9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc= \
alice@localhost" > ~alice/.ssh/authorized_keys
```

What if we decide this user should also be a member of the wheel group?

Example 2-2. Imperative user modification with BASH

```
useradd -g examplegroup alice
usermod -G wheel alice
```

And if we want to remove that user and that user's group?

Example 2-3. Imperative user removal with BASH

```
userdel alice
groupdel examplegroup
```

Notice a few things about this example:

- Each process is completely different
- The correct process to use depends on the current state of the user
- Each of these processes will produce errors if invoked more than one time

Imagine for a second that we have several systems. On some systems, example user is absent. On other systems, Alice is present, but not a member of the wheel group. On some systems, Alice is present and a member of the wheel group. Imagine that we need to write a script to ensure that Alice exists, and is a member of the wheel group on every system, and has the correct authorized key. What would such a script look like?

Example 2-4. Robust user management with BASH

```
#!/bin/bash

if ! getent group examplegroup; then
  groupadd examplegroup
fi

if ! getent passwd alice; then
  useradd -g examplegroup -G wheel alice
fi

if ! id -nG alice | grep -q 'examplegroup wheel'; then
  usermod -g examplegroup -G wheel alice
fi

if ! test -d ~alice/.ssh; then
  mkdir -p ~alice/.ssh
fi

chown alice.examplegroup ~alice/.ssh

if ! grep -q alice@localhost ~alice/.ssh/authorized_keys; then
    echo "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAm3TAgMF/2RY+r7KIeUoNbQb1TP6ApOtg\
JPNV0TY6teCjbxm7fjzBxDrHXBS1vr+fe6xa67G5ef4sRLl0kkTZisnIguXqXOaeQTJ4Idy4LZEVVb\
ngkd2R9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc= \
alice@localhost" >> ~alice/.ssh/authorized_keys
fi

chmod 600 ~alice/.ssh/authorized_keys
```

Of course, this example only covers the use case of creating and managing a few basic properties about a user. If our policy changed, we would need to write a completely different script to manage this user. Even fairly simple changes, such as revoking this user's wheel access could require somewhat significant changes to this script.

This approach has one other major disadvantage; it will only work on platforms that implement the same commands and arguments of our reference platform. This example will fail on FreeBSD (implements adduser, not useradd) Mac OSX, and Windows.

Declarative Code

Let's look at our user management example using Puppet's declarative DSL.

Creating a user and group:

Example 2-5. Declarative user creation with Puppet

```
$ssh_key = "AAAAB3NzaC1yc2EAAAABIwAAAIEAm3TAgMF/2RY+r7KIeUoNbQb1TP6ApOtgJPNV0T\
Y6teCjbxm7fjzBxDrHXBS1vr+fe6xa67G5ef4sRLl0kkTZisnIguXqXOaeQTJ4Idy4LZEVVbngkd2R\
9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc="

group { 'examplegroup':
  ensure => 'present',
}

user { 'alice':
  ensure     => 'present',
  gid        => 'examplegroup',
  managehome => true,
}

ssh_authorized_key { 'alice@localhost':
  ensure => 'present',
  user   => 'alice',
  type   => 'ssh-rsa',
  key    => $ssh_key,
}
```

Adding alice to the wheel group:

Example 2-6. Declarative group membership with puppet

```
$ssh_key = "AAAAB3NzaC1yc2EAAAABIwAAAIEAm3TAgMF/2RY+r7KIeUoNbQb1TP6ApOtgJPNV0T\
Y6teCjbxm7fjzBxDrHXBS1vr+fe6xa67G5ef4sRLl0kkTZisnIguXqXOaeQTJ4Idy4LZEVVbngkd2R\
9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc="

group { 'examplegroup':
  ensure => 'present',
}

user { 'alice':
  ensure     => 'present',
  gid        => 'examplegroup',
  groups     => 'wheel',        # (1)
  managehome => true,
}

ssh_authorized_key { 'alice@localhost':
  ensure => 'present',
  user   => 'alice',
  type   => 'ssh-rsa',
  key    => $ssh_key,
}
```

(1) Note that the only change between this example and the previous example is the addition of the `groups` parameter for the alice resource.

Remove alice:

Example 2-7. Ensure that a user is absent using Puppet

```
$ssh_key = "AAAAB3NzaC1yc2EAAAABIwAAAIEAm3TAgMF/2RY+r7KIeUoNbQb1TP6ApOtgJPNV0T\
Y6teCjbxm7fjzBxDrHXBS1vr+fe6xa67G5ef4sRLl0kkTZisnIguXqXOaeQTJ4Idy4LZEVVbngkd2R\
9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc="

group { 'examplegroup':
  ensure => 'absent',           # ❶
}

user { 'alice':
  ensure     => 'absent',       # ❷
  gid        => 'examplegroup',
  groups     => 'wheel',
  managehome => true,
}

ssh_authorized_key { 'alice@localhost':
  ensure => 'absent',           # ❸
  user   => 'alice',
  type   => 'ssh-rsa',
  key    => $ssh_key,
}

Ssh_authorized_key['alice@localhost'] -> # ❹
User['alice']                         -> # ❺
Group['examplegroup']
```

❶ ❷ Ensure values are changed from Present to Absent.

❸

❹ ❺ Resource ordering is added to ensure groups are removed after users. Normally, the correct order is implied due to the Autorequire feature discussed in Chapter 6

You may notice the addition of resource ordering in this example when it wasn required in previous examples. This is a byproduct of Puppet's Autorequire feature. PUP-2451 explains the issue in greater depth.

In practice, rather than managing Alice as 3 individual resources, we would abstract this into a defined type that has its own ensure parameter, and conditional logic to enforce the correct resource dependency ordering.

In this example, we are able to remove the user by changing the ensure state from present to absent on the user's resources. Although we could remove other parameters such as gid, groups, and the users key, in most cases it's better to simply leave the values in place, just in case we ever decide to restore this user.

It's usually best to disable accounts rather than remove them. This helps preserve file ownership information and helps avoid UID reuse.

In our procedural examples, we saw a script that would bring several divergent systems into conformity. For each step of that example script, we had to analyze the current state of the system, and perform an action based on state. With a declarative model, all of that work is abstracted away. If we wanted to have a user who was a member of 2 groups, we would simply declare that user as such, as in Example 2-6.

Non-Declarative Code with Puppet

It is possible to write non-declarative code with Puppet. Please don't do this:

```
$app_source  = 'http://www.example.com/application.tar.gz'
$app_target  = '/tmp/application.tar.gz'

exec { 'download application':
  command => "/usr/bin/wget -q ${app_source} -O ${app_target}",
  creates => '/usr/local/application/',
  notify => exec['extract application'],
}

exec { 'extract application':
  command     => "/bin/tar -zxf ${app_target} -C /usr/local",
  refreshonly => true,
  creates     => '/usr/local/application/',
}
```

This specific example has a few major problems:

1. Exec resources have a set timeout. This example may work well over a relatively fast corporate netowkr connection, and then fail completely from a home DSL line. The solution would be to set the timeout parameter of the exec resources to a reasonably high value.
2. This example does not validate the checksum of the downloaded file, which could produce some odd results upon extraction. An additional exec resource might be used to test and correct for this case automatically.

3. In some cases, a partial or corrupted download may wedge this process. We attempt to work around this problem by overwriting the archive each time it's downloaded.
4. This example makes several assumptions about the contents of application.tar.gz. If any of those assumptions are wrong, these commands will repeat every time Puppet is invoked.
5. This example is not particularly portable, and would require a platform specific implementation for each supported OS.
6. This example would not be particularly useful for upgrading the application.

This is a relatively clean example of non-declarative Puppet code, and tends to be seen when working with software that is not available in a native packaging format. Had this application been distributed as an RPM, dpkg, or MSI, we could have simply used a package resource for improved portability, flexibility, and reporting. While this example is not best practices, there are situations where is unavoidable, often for business or support reasons.

This example could be made declarative using the nanliu/staging module.

Another common pattern is the use of conditional logic and custom facts to test for the presence of software. Please don't do this unless it's absolutely unavoidable:

```
Facter.add(:example_app_version) do
  confine :kernel => 'Linux'
  setcode do
    Facter::Core::Execution.exec('/usr/local/app/example_app --version')
  end
end

$app_source       = 'http://www.example.com/app-1.2.3.tar.gz'
$app_target       = '/tmp/app-1.2.3.tar.gz'

if $example_app_version != '1.2.3' {
  exec { 'download application':
    command => "/usr/bin/wget -q ${app_source} -O ${app_target}",
    before  => exec['extract application'],
  }

  exec { 'extract application':
    command => "/bin/tar -zxf ${app_target} -C /usr/local",
  }
}
```

This particular example has many of the same problems of the previous example, and introduces one new problem: it breaks Puppet's reporting and auditing model. The conditional logic in this example causes the download and extraction resources not to appear in the catalog sent to the client following initial installation. We won't be able to audit our run reports to see whether or not the download and extraction commands are in a consistent state. Of course, we could check the `example_application_version` fact if it happens to be available, but this approach becomes increasingly useless as more resources are embedded in conditional logic.

This approach is also sensitive to facter and pluginsync related issues, and would definitely produce some unwanted results with cached catalogs.

Using facts to exclude parts of the catalog does have one benefit: it can be used to obfuscate parts of the catalog so that sensitive resources do not exist in future Puppet runs. This can be handy if, for example, your wget command embeds a pass-phrase, and you wish to limit how often it appears in your catalogs and reports. Obviously, there are better solutions to that particular problem, but in some cases there is also benefit to security in depth.

Idempotency

In computer science, an idempotent function is a function that will return the same value each time it's called, whether it's only called once, or called 100 times. For example: `X = 1` is an idempotent operation. `X = X + 1` is a non-idempotent,recursive operation.

Puppet as a language is designed to be inherently idempotent. As a system, Puppet designed to be used in an idempotent way. A large part of this idempotency owed to it's declarative resource management model, however Puppet also enforces a number of rules on it's variable handling, iterators, and conditional logic to maintain it's idempotency.

Idempotence has major benefits for a configuration management language:

- The configuration is inherently self healing
- State does not need to be maintained between invocations
- Configurations can be safely re-applied

For example, if for some reason Puppet fails part way through a configuration run, re-invoking Puppet will complete the run and repair any configurations that were left in an inconsistent state by the previous run.

Convergence vs Idempotence

Configuration management languages are often discussed in terms of their convergence model. Some tools are designed to be eventually convergent; others immediately convergent and/or idempotent.

With an eventually convergent system, the configuration management tool is invoked over and over; each time the tool is invoked, the system approaches a converged state, where all changes defined in the configuration language have been applied, and no more changes can take place. During the process of convergence, the system is said to be in a partially converged, or inconsistent state.

For Puppet to be idempotent, it cannot by definition also be eventually convergent. It must reach a convergent state in a single run, and remain in the same state for any subsequent invocations. Puppet can still be described as an immediately convergent system, since it is designed to reach a convergent state after a single invocation.

Convergence of course also implies the existence of a diverged state. Divergence is the act of moving the system away from the desired *converged* state. This typically happens when someone attempts to manually alter a resource that is under configuration management control.

There are many practices that can break Puppet's idempotence model. In most cases, breaking Puppet's idempotence model would be considered a bug, and would be against best practices. There are however some cases where a level of eventual convergence is unavoidable. One such example is handling the numerous post-installation software reboots that are common when managing Windows nodes.

Side Effects

In computer science, a side effect is a change of system or program state that is outside the defined scope of the original operation. Declarative and idempotent languages usually attempt to manage, reduce, and eliminate side effects. With that said, it is entirely possible for an idempotent operation to have side effects.

Puppet attempts to limit side effects, but does not eliminate them by any means; doing so would be nearly impossible given Puppet's role as a system management tool.

Some side effects are designed into the system. For example, every resource will generate a notification upon changing a resource state that may be consumed by other resources. The notification is used to restart services in order to ensure that the running state of the system reflects the configured state. File bucketing is another obvious intended side-effect designed into Puppet.

Some side effects are unavoidable. Every access to a file on disk will cause that file's atime to be incremented unless the entire file-system is mounted with the noatime attribute. This is of course true whether or not Puppet is being invoked in noop mode.

Resource Level Idempotence

Many common tasks are not idempotent by nature, and will either throw an error or produce undesirable results if invoked multiple times. For example, the following code is not idempotent because it will set a state the first time, and throw an error each time it's subsequently invoked.

Example 2-8. A non-idempotent operation that will throw an error

```
useradd alice
```

The following code is not idempotent, because it will add undesirable duplicate host entries each time it's invoked:

Example 2-9. A non-idempotent operation that will create duplicate records

```
echo '127.0.0.1 example.localdomin' >> /etc/hosts
```

The following code is idempotent, but will probably have undesirable results:

Example 2-10. An idempotent operation that will destroy data

```
echo '127.0.0.1 example.localdomin' > /etc/hosts
```

To make our example idempotent without clobbering /etc/hosts, we can add a simple check before modifying the file:

Example 2-11. An imperative idempotent operation

```
grep -q '^127.0.0.1 example.localdomin$' /etc/hosts \
    || echo '127.0.0.1 example.localdomin' >> /etc/hosts
```

The same example is simple to write in a declarative and idempotent way using the native Puppet host resource type:

Example 2-12. Declarative Idempotence with Puppet

```
host { 'example.localdomain':
  ip => '127.0.0.1',
}
```

Alternatively, we could implement this example using the *file_line* resource type from the optional *stdlib* Puppet module:

Example 2-13. Idempotent host entry using the File_line resource type

```
file_line { 'example.localdomin host':
  path => '/etc/hosts',
```

```
  line => '127.0.0.1 example.localdomain',
}
```

In both cases, the resource is modeled in a declarative way and is idempotent by it's very nature. Under the hood, Puppet handles the complexity of determining whether the line already exists, and how it should be inserted into the underlying file. Using the native host resource type, Puppet also determines what file should be modified and where that file is located. Example 2-12 will work on Windows.

The idempotent examples are safe to run as many times as you like. This is a huge benefit across large environments; when trying to apply a change to thousands of hosts, it's relatively common for failures to occur on a small subset of the hosts being managed. Perhaps the host is down during deployment? Perhaps you experienced some sort of transmission loss or timeout when deploying a change? If you are using an idempotent language or process to manage your systems, it's possible to handle these exceptional cases simply by performing a second configuration run against the affected hosts (or even against the entire infrastructure.)

When working with native resource types, you typically don't have to worry about idempotence; most resources handle idempotence natively. A couple of notable exceptions to this statement are the `exec` and `augeas` resource types. We'll explore those in depth in Chapter 6.

Puppet does however attempt to track whether or not a resource has changed state. This is used as part of Puppet's reporting mechanism and used to determine whether or not a signal should be send to resources with a notify relationship. Because Puppet tracks whether or not a resource has made a change, it's entirely possible to write code that is functionally idempotent, without meeting the criteria of idempotent from Puppet's resource model.

For example, the following code is functionally idempotent, but will report as having changed state with every Puppet run.

Example 2-14. Puppet code that will report as non-idempotent

```
exec { 'grep -q /bin/bash /etc/shells || echo /bin/bash >> /etc/shells':
  path     => '/bin',
  provider => 'shell',
}
```

Puppet's idempotence model relies on a special aspect of it's resource model. For every resource, Puppet first determines that resource's current state. If the current state does not match the defined state of that resource, Puppet invokes the appropriate methods on the resources native provider to bring the resource into conformity with the desired state. In most cases, this is handled transparently, however there are a few exceptions that we will discuss in their respective chapters. Understanding these cases will be critical in order to avoid breaking Puppet's simulation and reporting models.

This example will report correctly:

Example 2-15. Improved code that will report as Idempotent

```
exec { 'echo /bin/bash >> /etc/shells':
  path   => '/bin',
  unless => 'grep -q /bin/bash /etc/shells',
}
```

In this case, unless provides a condition Puppet can use to determine whether or not a change actually needs to take place.

Using condition such as unless and onlyif properly will help produce safe and robust exec resources. We will explore this in depth in Chapter 6

A final surprising example is the notify resource, which is often used to produce debugging information and log entries.

Example 2-16. The Notify resource type

```
notify { 'example':
  message  => 'Danger, Will Robinson!'
}
```

The notify resource generates an alert every time its invoked, and will always report as a change in system state.

Run Level Idempotence

Puppet is designed to be idempotent both at the resource level and at the run level. Much like resource idempotence means that a resource applied twice produces the same result, run level idempotence means that invoking Puppet multiple times on a host should be safe, even on live production environment.

You don't have to run Puppet in enforcing mode in production.

Run level idempotence is a place where Puppet's idea of change is just as important as whether or not the resources are functionally idempotent. Remember that before performing any configuration change, Puppet must first determine whether or not the resource currently conforms to policy. Puppet will only make a change if resources are

in an inconstant state. The practical implication is that if Puppet does not report having made any changes, you can trust this is actually the case.

In practice, determining whether or not your Puppet runs are truly idempotent is fairly simple: If Puppet reports no changes upon it's second invocation on a fresh system, your Puppet code base is idempotent.

Because Puppet's resources tend to have side effects, it's much possible (easy) to break Puppet's idempotence model if we don't carefully handle resource dependencies.

Example 2-17. Ordering is critical for run-level idempotence

```
package { 'httpd':
  ensure => 'installed',
}

file { '/etc/httpd/conf/httpd.conf':
  ensure => 'file',
  content => template('apache/httpd.conf.erb'),
}

Package['httpd'] ->
File['/etc/httpd/conf/httpd.conf']
```

The *file* resource will not create paths recursively. In Example 2-17, the httpd package must be installed before the httpd.conf file resource is enforced; and it depends on the existence of `/etc/httpd/conf/httpd.conf`, which is only present after the httpd package has been installed. If this dependency is not managed, the `file` resource becomes non-idempotent; upon first invocation of Puppet it may throw an error, and only enforce the state of httpd.conf upon subsequent invocations of Puppet.

Such issues will render Puppet convergent. Because Puppet typically runs on a 30 minute interval, convergent infrastructures can take a very long time to reach a converged state.

There are a few other issues that can render Puppet non-idempotent

Non-deterministic code

As a general rule, the Puppet DSL is deterministic, meaning that a given set of inputs (manifests, facts, exported resources, etc) will always produce the same output with no variance.

For example, the language does not implement a `random()` function; instead a `fqdn_rand()` function is provided that returns random values based on a static seed (the host's fully qualified domain name.) This function is by it's very nature not cryptographically secure, and not actually random at all. It is however useful for in cases where true randomness is not needed, such as distributing the start times of load intensive tasks across the infrastructure.

Non-deterministic code can pop up in strange places with Puppet. A notorious example is Ruby 1.8.7's handling of hash iteration. The following code is non-deterministic with Ruby 1.8.7; the output will not preserve the original order and will change between runs:

Example 2-18. Non-deterministic hash ordering with Ruby 1.8.x

```
$example = {
  'a' => '1',
  'b' => '2',
  'c' => '3',
}

alert(inline_template("<%= @example.to_a.join(' ') %>\n"))
```

Another common cause of non-deterministic code pops up when our code is dependent on a transient state.

Environment induced non-determinism.

```
    file { '/tmp/example.txt':
      ensure  => 'file',
      content => "${::servername}\n",
    }
```

Environment induced non-determinism will not be idempotent if you have a load balanced cluster of Puppet Masters. The value of `$::servername` changes depending on which master compiles the catalog for a particular run.

With non-deterministic code, Puppet loses run level idempotence. For each invocation of Puppet, some resources will change shape. Puppet will converge, but it will always report your systems as having been brought into conformity with it's policy, rather than being conformant. As a result, it's virtually impossible to determine whether or not changes are actually pending for a host. It's also more difficult to track what changes were made to the configuration, and when they were made.

Non deterministic code also has the side effect that it can cause services to restart due to Puppet's notify behavior. This can cause unintended service disruption.

Stateless

Puppet's client / server API is stateless, and with a few major (but optional) exceptions, catalog compilation is a completely stateless process.

A stateless system is a system that does not preserve state between requests; each request is completely independent from previous request, and the compiler does not need to consult data from previous request in order to produce a new catalog for a node.

Puppet uses a RESTful API over HTTPS for client server communications.

With master/agent Puppet, the Puppetmaster need only have a copy of the facts supplied by the agent in order to compile a catalog. Natively, Puppet doesn't care whether or not this is the first time it's generated a catalog for this particular node, nor whether or not the last run was successful, or if any change occurred on the client node during the last run. The nodes catalog is compiled in it's entirety every time the node issues a catalog request. The responsibility for modeling the current state of the system then rests entirely on the client, as implemented by the native resource providers.

IF you don't use a puppetmaster or have a small site with a single master, statelessness may not be a huge benefit to you. For medium to large sites however, keeping Puppet stateless is tremendously useful. In a stateless system, all Puppetmasters are equal. There is no need to synchronize data or resolve conflicts between masters. There is no locking to worry about. There is no need to design a partition tolerant system in case you lose a data-center or data-link, and no need to worry about clustering strategies. Load can easily be distributed across a pool of masters using a load balancer or DNS SRV record, and fault tolerance is as simple as ensuring nodes avoid failed masters.

It is entirely possible to submit state to the master using custom facts or other other techniques. It's also entirely possible to compile a catalog conditionally based on that state. There are cases where security requirements or particularly idiosyncratic software will necessitate such an approach. Of course, this approach is most often used when attempting to write non-declarative code in Puppet's DSL. Fortunately, even in these situations, the Server doesn't have to actually store the node's state between runs; the client simply re-submits it's state as part of it's catalog request.

If you keep your code declarative, it's very easy to work with Puppet's stateless client/server configuration model. IF a manifest declares that a resource such as a user should exist, the compiler doesn't have to be concerned with the current state of that resource when compiling a catalog. The catalog simply has to declare a desired state, and the Puppet agent simply has to enforce that state.

Puppet's stateless model has several major advantages over a stateful model:

- Puppet scales horizontally
- Catalogs can be compared
- Catalogs can be cached locally to reduce server load

It is worth noting that there are a few stateful features of Puppet. It's important to weigh the value of these features against the cost of making your Puppet infrastructure stateful, and to design your infrastructure to provide an acceptable level of availability and fault tolerance. We will discuss how to approach each of these technologies in upcoming chapters, but a quick overview is provided here.

Sources of State

In the beginning of this section, I mentioned that there are a few features and design patterns that can impose state on Puppet catalog compilation. Let's look at some of these features in a bit more depth.

Filebucketing

File-bucketing is an interesting and perhaps under-appreciated feature of the File resource type. If a file-bucket is configured, the file provider will create a backup copy of any file before overwriting the original file on disk. The backup may be bucketed locally, or it can be submitted to the Puppetmaster.

Bucketing your files is useful for keeping backups, auditing, reporting, and disaster recovery. It's immensely useful if you happen to blast away a configuration you needed to keep, or if you discover a bug and would like to see how the file is changed. The Puppet enterprise console can use file-bucketing to display the contents of managed files.

File-buckets can also be used for content distribution, however using a file-bucket this way creates state. Files are only present in a bucket when placed there; either as a backup from a previous run, or by the static_compiler terminus. Placing a file in the bucket only happens during a Puppet run, and Puppet has no internal facility to synchronize buckets between masters. Reliance upon file buckets for content distribution can create problems if not applied cautiously. It can create problems when migrating hosts between data-centers, when rebuilding masters. Use of file bucketing in your modules can also create problems during local testing with `puppet apply`.

Exported Resources

Exported resources provide a simple service discovery mechanism for Puppet. When a puppetmaster or agent compiles a catalog, resources can be marked as exported by the compiler. Once the resources are marked as exported, they are recorded in a SQL database. Other nodes may then collect the exported resources, and apply those resources locally. Exported resources persist until they are overwritten or purged.

As you might imagine, exported resources are, by definition stateful and will affect your catalog if used.

We will take an in depth look at PuppetDB and exported resources in (to come). For the time being, just be aware that exported resources introduce a source of state into your infrastructure.

In this example, a pool of web-servers export their pool membership information to a haproxy load balancer, using the puppetlabs/haproxy module and exported resources.

Example 2-19. Declaring state with an exported resource

```
include haproxy

haproxy::listen { 'web':
  ipaddress => $::ipaddress,
  ports     => '80',
}

Haproxy::Balancermember <<| listening_service == 'web' |>>
```

Example 2-20. Applying state with an exported resource

```
@@haproxy::balancermember { $::fqdn:
  listening_service => 'web',
  server_names      => $::hostname,
  ipaddresses       => $::ipaddress,
  ports             => '80',
  options           => 'check',
}
```

This particular example is a relatively safe use of exported resources; if PuppetDB for some reason became unavailable the pool would continue to work; new nodes would not be added to the pool until PuppetDB was restored. TODO: Validate what I just said is true given the internal use of concat on this module…

Exported resources rely on PuppetDB, and are typically stored in a PostgreSQL database. While the PuppetDB service is fault tolerant and can scale horizontally, the PostgreSQL itself scales Vertically and introduces a potential single point of failure into the infrastructure. We will discuss approaches to scale and mitigate risk in (to come)

Hiera

Hiera is by design a pluggable system. By default is provides JSON and YAML backends, both of which are completely stateless. However, it is possible to attach Hiera to a database or inventory service, including PuppetDB. If you use this approach, it can introduce a source of state into your Puppet Infrastructure. We will explore Hiera in depth in [Link to Come].

Inventory and Reporting

The Puppet Infrastructure maintains a considerable amount of reporting information pertaining to the state of each node. This information includes facts about each node, detailed information about the catalogs sent to the node, and the reports produced at the end of each Puppet run. While this information is stateful, this information is not typically consumed when compiling catalogs. We'll be taking a close look at inventory and reporting services in [Link to Come]

There are plug-ins to Puppet that allow inventory information to be used during catalog compilation, however these are not core to Puppet.

Custom Facts

Facts themselves do not inherently add state to your Puppet manifests, however they can be used to communicate state to the Puppetmaster, which can then be used to compile conditional catalogs. We saw an example of this in [Link to Come] when discussing non-declarative code. Using facts in this way does not create the scaling and availability problems inherent in server site state, but it does create problems if you intend to use cached catalogs, and it does reduce the effectiveness of your reporting infrastructure.

Summary

In this chapter, we reviewed the major design features of Puppet's language, both in terms of the benefits provided by Puppet's language, and the restrictions it's design places on us. Future chapters will provide more concrete recommendations for the usage of Puppet's language, overall architecture of Puppet, and usage of Puppet's native types and providers. Building code that leverages Puppet's design will be a major driving force behind may of the considerations in future chapters.

Takeaways from this chapter:

- Puppet is declarative, idempotent, and stateless
- In some cases violation of these design ideals is unavoidable
- Write declarative, idempotent, and stateless code whenever possible

CHAPTER 3

Code and Data; High Level Design

In this chapter, we will look at the primary structures of a Puppet code-base, and discuss how the design of your site will impact module maintenance, code reuse, debugging, and scaling.

Here, we introduce the major systems of Puppet, including Hiera and Puppet Modules as well as several common design patterns. We also attempt to broadly classify the common kinds of code and data-sources seen with puppet into categories, so that we can discuss the most appropriate way of handling each concern.

Organization of your Puppet infrastructure becomes critical as your environment grows; correct placement of your data will improve the flexibility of your code-base and control the disruption caused by environmental changes. Correct separation of your code will help to create small problem domains that are much easier to debug and improve. Correct organization and design will help promote code re-use as you bring up new applications, and will reduce the impact of changes to business logic and application stack design driven by ever changing business requirements.

As with Chapter 2 this document focuses primarily on "why" rather than "how," and will help lay a foundation for many of the recommendations in coming chapters.

Code and Data

When I discuss Puppet architecture with new clients, I generally define 5 major categories of code and data typically seen in the infrastructure.

1. Application logic
2. Business logic (site specific logic)
3. Site specific data
4. Node specific data (node classification)

5. Service discovery data (optional)

These terms are not universally accepted, but are useful for discussing the organization of your code, and how it maps to the various features and design patterns of Puppet.

These concepts loosely map to a number of Puppet features and design patterns:

- The logic to manage an application, service, or subsystem can usually be contained in a single Puppet module.
- Business logic is usually contained within roles and profiles
- Site specific data is usually stored in Hiera
- Node specific data is usually managed using some sort of console and/or stored in a database
- Service data is usually managed with Hiera, PuppetDB, or a service discovery system

Application Logic

Application logic is the logic to manage a single service, application, or subsystem. As a general rule, application logic will manage a single named product and some of it's dependencies. E.g. Apache, Nginx, and IIS are applications. A web-server is a higher level abstraction that will probably contain multiple applications, services, and their dependencies.

The application logic for Apache will contain resources to deploy Apache's packages, manage Apache's configuration files, and ensure that the Apache service is running. The application logic may also contain code to setup virtual hosts that can be declared by other modules or as part of an application stack.

Application logic may also contain conditional logic and data to handle platform specific implementation details. For example, an Apache module should contain platform specific logic to ensure that the correct package is installed (E.g. `httpd` on RedHat and `apache2` on Debian platforms) that the configuration file is installed to the correct location and contains the correct platform specific configurations, and that the correct service name is used when managing the apache service state.

The Apache module contains a lot of data about Apache; package names, configuration details, file locations, default port numbers, docroot, and file ownership information. It may even include generic usernames, passwords, hostnames, and contact information for the cases where the user simply wants to test the module or isn't concerned about such details.

In some cases, the application may include some data that tends to be universal but may be overridden in a site specific way. For example, the NTP module includes a set of publicly available NTP servers that will be used if site specific servers are not configured.

Business Logic

Business logic also manages applications, but typically at a higher level. At this level, we aren't managing Apache, we are managing a web-server. At this level, we aren't concerned with package names or the platform specific implementation details of our services; instead we're concerned with what components are deployed to our web-server. The implementation details of each components are abstracted away as application code.

Business logic contains code that ensures our web-servers have the appropriate HTTP server installed with the necessary extensions, and that the module deploying our website is applied. Business logic may still contain conditional code; e.g. we might declare that Windows hosts get the IIS and Linux hosts get the Apache Module. The database names used by your applications are site specific data.

Business logic may pass higher level data on to your applications. For example, your application may require that web-server listen on a nonstandard port.

Site Specific Data

Site specific data is the data that is proprietary to your site, and isn't fundamental to any of your applications. The usernames and passwords your applications use to authenticate to their respective database servers are site specific data. Your internal YUM repositories and NTP servers are site specific data.

In some cases, Puppet resources may also be handled as site specific data. It's very easy to store users, packages, or yum repositories as a data structure and convert that data into resources inside a profile. The key here is that these resources need to be site specific; the fact that mlocate and tree should be installed on all your hosts is site specific. The dependencies for your Apache module are business logic, and the name of the package that contains the Apache2 daemon is application logic.

Node Specific Data

Node specific data is a special case of site specific data. Node specific data is the list of hosts managed by Puppet, basic properties about those systems such as their tier, owner, or location, and a list of what Puppet should apply to that host.

While this list can be maintained with the same system that manages your site specific data (see Hiera below) it's usually separated. Node classification is incredibly transient; nodes are constantly brought up, shut down, modified, and re-assigned.

In the olden days, Nodes were typically managed using node statements within the site wide Puppet manifest. Nodes classification could be a mix of code and data with Inheritance thrown on top. In modern sites, nodes definition is handled entirely as data, and any logic that needs to be applied on top of the node classification is handled using a hierarchical data lookup if site specific data, or at a higher level as business or application specific logic.

Service Data

Service data is another subset of site specific data, and is related to node classification.

Service data is the record of what hosts provide a specific service as part of a specific application stack. This kind of data is prevalent with horizontally scaled and multi-tier applications.

For example, the classic 3-tier application stack consisting of Apache, Tomcat, and MySQL would typically have 2 pools of service data that would need to be maintained to configure every host in the stack.

One pool of application data would consist of service data for the Tomcat servers. This service data would be consumed by the Apache servers in order to configure reverse proxies. The Apache servers could use this data to load balance requests to the application servers. Service data about the Tomcat servers could also be consumed by the MySQL servers in order to define database grants. Doing so would provide much more granular control than authorizing access by user-name and password alone or by authorizing an entire subnet of hosts.

The other pool of application data would contain information about the MySQL servers. This could allow the Tomcat servers to load balance read requests across a set of read-only slaves. It might also identify the write master in a pool of database servers.

Service discovery is often handled as site-specific data and updated manually, however this approach does not facilitate auto-scaling, or automatic fail-over, and adds extra manual steps to the process of bringing up new nodes in a pool of hosts. For these and other reasons, it's typical to handle service data as a special class of data, managed using a special set of tools.

More on service discovery tools below, and in (to come)

Breaking it down

Let's take a look at how these concerns map to the features and common design patterns of Puppet.

Application Logic & Puppet Modules

The Puppet Labs documentation describes modules as "self-contained bundles of code and data." Modules serve many purposes in Puppet, and it's difficult to provide a more specific description than that while remaining concise and accurate.

For this chapter, when we discuss Modules and module design we are almost always looking at modules from the perspective of service or application management using the Puppet DSL.

With this scope in mind, Puppet modules map fairly well to the concept of application logic. A single module can easily manage a single application, service, or component of a larger system. The prerequisites, and dependencies of the service can be managed by other modules, and multiple modules can be combined into application stacks at a higher level of abstraction.

Modules are most effective when the serve a single purpose, limit dependencies, and concern themselves only with managing system state relating to their named purpose. Puppet modules are intended to be portable and reusable. A good module will usually manage a single service, and accept just enough input to be re-used with multiple application stacks. In an ideal world, our modules are entirely self contained. They provide complete functionality without creating dependencies on any other modules, and can be combined as needed to build different application stacks.

Modules need not contain or even declare all of their own dependencies; doing so almost invariably begins to embed business logic into our modules. Let's consider the example of a Java application that depends on the availability of a Java Runtime Environment (JRE.) On a modern Linux distribution, installing the default JRE is usually just a simple package resource declaration, and at first it makes sense to install a JRE as part of our application. However, there are several different JRE offerings available, and usually different releases and feature sets for each offering. If we bundle OpenJDK into our module, we would prevent our users installing Oracle JRE instead. We would force our users to use our preferred JRE and our own installation parameters, but that could create a conflict with any other module that attempts to install it's own JRE. In this case, the cleanest approach would be to document our module's Java dependency, and provide a test manifest demonstrating how to install a JRE and other dependencies along with our module. This approach allows our users to handle the JRE dependency themselves with their own business logic. It also allows our users to manage conflicts and interdependencies using their own site-specific logic.

As mentioned earlier, a good module has no interdependencies. This should not be taken as advise to eliminate module interdependencies altogether; instead this is a recommendation to create interdependencies with care. There are many cases where modules extended the features to Puppet and provide useful functionality to be used in your own code. For example, it's typical for a module to depend on puppetlabs/stdlib due to

the number of useful functions it offers. As another example, it's usually better to create a dependency on nanliu/staging than to re-implement your own archive retrieval and extraction code on a per-module basis.

Tightly scoped modules provide a lot of flexibility, but their greatest benefit is their ease of maintenance. Writing test cases for small simple modules is much easier than writing test cases for big complex modules. Debugging a problem is much simpler when you can test each module in isolation and confirm that each module's behavior is reasonable and its test cases are correct.

Modules may contain Puppet resources, Puppet code, and data relating to the module.

Let's take a look at the Puppetlabs/ntp module as an example.

- Package, File, and Service resources manage the basic components of the NTP service.
- Conditional logic provides platform specific defaults for NTP servers, service names, and file locations
- A list of default, platform specific public NTP servers
- Parameters to change the behavior of the module or set your own NTP servers
- Input validation to ensure that data supplied to the module is sane (if not correct)
- Documentation describing the features use of the module
- Test cases for ensuring correctness and stability of the module
- Meta-data that can be consumed by the Puppet Forge and Puppet Module utility

The NTP module is fairly simple as far as Puppet modules are concerned. It contains few resources, and a lot of data. However, all of this data is specific to the NTP module. You can easily use this module as is in your site either by applying it with default parameters, or you could overriding the the default parameters with your own site specific data using site-specific logic or automatic parameter lookups.

From a business logic perspective, this module would probably be part of your baseline system configuration, and would most likely be applied to every node in your infrastructure.

Although this module is concerned with service data, the list of NTP servers in your site will tend to remain fairly static. In this case, service discovery can be handled as generic site-specific data provided by Hiera.

Identifying business logic in Puppet modules

As a general rule, you are writing business logic rather than application logic when:

- Your code could create a conflict with another module that otherwise does not overlap with your module
- The thing you're including is outside the explicit scope/concern of your module
- The thing you're including could be a standalone application or service
- The thing you're including has usefulness outside your module
- The dependency you're resolving could be handled using other solutions

When you start to notice that your code meets one or more of the above criteria, consider how your module might be split into multiple modules and managed using the Roles and Profiles design pattern.

Site specific data vs application data in Puppet Modules

Modules will tend to contain a lot of data. This data takes the form of variable defaults, package and file names, hostnames, ports, files, and templates. As a guideline, the ideal module contains no data specific to your site, but does contain the necessary data to bring this application or service up in a generic way with appropriate values for supported platforms.

An example of site-specific data is a URL to download a file from an internal server. An example of application specific data is the name of a package resource that should be deployed, and the general layout of its configuration file.

There are several important reasons to keep site specific data out of your module, even if your module is completely proprietary and would never be released to the public.

One issue is that embedding data in a module module tends to create module interdependencies. When a module contains data about your site, it's tempting to de-duplicate the data by referencing data stored in other modules rather than redefining that data in each module. Doing so creates explicit interdependencies. This pattern also tends to violate the principles of interface driven design; Puppet's language doesn't offer getter methods, or the concept of private vs public variables.

Data changes. Constantly. This is by far the greatest issue created by embedding site-data in modules. If your site specific data is spread across multiple modules, simple changes to your site's configuration can suddenly require a massive effort to hunt down and update each module that might be affected by the change.

Data stored inside your modules is rarely going to be formed in a consistent way, and it will tend to be spread across a large set of files. If you are using R10k and the module-per-repo pattern, all this data may not even be stored in the same RCS repository, and you may even have to be concerned about situations where the data has to be updated in more than one revision of the module, and the module revisions need to be updated in multiple control repositories.

By designing your module to be portable, you force your site specific data to be centralized and stored outside of the module. By adhering to this simple pattern, you encourage use of node classification and Hiera.

Conversely, it is a good idea to contain application specific data entirely inside your module. If application data, such as the name of the packages your module installs or the template used to configure a service are externalized, the module becomes unusable in isolation; it can only be tested in the greater context of your infrastructure. It can't be reused without also supplying the data that lives outside the module, and it isn't going to be easily portable since that data is often stored or formulated in such a way to become non portable. Storing application data outside the module also tends to increase dependencies; if your module depends on data stored in Hiera in order to perform a basic smoke test, you must have hiera configured and your data deployed in order to simply test the module.

One caution about this recommendation: Always try to provide sane defaults for site specific data, even if those defaults aren't terribly useful in the real world. This provides significant benefits for anyone attempting to debug, test, or simply play with your module.

Business logic with Roles and Profiles

Roles and Profiles are not a design feature of Puppet, instead they are a desgin patern originally described in Craig Dunn's blog post Designing Puppet - Roles and Profiles. We will take a closer look at the Roles and Profiles design pattern in [Link to Come].

Roles and profiles have a few important design properties:

1. They describe application stacks and business logic
2. They are typically site specific

A major feature of Puppet is code reuse. Modules provide a way to model applications and services in a portable and reusable way. Application stacks can be created by combining multiple modules together in interesting ways using the roles and profiles pattern.

For example, if you wanted to setup an instance of WordPress, we would need an application stack containing the following components:

- A database server (MySQL)
- A web-server (such as Apache)
- PHP
- WordPress

While you could create a monolithic Puppet Module to configure Wordpress and all of its dependencies, such a module would be quite complex and inflexible. We could build a module that concerns itself only with the deployment of Wordpress, and relies on the official puppetlabs/apache, puppetlabs/mysql Puppet modules. Using this approach allows us to rely on the development efforts and support provided by the authors of the Apache and PHP modules (in this case, Puppet Labs) and provides the benefits of community experience and bugfixes.

By writing a module that only concerned it's self with WordPress, the consumer of your module could easily swap in arbitrary web-servers and database servers. Each module could be self contained, which would make it simpler to test each system in isolation, and help reduce the amount of code that would need to be reviewed to identify and isolate bugs.

The modular approach also facilitates code re-use. Other users will consume public Apache and MySQL modules even if they aren't interested in WordPress. By using those modules, you can leverage their shared experience, improvements, and bug fixes.

Leveraging public modules creates a new design problem: if we have two modules assigned to a host that both depend on Apache, which of those modules should be the one to own and declare it?

The answer is neither. Instead, we abstract our application stacks into a profile. The profile simply defines the modules needed to build our application stack, their ordering dependencies, and if absolutely necessary any parameters that should be passed to the modules. Because a profile is site specific, we can design our profiles to avoid conflicts. For example, we could have the profiles both include apache, and only use resource style declaration for our application vhost, via defined types. With this approach, the one node could potentially apply multiple profiles that declare Apache without issue.

Roles and Profiles vs. Node Classifiers

Roles and profiles are a design pattern. Strictly speaking, both roles and profiles are simply modules containing Puppet Manifests. What makes roles and profiles special are their purpose as an abstraction layer between your modules and your node classifier.

In the old days, this sort of thing was common:

Example 3-1. Node classification with Inheritance

```
node base {
  include ::accounts
  include ::security
  include ::repos
  class { '::ntp':
    servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
  }
}
```

```
node web inherits base {
  include ::apache
  include ::php
}

node db inherits base {
  include ::mysql
}

node www1.example.com www2.example.com inherits web { }

node www-dev1.example.com inherits web {
  include ::mysql
}

# etc.
```

This approach tended to be difficult to maintain, and had a few major limitations; node definitions do not support multiple inheritance, so it's impossible to build a development box that contains both the db and web node classifications. Node inheritance also has some very surprising behavior relating to variable inheritance and scope.

Trying to migrate this approach to an external node classifier creates some issues as well. Many web console ENCs support the concept of groups and group inheritance, however they often don't support parameterized classes, and when they do they may only support strings as a data-type.

Of course, conditional logic isn't really an option with an off-the-shelf console; if you want to install IIS on your Windows web nodes and Apache on your RHEL nodes, you need to create two groups and assign them manually.

With roles and profiles, you move node classification logic to a set of conventional puppet manifests/classes.

Example 3-2. Node Classification with Roles & Profiles

```
class profiles::base {
  include ::accounts
  include ::security
  include ::repos
  class { '::ntp':
    servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
  }
}

class profiles:web {
  include ::apache
  include ::php
}
```

```
class profiles::db {
  include ::mysql
}

class roles::webserver {
  include ::profiles::base
  include ::profiles::web
}

class roles::webserver::dev {
  include ::profiles::base
  include ::profiles::web
  include ::profiles::db
}
```

With this approach you can simply apply `::roles::webserver` or `::roles::db` using our external node classifier. We can use an extremely simple ENC. This approach also has the benefit of codifying what belongs on a pariticular type of host.

We will explore roles and profiles in a lot more depth in [Link to Come]

Hiera & Site Specific Data

Hiera is a store for your site specific data; that is, the stuff that is specific to you; data that no one else would really want to use in their own environments.

An example of site specific data is user account information, SSH keys, your read-only LDAP bind password, or the email address that needs to be stamped on all your DNS zones.

As a general rule, application defaults and application data do not belong in Hiera. For example, it might be tempting to embed platform specific package names in Hiera, because Hiera provides a simple mechanism for returning the correct value based on the `osfamily` fact. Doing so creates an explicit dependency on Hiera that might not be easy to satisfy in all situations.

Site specific data tends to change far more often than application code or business logic. New sites come up, infrastructure changes, hosts are upgraded or decommissioned. Keeping all of the data describing your site in a centralized location makes it very easy to identify anything that might be affected by an environmental change. Keeping it all in a well formed data file makes searching and manipulating the data in mass simple. Using such a database also offers a benefit that it's fairly easy to query your data using a standard set of tools; this is extremely valuable for debugging.

Although Hiera is almost ubiquitous as a datastore for Puppet, other solutions do exist, including the `extlookup()` function call.

We will look at Hiera in depth in [Link to Come]

Hiera vs. Service Discovery

Service information is a very specific form of site specific data. Service data is information about what hosts currently provide a specific service. For example, you might have a particular web application running on a pool of web-servers. In order to configure your load balancers or reverse proxies, you need to enumerate what hosts belong to that particular pool.

In a small site that changes infrequently or with fairly stable services, it's simple to embed such service data in your site data-store (Hiera.) Even with powerful service discovery data, it's not uncommon for core services such as the IP addresses of your DNS servers or the hostnames of your package repositories to be fairly static; good candidates for Hiera or another site specific data pool.

However, modern environments are rarely static. Auto-scaling pools are becoming increasingly common, and service clusters are becoming increasingly large, and distributed. In these cases, manually editing data files every time a host comes up becomes unrealistic, slow, and in many cases error prone. It also tends to drive incredible levels of configuration churn in your code repositories, which can become annoying to deal with at integration time.

If you have a complex environment, it often makes sense to use a purpose built tool to handle service discovery data. PuppetDB and exported resources offer service discovery features as part of a fairly standard Puppet Installation, however there are other solutions with their own strengths and weaknesses.

We will discuss service discovery and other ecosystem tools in (to come).

Node Classification

Node classification is the system that determines what modules (or roles) should be assigned to a specific node. Node classification typically also stores some basic properties of each node, such as the physical location of the node, the tier of the node, and the Puppet Environment assigned to that note. These properties can be consumed by Hiera to perform hierarchical lookups, or by Puppet to determine what puppet environment should be used when building a node's catalog.

There are many many options available for node classification. The Puppet Enterprise Console provides a node classifier, and there are several other mature web consoles as Foreman and Cobbler. A node classifier does not have to be a web application; Hiera can be used as a classifer. Node classification can be driven by facts. Node classification can be stored in an LDAP database, or in MongoDB. Integrating with a data-center database or data warehousing application is also fairly simple. Because classifers are simple to write, virtually anything that can store information about the nodes in your environment can be used as a node classifier.

Using an existing data store as a classifier does not preclude you from using a web console for reporting and analysis purposes.

Node classification is a form of site specific data, but it is usually handled separately from more generalized forms of site data due to it's nature.

Site specific data tends to be relatively static. Nodes may be added to your classification store every time a node is brought up or down in your infrastructure.

Node classification also tends to integrate with provisioning systems. Such systems typically maintain their own host databases and are concerned with managing many more properties of a system than Puppet requires to classify a host.

Conversely, such systems rarely provide the flexibility of the Puppet DSL for defining application stacks and setting class parameters. For example, the node classifier in Cobbler cannot set class parameters, and has limited support for defining groups of classes. The Puppet Enterprise node classifier is much fuller featured, but still cannot define class ordering relationships (this is a limitation of the Puppet Classifier API.)

Exported Resources & Service Discovery

Service discovery is the act of discovering or broadcasting what hosts offer a particular service, so that inter-node relationships can be automatically managed. We touched on service discovery earlier when we looked at the place of Hiera in your infrastructure.

Here's an example of how Service Discovery with Exported Resources can simplify monitoring a pool of web-servers.

First, we must export a nagios_host resource definition:

Example 3-3. Exporting a nagios_host resource definition

```
@@nagios_host { $::fqdn:
  ensure        => 'present',
  alias         => $::hostname,
  address       => $::ipaddress,
  use           => 'generic-host',
  check_command => 'check_http',
}
```

With each of our web-servers exporting their Nagios configuration, we can now apply those resources to our Nagios host using a resource collector. This simple example would automate the process of monitoring our web-servers using Nagios.

Example 3-4. Collecting nagios_host exported resources

```
nagios_command { 'check_http':
  command_line => '$USER1$/check_http -I $HOSTADDRESS$ $ARG1$'
}
```

```
Nagios_host <<||>>
```

This example is extremely simplified for illustrative purposes. It also makes a lot of assumptions about the configuration of your Nagios configuration. Typically we would use Nagios hostgroups, service groups and service definitions to better organize our configuration and abstract out our checks.

Exported resources provide one method for service discovery. With exported resources, a resource is defined on a node, but instead of being applied locally, it's stored in a database, and made available for other nodes to apply.

In simple sites with minimal churn, exported resources may not be necessary; you can simply enumerate service relationships in Hiera. Your nodes can consume that data, and apply it using conventional resources.

Such a solution is untenable in environments that have thousands of nodes, or constantly have new nodes being provisioned and old nodes being decommissioned; managing a constantly changing list of service data becomes burdensome, and often creates delays in node provisioning.

Exported by resources are not by any means the only option for service discovery. In fact, there are many third party solutions that provide very real benefits compared to exported resources.

With that said, exported resources are usually free. You do have to configure PuppetDB in order to leverage exported resources, but it's a good idea to setup PuppetDB regardless of whether or not you intend to use them. Exported resources also have the benefit that they are fairly simple to use if you're already familiar with Puppet, and are well supported by Puppet Labs and generally well understood by the community.

As we discussed in "Stateless" on page 24, exported resources do create state that must be maintained in our Puppet infrastructure, and all the issues associated with server-side state. Before employing exported resources, it's critical to compare the cost of maintaining state against the benefits of automating service discovery.

We can sidestep the issue of state using a 3rd party service discovery tool. We will look at 3rd party service discovery solutions in (to come).

Summary

This chapter introduced the major types of code and data you'll be working with as part of a Puppet Deployment, and discussed how each major component of puppet and design pattern can be most effectively used to to manage that code and data.

Takeaway from this chapter:

- Before writing new code, try to categorize what you are adding into one of the above classes of code and data
- Keep modules clean and modular so that they can be reused
- Contain application logic inside Puppet modules
- Manage business logic and application interdependencies using roles & profiles
- Chose a node classification tool and contain node definitions and properties there.
- Use Hiera to manage site specific data
- Consider using exported resources or another service discovery tool to manage inter-node relationships and service data

CHAPTER 4
Coding Practices

In this chapter, we will be looking at best practices related to using the Puppet Language. This chapter will focus predominantly on Puppet's conditional logic, operators, function calls, and data structures. We will focus on Resource types and classes at a high level so that we can delve deeper on those subjects in Chapter 5 when we discuss Module Design, and Chapter 6 when we explore best practices relating to a number of common resource types, and [Link to Come] when we look at code organization and inter-module relationships.

The Style Guide

Before we begin this chapter, I strongly advise you to review the Puppet Language Style Guide which covers most of the language basics. We are not going to attempt to repeat the style guide in this chapter; instead we are going to expand upon the information Puppet Labs provides in the guide, emphasizing a few key and often overlooked aspects of the guide.

The Puppet Labs style guide focuses on keeping the your code clear and easy to understand, as well as maintaining a clean revision history.s

A major benefit of following the official style guide is that you can better leverage the available lint and editor plug-ins to automatically correct and identify violations of the guide.

Be aware that recommendations in the Puppet Labs style guide may not always reflect best practices for the most recent releases of Puppet. Often, there are recommendations made to for the purposes of ensuring backwards compatibility with older Puppet re-

leases [1]. If you're planning to release code that might be used with older revisions of Puppet, you should follow those elements of the style explicitly. Otherwise, it's usually best to code to current best practices, ignoring conflicting recommendations.

Having a set style guide is invaluable for peer review. Consistent formatting allows reviewers to focus on the function of your code without spending as much time understanding the form. Simple rules such as indentation can make errors pop out. Correct use of white space, variable assignment and interpolation make the code cleaner and easier to follow. If you chose not to follow the Puppet Labs style guide, I recommend documenting and publishing a style guide for your site. When working at a site that follows it's own style, it's a good idea to adapt that style for site-wide consistency. In many cases, having one consistent *imperfect* style is much better than having multiple coding styles [2]

I strongly encourage you to deploy puppet-lint and to include it as a pre-push hook for your projects in order to encourage conformity to the style guide. We'll discuss puppet-lint more extensively in [Link to Come].

This book does not attempt to replace the official style guide; instead the material here can be viewed as a supplement to the guide.

Coding Principles

Principles are guidelines for development that will tend to improve the quality of your code when followed. Books such as *Practical Object-Oriented Design in Ruby* by Sandi Metz and *Clean Code* by Robert C. Martin go into great depth on coding practices, and do so in a way that is applicable beyond the scope of the language examples used in the book.

Many, but not all Object Oriented development principles apply to Puppet, though some principles can even be counter-productive due to the nature of Puppet's declarative language and the limitations of Puppet's DSL. In this section, we will briefly introduce some of the more useful coding principles that will be referenced throughout the book.

KISS

Keep It Simple. Good Puppet code is not clever. Good code, is simple, obvious, and boring. There is no twist, no mystery. While there are many neat tricks that can reduce the amount of code you have to write, ultimately you will spend a lot more time **reading** your code than you will writing it. Simple code is less error prone, easier to debug,

1. For example, until recently Puppet advised against using the *params.pp pattern* in the style guide in order to maintain compatibility with 2.x releases

2. This doesn't mean adopting bad practices, however.

and much easier to extend. The DRY principle is not a means-unto it's self. Code reduction is most beneficial when it eliminates potential bugs and improves readability.

Clever code tends to make sense when written, but often becomes confusing when read months later. Clever code often hides subtle bugs, and can be difficult to extend or refactor. In the worst cases, it solves unexpected problems in non-obvious ways, and bites anyone who attempts to re-factor it.

If trick improves the readability and clarity of your code by all means use it. If the neat trick has obtuse inner workings or makes you scratch your head, consider using a more conventional approach that's easier to understand.

Please remember: Code that's obvious to you may not be obvious to the person that inherits your work. Sometimes even documented design patterns and techniques can be confusing to those new to Puppet [3].

How simple can we get?

Some of our clients have only had part time support staff to maintain their site. Inexperienced staff were often expected make configuration changes in emergencies. At one site, we deployed Puppet without using Hiera or ERB templates. The result was a dead simple installation that allowed staff to make application level configuration changes with minimal knowledge of Ruby and Puppet.

Here's an example of needless complexity.

Example 4-1. Violating the KISS principle using create_resources()

```
$files = {
  '/tmp/foo' => {
    'content' => 'foo',
  },
  '/tmp/bar' => {
    'content' => 'bar',
  }
  '/tmp/baz' => {
        'ensure' => 'link',
        'target' => '/tmp/foo',
  }
}

$defaults = {
        ensure => 'file',
        mode   => '0644',
}
```

3. the benefit of documented tricks is that they tend to be well understood and well explained.

```
create_resources('file', $files, $defaults)
```

Example 4-1 has no benefits over using conventional file resource declarations with (or even without) resource defaults. See "Don't Repeat Yourself (the DRY principle)" on page 52 for more examples.

While this example is entirely written using the Puppet DSL, it is fairly common to see code like this implemented when a developer wishes to store resources in Hiera. There are a few specific cases where such an approach is entirely valid. Those considerations are discussed in [Link to Come].

The Single Responsibility Principle

Resources, data Structures, classes, and function calls in Puppet should have one responsibility, and no more than one responsibility. Ideally, you should be able to describe anything you build in Puppet without using the word *and*.

For example:

- This class manages Apache's packages
- This case statement sets platform specific defaults
- This conditional determines the correct resource ensure state
- This function call validates URLs
- This resource downloads a file
- This hierarchy supplies location specific data.
- This module installs Tomcat
- That other module is responsible for installing Java
- This profile builds a web-server
- This role builds our 3-tier application stack on a single machine for our developers

The Single Responsibility Principle applies to data structures, control structures, and classes. When applied correctly, it helps create modular, extensible, and reusable code that's easier to debug. It also helps ensure that you have minimized side-effects of your code.

Example of things that might violate the Single Purpose Principle:

- This module installs Java and Tomcat
- This conditional sets the ensure state and determines the correct version.
- This resource downloads and extracts a tar-ball
- This class configures Nginx and installs a baseline environment

- This module encrypts data and syncs the system clock

A Puppet module that follows the single purpose principle is easy to re-use, maintain, and extend. That module doesn't include it's own version of Java; it allows you to chose your own Java binary, satisfying that dependency in domain logic. It's organized into subclasses with a single purpose, making it easier to understand the flow of the module and the relationships between the module's components.

Because everything has a single responsibility, data structures and function calls can be given a clear concise name that clearly communicates it's intent and purpose. Testing is also simplified because the behaviors of each individual function call or values of each data structure are simple.

Modular code is much less likely to be re-written as your site grows and changes. For example, if you have a monolithic *base* module containing policy enforcement, user accounts, drivers, and packages, you may run into problems if you decide that a new type of host needs to have a non-standard *base*. In this scenario, you could probably parameterize your module to support this special exception, but doing so could potentially break nodes that would not otherwise be affected by the change. However, if you were to build your base from several modules and codify it using a profile, you could easily create a new profile for your specific case with absolutely no risk to existing hosts.

Seperation of Concerns

Separation of concerns is the creation of loosely integrated, modular code. It is is strongly related to interface driven design and the single responsibility principle.

Many of the concepts discussed early in this book service the separation of concerns; understanding the purpose and use of Hiera, Roles & Profiles, and Modules helps to create clear lines between component of a complete Puppet code base.

The concept of modularity should be emphasized. With modular Puppet code, dependencies are minimized; testing one of your modules should mean deploying that module, `stdlib`, and perhaps one or two other dependencies (if necessary.) In sites that do not separate concerns, it's very common to find situations where it's impossible to test a module without deploying a large portion of the entire site's code base. With modular code, you can learn the system and debug the system by isolating one component at a time. With a highly integrated code-base, understanding and troubleshooting any single component of the system involves troubleshooting and understanding the whole system. This isn't usually a problem in a program with a few hundred lines of code, but can become a major issue in a site with thousands of resources.

Use of Puppet modules doesn't necessary mean your code is modular

There is a huge difference between using Puppet modules and writing modular code. It is absolutely possible to write non-modular code using modules. As a general rule, if your module calls other modules using the include or contain function, your module is no longer modular.

This should be not taken to mean that your module cannot have dependencies. You might depend on the apt module to install packages on Debian hosts, or one of Puppet's many utility modules such as concat and staging.

Your module may also be dependent by necessity on functionality provided by other modules. For example, your web service might need to have a web-server. The key to modularity in this case is that while your module might need a web-server, your module doesn't concern it's self with deploying a web-server. Your module allows that dependency to be handled at a higher layer of abstraction. In this example, a profile would install the web-server and your module. You would also provide a test manifest using an example web-server to satisfy the dependency for the purpose of testing and experimentation.

Separation of concerns means that the purpose of each chunk of code is well defined and clearly thought out. It means asking questions such as "does this belong in Hiera or in a profile?" "Should this be module logic or profile logic?" "Should this be one module or two?"

Separation of concerns usually requires careful thought to where a particular bit of code or a resource belongs. For example, does a VMWare specific time setting belong in your NTP module or in your VMWare module? Footnote:[In this scenario, I would recommend putting the resource inside the NTP module, or possibly even splitting it into it's own module (if necessary.) The main reason being that the code is almost certainly going to be more dependent on the NTP code than on the VMWare code. If the code is highly dependent on both, the correct solution may be to write an additional module and handle integration between your NTP module and VMWare module at a higher level of abstraction, such as in a profile.]

Separation of concerns can be painful when building a new site. Often, referencing an existing variable from another module seems like a quick and easy way to feed data into your new module. It avoids duplicating data, and avoids re-factoring the data to use Hiera. It avoids the need to re-think the way your site is structured. Unfortunately, this is often a trap; while such solutions are easy to implement, they tend to be difficult to debug and maintain, and often constrain design and hamper refactoring further down the road.

Separation of concerns often requires a lot of long term thinking, and a mentality that long term support is as important as the rapid implementation of new code.

Interface Driven Design

Your Puppet modules should interact with each other using well defined interfaces. In most cases, this means you interact with classes via class parameters and class based resource relationships rather than using fully qualified variables and direct resource relationships.

Because Puppet is a declarative language and does not offer getter or setter methods, Interface driven design tends to enforce a structure to the way data flows through your site. Data originates from sources such as Hiera, ENCs, and your site.pp manifest up through your roles and profiles and finally into your modules. This all happens using parameterized classes following the path of module inclusion. With this approach, data isn't passed laterally between classes.

Using class parameters as your primary interface has some major troubleshooting benefits. Information about the classes and class parameters applied to a node are sent to that node as part of its catalog. Using the catalog information, you can isolate and debug a single module, without worrying so much about **how** it's parameters are being generated. You can also use the parameters to better understand how data flows through your manifests.

Example 4-2 is an example of code that abuses fully qualified variables in violation of the principle of interface driven design.

Example 4-2. Vhost template violating the principle of Interface Driven Design

```
class myapp {
  include apache
  $vhost_dir = $apache::vhost_dir

  file { "${vhost_dir}/myapp.conf":
    ensure  => 'file',
    content => template('myapp/vhost.conf.erb')
  }
}
```

Example 4-3 is functionally the same as Example 4-2 except that it uses a defined type as an interface:

Example 4-3. Interface Driven Design vhost template

```
class myapp {
  include apache

  apache::custom_vhost { 'myapp':
    ensure  => 'present',
```

```
    content => template('myapp/vhost.conf.erb'),
  }
}
```

In Example 4-2, we reach into the Apache module and grab the contents of an internal variable. If it is not a parameter or otherwise documented interface of the class, there is no guarantee the variable or it's contents will be available or consistent between releases [4].

Rather than querying the apache module to determine the directory for vhost files, in Example 4-3 we create a defined type inside the apache module which accepts our template as a parameter. With this approach, we can declare a stable interface into the apache module, while retaining our ability to restructure the implementation of the module. And because apache::custom_vhost is internal to the apache module, it can access other internal variables without violating the principles of interface driven design.

Although Apache is used in this example, this code will not work with the puppetlabs/apache module. As of this writing, puppetlabs/apache does not provide an apache::custom_vhost defined type accept content as a parameter of the apache::vhost defined type. If you wish to apply this concept to the puppetlabs/apache module, I'd recommend forking and extending the existing module.

Again, I should emphasize that there isn't a technical reason you can't simple access variables from other modules using fully qualified variable names. The advisory against that technique is to improve the quality of your code, facilitate debugging and analysis, and to avoid headaches as you re-factor your code.

Don't Repeat Yourself (the DRY principle)

The DRY principle suggests that if you find yourself writing the same code more than once, it should be abstracted into a function call or or defined type.

You cannot declare function or method calls natively in the Puppet DSL, but Puppet does provide defined types, inheritance, default values, and native functions that can be used to reduce repetition in your code.

Of all the coding practices, DRY is the most subjective. In some cases, reducing repetition in your code can obfuscate your code, reducing the code quality rather than enhancing it. I strongly advise preferring readability over DRY in any case where the principles conflict.

Consider the following example:

4. If you insist on doing this, remember that parameters tend to be fairly sable and they are part of a module's interface.

Example 4-4. Not very DRY resource declarations

```
file { '/etc/puppet/puppet.conf':
  ensure => 'file',
  content => template('puppet/puppet.conf.erb'),
  group  => 'root',
  mode   => '0444',
  owner  => 'root',
}

file { '/etc/puppet/auth.conf':
  ensure => 'file',
  content => template('puppet/auth.conf.erb'),
  group  => 'root',
  mode   => '0444',
  owner  => 'root',
}

file { '/etc/puppet/routes.yaml':
  ensure => 'file',
  content => template('puppet/routes.yaml.erb'),
  group  => 'root',
  mode   => '0444',
  owner  => 'root',
}
```

This example could be made more DRY using a resource default:

Example 4-5. Resource declarations with resource defaults

```
File {
  ensure => 'file',
  owner  => 'root',
  mode   => '0444',
  group  => 'root',
}

file { '/etc/puppet/puppet.conf':
  content => template('puppet/puppet.conf.erb')
}

file { '/etc/puppet/auth.conf':
  content => template('puppet/auth.conf.erb')
}

file { '/etc/puppet/routes.yaml':
  content => template('puppet/routes.yaml.erb')
}
```

In this case, readability is improved using a resource default; the code is condensed, and exceptions are highlighted.

Resource defaults should only be used in classes that do not declare or include any other classes. Resource defaults are inherited, and often produce surprising behavior when unexpectedly inherited into new contexts.

This example could be further simplified using a defined type:

Example 4-6. Resource declarations with a defined type

```
$files = [
  '/etc/puppet/puppet.conf',
  '/etc/puppet/auth.conf',
  '/etc/puppet/routes.yaml',
]

puppet::file { $files: }
```

Although this approach reduces verbosity, the resulting behavior relies on the internal implementation of the *Puppet::File* defined type, which I've excluded from this example to highlight the inherent problem with this approach. Because of the way the Puppet class loader works, the defined type used in this case will be declared in it's own manifest. While this isn't a huge hurdle, it means that you'll have yet another tab open while attempting to extend or debug this code, and it could result in some ugly bugs if you decide down the road to use resource defaults in this class, or you make an invalid assumption about the implementation of the defined type.

See "The puppet::file defined type" on page 80 if you're interested in seeing the implementation of `puppet::file`.

With Puppet 4 or with the future parser enabled, you can accomplish the same thing using an each loop. This example has an implicit dependency on the `basename()` function call from `puppetlabs/stdlib`.

Example 4-7. Resource declarations with an each loop

```
$files = [
  '/etc/puppet/puppet.conf',
  '/etc/puppet/auth.conf',
  '/etc/puppet/routes.yaml',
]

$files.each |$file| {
  file { $file:
    ensure => 'file',
    content => template("puppet/${basename($file)}"),
    group  => 'root',
    mode   => '0444',
    owner  => 'root',
```

```
  }
}
```

If you're already using Puppet 4, this approach is preferable to using a defined type; by keeping the resource declaration logic and iterator in the same file and context as the resource declaration, the code is better contextualized, and we reduce our problem space when trying to fix a bug or extend our code.

In practice, I find Example 4-5 to be the most readable approach, despite being less DRY than Example 4-6 or Example 4-7. This is a case where simplicity wins out over other considerations.

There are a few cases where we can produce very clean and DRY code using auto-requires and array resource declaration:

Example 4-8. Directory creation using arrays

```
$directories = [
  '/tmp',
  '/tmp/example1',
  '/tmp/example2',
  '/tmp/example1/foo',
  '/tmp/example1/bar',
]

file { $directories:
  ensure => 'directory',
  mode   => '0755',
}
```

General Coding Recommendations

In the following sections, we will discuss specific coding recommendations.

This section is designed to provide very generalized recommendations. You will find additional coding practices in Chapter 5 and Chapter 6.

The balance of Code and Resources

Puppet code is best when the bulk of your manifests are comprised of resource declarations rather than logic. Although Puppet boilerplate, conditionals, input handling, and variable declaration is absolutely necessary, the Puppet language at its core works best when it can be treated as a simple collection of data about your systems.

Puppet is often described as an "infrastructure as code" tool rather than an "infrastructure as data" tool. At their heart however, native Puppet types are simply well-formed data declaring resource states.

Compare the following examples:

Example 4-9. Code

```
case $::osfamily {
  'RedHat': {
        exec { 'install_apache':
      command => '/usr/bin/yum install -y nmap',
      unless  => '/bin/rpm -q nmap',
    }
  }
  'Debian': {
    exec { 'install_apache':
      command => '/usr/bin/apt-get install -y nmap',
      unless  => '/usr/bin/dpkg -p nmap',
    }
  }
}
```

Example 4-10. Data

```
package { 'nmap':
  ensure => 'installed',
}
```

When you find your modules becoming very heavy with conditional logic, case statements, and variable handling code take a moment to re-assess your approach. Is there a way to re-factor your code to be more declarative?

There isn't always a one-size-fits-all approach to reducing conditional logic within your code, rather there are a number of strategies and approaches that often need to be employed.

Reinventing the Wheel

In my professional experience, attempts to re-invent and existing tool using the Puppet DSL are the most common cause of code heavy Puppet manifests. For example, it can take several resources and a lot of conditional logic to download, cache, and extract a tar-ball, where the same process is a single resource with no logic using a package.

Puppet is not a software packaging tool; there are literally 3 dozen package providers for this purpose. Puppet is also not in it's heart an orchestration tool. There are cases where using Puppet this way is unavoidable, but the quality of code tends to suffer for it, and the complexity of code tends to increase dramatically.

Conditional Logic

As we saw in Example 4-9, it's often tempting to wrap resources inside of conditional logic. Doing so often results in a lot of duplication in your code, and often compromises readability. Consider the following example:

Example 4-11. Resources embedded in conditional logic

```
case $::osfamily {
  'RedHat': {
    package { 'httpd':
      ensure => 'installed',
    }
  }
  'Debian': {
    package { 'apache2':
      ensure => 'installed',
    }
  }
}
```

Ultimately, `Package['httpd']` and `Package['apache2']` describe the same resource for two different platforms. The same code is simpler and it's purpose much cleaner using conditional logic to declare a variable rather than conditional logic to declare a resource.

Example 4-12. Use of variable declaration to simplify resource management

```
$package_name => $::osfamily ? {
  'RedHat' => 'httpd',
  'Debian' => 'apache2',
}

package { 'apache':
  ensure => 'installed',
  name   => $package_name,
}
```

Example 4-12 is somewhat more DRY than Example 4-11 and better communicates its intent. It also allows use of a selector rather than a case statement. I find that selectors are often cleaner than case statements for such simple cases.

This concept extends to resources that might be optional within a catalog. Take for example, a puppetmaster module that optionally creates an autosign.conf file:

Example 4-13. Conditionally adding a resource to a catalog

```
class puppetmaster (
  $autosign = false,
) {
  if $autosign {
```

```
    file { '/etc/puppet/autosign.conf':
      ensure  => 'file',
      content => template('puppetmaster/autosign.conf.erb'),
    }
  }
}
```

Example 4-14. conditionally managing resource state

```
class puppetmaster (
  $autosign = false,
) {
  $autosign_ensure = $autosign ? {
    true    => 'file',
    'true'  => 'file',
    default => 'absent',
  }

  file { '/etc/puppet/autosign.conf':
    ensure  => $autosign_ensure,
    content => template('puppetmaster/autosign.conf.erb'),
  }
}
```

Although Example 4-14 seems to be slightly more complex than Example 4-13, the latter approach is significantly more functional.

The major advantage of the second approach is that if at some point we decide we wish to disable auto-signing on this master, Puppet will correctly remove the `autosign.conf` file. In the first example, the file simply becomes unmanaged; if it's already present on the system, it will remain present.

The use of a selector also improves handling of input. Our second case will correctly evaluate the string `'false'` as a `false` value, where the first example would evaluate the string `'false'` as a `true` value.

There are some cases where it makes absolute sense to wrap a resource in conditional logic:

Example 4-15. OS specific resources

```
if $::osfamily == 'Debian' {
  file { "/etc/apache2/sites-enabled/${title}.conf":
    ensure => 'link',
    target => "../sites-available/${title}.conf",
  }
}
```

This resource would be nonsensical on a RedHat based system because RedHat does not manage VirtualHosts using symbolic links. In this case, conditionally adding the

resource to the catalog is the correct approach. We might still add a variable to manage the ensure property of this resource so that it can be removed if desired.

Selectors

Although selectors can be embedded in resource declarations, doing so almost always produces obtuse code.

Example 4-16. Selectors embedded in resources

```
class example (
  $ensure = 'present',
) {
  service { example:
    ensure => $ensure ? {
            'absent' => 'stopped',
            default  => 'running',
    },
    enabled => $ensure ? {
            'absent' => false,
            default  => true,
    },
  }

  file { '/tmp/example.txt':
    ensure => $ensure ? {
            'absent' => 'absent',
            default  => 'file',
    },
   content => template('example/example.txt.erb'),
  }
}
```

This same example is significantly improved by using selectors to declare a variable that is then applied by the resource, such as in Example 4-17

Example 4-17. Variable declaration and selectors

```
class example (
  $ensure = 'present',
) {

  $service_ensure = $ensure ?
        'absent' => 'stopped',
          default  => 'running',
  }

  $service_enable = $ensure ? {
    'absent' => false,
    default  => true,
  }
```

```
  $file_ensure = $ensure ? {
    'absent' => 'absent',
    default  => 'file',
  }

  service { example:
    ensure => $service_ensure,
    enable => $service_enable,
  }

  file { '/tmp/example.txt':
    ensure  => $file_ensure,
    content => template('example/example.txt.erb'),
  }
}
```

In this case, the code would be further improved using a case statement.

Example 4-18. Using a case statement for variable declaration

```
class example (
  $ensure = 'present',
) {
  case $ensure {
    'present': {
      $service_ensure = 'running'
      $service_enable = true
      $file_ensure    = 'file'
    }
    'absent': {
      $service_ensure = 'stopped'
      $service_enable = false
      $file_ensure    = 'absent'
    }
  }

  service { example:
    ensure  => $service_ensure,
    enable => $service_enable,
  }

  file { '/tmp/example.txt':
    ensure => $file_ensure,
    content => template('example/example.txt.erb'),
  }
}
```

Another solution would be to embed resources inside the case statement, but as discussed in "The balance of Code and Resources" on page 55. Such an approach is not advised.

Variables

This section covers a number of best practices relating to use of variables in the Puppet DSL.

Variable Naming

Correct naming of variables is very important for both usability and compatibility.

Puppet 4 introduces new restrictions on variable naming, the most major of which is that you can no-longer use capital letters in your variable names.

The following guidelines are recommended:

- Begin variable names with an underscore or lower case letter
- Subsequent characters may be lowercase letters, numbers, or underscores
- Favor descriptive variable names over terse variable names

Most identifiers in Puppet loosely follow Ruby symbol naming restrictions; the identifier cannot begin with a number, and cannot contain dashes. Variables are slightly more forgiving than class names, however I would advise erring on the side of using more restrictive conventions.

Referencing Variables

The following guidelines are recommended when referencing a variable:

- Avoid variable inheritance
- Local variables should be unqualified
- Global variables should be fully qualified
- Reference facts using the `$facts[]` array
- Avoid referencing undefined variables

Variable Inheritance

Versions of Puppet since 3.0 severely limit variable inheritance. Prior to Puppet 3.0, variables would be inherited between scopes. Sites that leveraged variable inheritance would somewhat reduce the amount of variable declarations and scope calls in their code, at the cost of introducing very difficult to maintain and debug code.

Puppet 3.0 still permits a few special cases of variable inheritance. One such case is the use of class inheritance. If one class inherits another class, the parent class' variables are inherited into the child class.

Example 4-19. Class inheritance and variable inheritence

```
class parent {
  $foo = 'alpha'
  $bar = 'beta'

  alert($foo) # Prints 'alpha'
  alert($bar) # Prints 'beta'
}

class child inherits parent {
  $foo = 'delta'

  alert($foo) # Prints 'delta'
  alert($bar) # Prints 'beta'
}
```

Class inheritance is strongly counter-advised, but has a few specific use cases that will be explored in greater depth in Chapter 5.

Top-level (or global) variables and facts also have some inheritence like behaviors, but because there is only one possible scope in which global variables can be defined, unqualified global variables are somewhat less problematic than general case variable inheritance.

Variable Qualifiecation

Local variables should be unqualified, global variables should be fully qualified, and fully qualified out-of-scope variable references should be avoided if possible.

Fully qualifying variable names accomplishes two goals:

1. Clarifies your intent
2. Disambiguates local and global variables

When a variable is fully qualified, it becomes clear that your module is attempting to consume a top level variable, and eliminates the possibility that you simply forgot to define that variable or are attempting to inherit a variable from a higher scope. This disambiguation is important when you revisit your code in the future, either to extend the code or debug a problem.

Many validation tools also assume that unqualified variables are local, and will throw a warning if the variable is not defined in scope. With *puppet-lint*, this behavior can be disabled, however I recommend against doing so as it's a useful way to catch subtle bugs.

While you can fully qualify references to local variables, using unqualified names makes it clear at a glance that the variable is local and has been defined in the current scope. This hint again is used by the validators.

Finally, I strongly recommend against creating inter-class variable references using fully qualified variable names. Such references are a useful stop-gap measure when upgrading code that relies on variable inheritance, however it usually violates the principle of separation of concerns. A major issue with inter-class variable references is that there's no way to tell from the referenced class that such a reference exists. As a result, it's very easy to break the reference when re-factoring code. Instead, consider parameterizing your classes, using class parameters to pass data from one module to another. A side benefit of this approach is that it tends to reduce and eliminate circular dependencies.

Example 4-20. Passing variables using fully qualified variables

```
class parent {
  $foo = 'alpha'
}

class parent::child {
  $foo = $::parent::foo
  alert($foo) #Prints 'alpha'
}
```

Example 4-21. Passing variables using class parameters

```
class parent {
  $foo = 'alpha'

  class { 'child':
    foo => $foo,
  }
}

class parent::child (
  $foo,
) {
  alert($foo) # Prints 'alpha'
}
```

Example Example 4-21 is somewhat more verbose than Example 4-20, but is designed in such a way that variable handling is strictly enforced and clearly documented in code. This increase in verbosity improves the quality of our code.

Of course, separation of concerns would also dictate that a class should only declare its own subclasses; declaration of unrelated classes should be handled at a higher layer of abstraction; our domain logic [5].

Example 4-22. Passing variables using class parameters

```
class profiles::variable_scope {
  $foo = 'alpha'
```

5. see [Link to Come] for more information.

```
  class { 'first':
    foo => $foo,
  }

  class { 'unrelated':
    foo => $foo,
  }
}

class first (
  $foo,
) {
  alert($foo) # Prints 'alpha'
}

class unrelated (
  $foo,
) {
  alert($foo) # Prints 'alpha'
}
```

I generally recommend referencing out of scope variables. Instead, pass variables into classes using class parameters. This may seem like an arbitrary guideline at first, but it helps to control a flow of data through your code and avoids surprises when re-factoring code. Fully qualified variables make it simple to pass data laterally, where use of parameters for passing values tends to enforce a top-down approach to data handling.

There are a few exceptions to this guideline of course; the *params.pp pattern* discussed in "params.pp pattern" on page 96 is the primary example.

Trusted variables

Puppet 3.4 introduces the `$trusted[]` data hash and Puppet 3.5 introduces the `$facts[]` data hash. These hashes are disabled by default as of Puppet 3.7.5, but can be enabled with the setting `trusted_node_data = true` in your `puppet.conf` file.

Because these variables are not enabled by default, you should not rely on their presence when developing code for public release via GitHub or the Puppet Forge. While you can test for the presence of these variables, the cleanest approach for the time being is to use the old method of referencing facts as top-scope variables. In security sensitive contexts, you may wish to rely on the `$trusted[]` hash regardless.

For modules intended for internal use, I would strongly advise using trusted data hashes. Doing so helps disambiguate your code, and can help avoid some nasty attacks against exported resources.

Remember that facts can be arbitrarily defined by the client. Data in the `trusted[]` hash is guaranteed to be declared or validated by your Puppet master.

If you are using global variables for anything security sensitive, I strongly advise ensuring that they are declared unconditionally in your site.pp file or ENC, in order to avoid the risk of abuse via client-side facts.

Order of authority for Global variables

Global variables may be defined from a number of different sources, including your ENC, your site-wide manifests, by the Puppet interpreter, by the Puppet Master, and via facts supplied by the client.

In some cases, the variables defined by different sources are reserved; for example interpreter defined variables such as `$name` `$title` and `$module_name`. In other cases, more authoritative data sources will override less authoritative data sources.

For example:

- Global variables from site.pp override ENC supplied global parameters.
- ENC supplied variables override facts
- Facts have the lowest preference

Understanding this can be important when writing security sensitive code, and can be leveraged to enable or disable user-overrides based on conditions such as the tier of a host.

Strict variables

Puppet 3.5 introduces the *strict_variables* setting in `puppet.conf`. With strict_variables enabled, referencing an undefined variable will throw an error. Enabling this setting can help you catch all kinds of bugs and typos.

Example 4-23. Strict variables example

```
$field_color = 'green'

notify { "The field is ${feild_color}"}
```

Example 4-23 would compile and produce an incorrect result without `strict_variables` and throw an easily identified error with `strict_variables` enabled.

This setting may cause problems with publically available Forge modules and legacy code. It is however a hugely beneficial setting. I strongly recommend enabling it if possible.

If you happen to encounter a module that fails with strict_variables enabled, I recommend submitting a fix to the module author. I strongly encourage you to test your modules with this setting enabled.

Other Variable Use Cases

Variables are tremendously powerful for cleaning up your code. Variables can be used to give meaning to complex data and can tremendously simplify complex quoting and escaping.

A non-puppet example: the purpose of the following regular expression might not be immediately clear:

Example 4-24. Regex without a name

```
grep -o '`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`' logfile
```

The purpose of this regular expression is immediately appearent by assigning it to a variable before using it:

Example 4-25. Regex with a variable assignment

```
IPV4_ADDR_REGEX='\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b'

grep -o $IPV4_ADDR_REGEX logfile
```

Variables are invaluable when you have complex quoting requirements, as is often the case when working with directory paths on Windows hosts.

Consider the following example:

Example 4-26. OS specific resources

```
$sevenzip    = '"C:\Program Files\7-Zip\7z.exe"'
$archive     = '"C:\temp\archive.zip'
$output_path = '"C:\temp\deploy"'

exec { 'extract_archive.zip':
  command => "${sevenzip} e ${archive} -o${output_path} -y"
  creates => $output_path,
}
```

This example is dramatically simplified by declaring our paths as single quotes strings and interpolating everything together in a double quoted string. Single quotes strings allow us to embed quotes in our paths to better handle spaces, and avoids the need to double escape our directory delimiting backslashes.

Function Calls

Puppet's rich assortment of built-in and stdlib function calls are one of the more frequently overlooked features of Puppet. I strongly recommend reading through both the list of built-in functions and the list of `puppetlabs/stdlib` function calls once or twice to familiarize yourself with what's already available.

Very often there is a function call that can help you debug or solve a problem you're currently experiencing while developing a module.

stdlib compatibility

Be aware that puppetlabs/stdlib has 4 major releases, not all of which are compatible with older releases of Puppet. You may also find situations where the `puppet module` tool will not install modules due to version dependency conflicts between other modules and stdlib.

Be aware that many useful functions available in modern versions of stdlib may not be available in the version of stdlib installed at your site.

Know your minimum version compatibility with stdlib

When writing a module that depends on stdlib, it's good to take a moment to determine the actual minimum version of stdlib your module depends on. If all of the features you require are available in stdlib 3.x, declaring that as a minimum version in your module metadata will make life much easier for folks wishing to use your module on older existing sites. This is especially true for PE users, which often have a slightly outdated release of stdlib bundled with PE.

Functions for logging and Debugging

`alert()` `crit()` `debug()` `emerg()` `err()` `info()` `warning()` are useful for adding debugging information to your code. I strongly recommend these function calls over the `notify` resource type, as they do not insert extra resources into the catalog and do not report as a change to the client state.

Notify resources should be used in cases where you do wish to see a change reported to the puppet console.

If in doubt, use a function call rather than a notify resource.

`fail()` is also incredibly useful in this regard. Fail allows you to use conditional logic to test for unexpected conditions, and terminate catalog compilation if such a condition is met. It is commonly used by module authors to kill a puppet run when modules are applied to an unsupported platform.

`fail()` is also valuable when decommissioning obsolete code. Inserting a `fail()` statement can make it clear that a module or code path has been intentionally removed, and allows you to supply a message explaining why the code is gone, and what the user should do to rectify the problem.

String manipulation functions

The list of string manipulation functions in puppetlabs/stdlib is fairly extensive. I strongly recommend reading the documentation for stdlib for an overview of all the available functions.

A few specific useful function calls:

`split()` converts a string into an array, splitting at a specific character. This is very useful for handling delimited inputs, or for for performing directory and filename manipulation. `join()` from puppetlabs/stdlib can be used to convert an array back into a string.

`strip()` `lstrip()` `rstrip()` are useful for trimming whitespace from input. With Puppet, most input is machine generated, and striping the data is rarely necessary, except perhaps to remove problematic newline characters.

`downcase()` accepts a string as input and returns it converted to lowercase. `upcase()` and `swapcase()` are also available.

Path Manipulation

`dirname()` and `basename()` can be used to parse the filename or directory name out of a fully qualified file path. This is often very useful for file manipulation.

You can also use `split()` `join()` and array indexing to perform more complex path manipulation if necessary. This approach can be useful when working with older releases of puppetlabs/stdlib which do not provide the `dirname()` function call.

Input validation functions

puppetlabs/stdlib provides a very rich assortment of input validation function calls. In many cases, the calls simply accept an input parameter and produce a useful failure message if the input fails validation. In other cases, function calls can be used with conditional logic to declare the `fail()` function.

Before we continue, a best practices recommendation regarding input validation: Unlike your typical web application, you probably don't need to validate absolutely every user input from Puppet, especially when those inputs originate from the Puppetmaster itself. In many cases, data supplied by the client is returned directly to the client, and is unlikely to have tangible impact on any other node in your infrastructure. The security risk in such cases is low; unless you are using dangerously insecure permissions on your client infrastructure, only already privileged users should have the necessary permissions to modify Puppet or its environment.

If you are operating in a high security environment and cannot trust the authors of your code [6], I strongly recommend using a peer review / maintainer process to audit code and data before it is deployed to your puppet-masters rather than relying on input validation for security [7]

Do however consider validating inputs that might be executed locally to the Puppet-master via function calls, or might be exported to other nodes in the infrastructure.

Input validation can create problems when when it rejects otherwise valid inputs. Use validation to improve user experience; to produce more useful information than the user would otherwise have by allowing malformed data to create failures elsewhere.

We will discuss input validation in more depth in Chapter 5.

Automatic Validation Functions

These function calls accept an input and automatically cause the Puppet run to fail with a useful error if the input does not match the function calls validation critera. For example:

Example 4-27. Input validation using function calls and conditional logic

```
validate_absolute_path('/tmp/test.txt')
```

`validate_absolute_path()` is useful for ensuring that that a supplied path is valid and fully qualified.

`validate_re()` validate input against a regular expression. This can be used to validate any string within reason.

`validate_slength()` Validate that a string or elements of an array are not longer than a specified character count.

`validate_array()`, `validate_bool()`, `validate_hash()`, `validate_string()` validate that the argument passed is of the correct data type.

`validate_augeas()` Validate the content passed to this function call using an augeas lens, and optionally ensure that certain nodes do not exist in the supplied configuration.

`validate_cmd()` Validate input string using an external command. Be cautious, as this validation function could potentially be used to execute arbitrary code on your Puppetmaster.

`assert_type` introduced in Puppet 4 provides a more generalized way of validating an objects type. Although not required, it allows you to define a block to be called if the

6. this is common in multi-tenant enterprise environments

7. see [Link to Come] for a deeper dive into release management.

object is of the wrong type. I strongly advise against abusing this function for non-input validation related uses.

Other Useful Functions for Validation

These function calls will return true or false depending on the supplied data. Using conditional logic, it's fairly simple to fail catalog compilation if one of these tests returns a negative value.

For example:

Example 4-28. Input validation using function calls and conditional logic

```
unless is_domain_name($domain) {
  fail("'${domain}' does not appear to be a valid domain name")
}
```

`is_domain_name()` `is_ip_address()` `is_mac_address()` validate that the input is a syntactically valid domain name, IP address, or mac address. These function calls do not check to see if the input is actually reachable on the wire.

`is_numeric()` `is_float()` `is_integer()` Returns true if the supplied value is a number, is a floating point number, or is an integer respectively.

`has_interface_with()` `has_ip_address()` `has_ip_network()` Uses the *interfaces* facts to validate that the supplied IP address, network name, or interface is present on the node. Because these tests rely on untrusted client supplied facts, it is possible to spoof them.

`grep()` `has_key()` `member()` Test if the supplied value is part of a data structure. These tests are very useful for ensuring that a supplied value is a member of an explicit list of values.

Example 4-29. Input validation using the `member()` function

```
$ensure_values = ['present','absent']

unless member($ensure_values, $ensure) {
  fail('$ensure must be 'present' or 'absent')
}
```

Catalog tests

`defined()` `defined_with_params()` `ensure_packages()` `ensure_resource()`

These function calls either return a boolean value if a resource exists in the catalog, or they check if a resource exists, and define it if it does not.

At first glance, these functions appear to be a useful way to solve duplicate resource declaration issues.

Example 4-30. Using defined() to see if Java is in the catalog

```
unless defined(Package['openjdk']) {
  package { 'openjdk':
    ensure => 'installed',
  }
}
```

Unfortunately, the above code won't work, as it's parse order dependent. If OpenJDK is added to the catalog after this block is evaluated, a resource conflict will still be created.

Example 4-31. Example of parse order dependent defined() behavior

```
alert (defined(package['openjdk'])) # returns false

Package { 'opendjk':
  ensure => 'installed',
}

alert (defined(package['openjdk'])) # returns true
```

The obvious answer would be to wrap every potentially conflicting resource in a defined() test, or to declare every resource using the `ensure_resource()` function calls. Doing so creates other potential issues:

Example 4-32. conflicting ensure_resource() declarations

```
ensure_resource('package', 'jre', {'ensure' => '1.7.0'})
ensure_resource('package', 'jre', {'ensure' => '1.8.0'})
```

Example 4-32 raises a duplicate resource declaration error. The same code written using the `defined()` function call would result in non-deterministic and potentially non-idempotent Puppet runs. `defined_with_params()` works around the problem, but creates other issues.

Managing conflicts using `defined()` or `ensure_resource()` also subtly violates the principle of separation of concerns; it requires that all modules use the same approach to handling package version tests. To update the test or resource declaration in one module, you would have to update all potentially conflicting modules.

These function calls do have two valid use cases. You can use these tests to ensure that a defined type only declares a resource one time, even when invoked more than once. You can also use these tests when one or more subclasses of the same module could potentially declare the same resource.

Be aware that these function calls are functionally similar to using the `include()` suite of function calls, and they are functionally similar to virtual resources. You may find designing around one of those approaches to be more robust.

Iteration

Iteration is the act of applying the same process to multiple elements in a set. Puppet 3 has limited support for iteration, generally using a functional approach to iteration. Puppet 4 introduces new parameterized block iterators that provide a great deal more flexibility, and are often more readable than previous solutions.

Iteration with Puppet 3

Puppet 3 provides two primary approaches to iteration:

1. Array style resource declaration
2. The create_resources() function call

Either approach may be combined with defined types to iterate over a set of resources. `create_resources()` works by accepting a resource type as an argument, and a hash where the resource name is the key, and the values are a nested hash containing parameter name / parameter value pairs. Because of the way `create_resources()` is implemented, supplied data must take a specific structure; `create_resources()` is incapable of iterating over a flat hash because it has no way of determining which property the value should be applied to. It must be provided a nested hash of parameter names and values.

Example 4-33. Iteration using create_resources()

```
$services = {
  'puppet'       => { 'ensure' => 'running', },
  'puppetserver' => { 'ensure' => 'running', },
  'httpd'        => { 'ensure' => 'stopped', },
}

create_resources('service', $services)
```

In cases where you wish to declare a bunch of resources and are able to simply pass a parameter hash to each resource, this approach can be simpler than using a Future Parser `each` loop. The only disadvantage to this approach is that `create_resources()` isn't always intuitive.

However, you may run into situations where you don't simply wish to turn a data structure into resources. Example 4-34 is a fairly arbitrary example; in this case, we want to automatically determine the enabled state of a resource based on the ensure parameter.

Example 4-34. Iteration using create_resources()

```
$services = {
  'puppet'       => { 'ensure' => 'running', },
  'puppetserver' => { 'ensure' => 'running', },
  'httpd'        => { 'ensure' => 'stopped', },
```

```
}

create_resources('ensure_service', $services)

define ensure_service (
  $ensure => undef,
) {
  $enabled = $ensure ? {
        'running' => true,
        default   => false,
  }

  service { $title:
    ensure  => $ensure,
    enabled => $enabled,
  }
}
```

In this case, the only solution to our problem is to combine a nested data structure, with the `create_resources()` function call, and a defined type. The defined type should be declared in it's own puppet manifest so that it can be handled by the autoloader. The result is fairly unwieldy and difficult to debug.

Iteration and KISS

Example 4-33 creates a huge amount of complexity in order to automagically determine the correct value for the enabled parameter. I would recommend against using this example in practice; it would be much simpler to use the example in Example 4-33. The complexity of trying to automatically set the enabled parameter simply isn't justified in this case. You will however see this pattern in other situations, for example if you have an account defined type to handle user, group, and home directory management as a single resource.

Iteration with Puppet 4 and the Future Parser

Puppet 4 introduces a number of block iterators. Most, such as `filter`, `map`, `reduce`, `scanf`, and `slice` are most useful for data transformation and iteration within EPP templates.

The `each` block provides a useful way to iterate over data sets, and provides a level of flexibility not previously available to Puppet. For example, each may be used to transform a flat hash into a list of resources.

Example 4-35. Iterating over a flat hash using each

```
$services = {
  'puppet'       => 'running',
  'puppetserver' => 'running',
```

```
  'httpd'        => 'stopped',
}

$services.each |$service, $ensure| {
  $enabled => $ensure ? {
        'running' => true,
        default   => false,
  }

  service { $service:
    ensure  => $ensure,
    enabled => $enabled,
  }
}
```

Compare Example 4-35 to Example 4-33. In this case, the `each` block is cleaner, and much more flexible than the `create_resources()` equivalent.

Be aware that there are many situations where you should still use a defined type and the `create_resources()` iterator even with Puppet 4 or the future parser. The following example code makes good use of both features; an `account` defined type (not defined here) abstracts away the logic for defining users, groups, and home directories, while `create_resources()` handles the logic of turning a hash into resource definitions. The equivalent code using an `each` iterator would be much more complex, and would violate several coding principles, including separation of concerns [8].

Example 4-36. Iterating over a flat hash using each

```
$accounts = {
  'alice  => { 'uid' => 101, },
  'bob'   => { 'uid' => 102, }
  'carol' => {
    uid    => 103,
    groups => 'admin',
  },
}

create_resources('account', $accounts)
```

The other block iterators can prove to be incredibly useful in certain situations. For example, we can use `filter` to test if our host has an interface belonging to a list.

Example 4-37. Testing for membership using filter

```
$cluster = [ '192.168.1.1', '192.168.1.5', '192.168.1.10']

$membership = $cluster.filter |$ip| { has_ip_address($ip) }
```

8. The class responsible for defining accounts is domain logic, and should not be concerned with what is actually involved with defining an individual user.

```
if size($membership) > 0 {
  alert('I am a member of the cluster')
}
```

Example 4-37 only works with Puppet 4. This code example fails on Puppet 3 with the future parser.

Each blocks are very sensitive to the use of boolean function calls, such as `has_ip_address()` in Example 4-37. While Puppet typically allows the omission of round brackets on function calls, the syntax `has_ip_address $ip` would throw an error in this example, where `has_ip_adress($ip)` is accepted.

Generating lists

There are a number of cases where you may wish to dynamically generate a list of items to iterate over. This can happen for benchmarking purposes, and is often required when creating cron jobs.

`range()` A useful function that accepts a beginning point, end point, and iterator and returns an array of elements based on that criteria. While the documentation explains that it can iterate through names, I find it's much more reliable to generate an array of integers, and use `prefix()` and `suffix()` to create names.

`prefix()` `suffix()` are useful functions with generate. They accept an array and a prefix or suffix respectively, and return an array with the prefix or suffix applied to each element.

Example 4-38. Iterating over a flat hash using each

```
$range             = range('0', '3')
$range_with_prefix = prefix($range, '/tmp/test')
$files             = suffix($range_with_prefix, '.txt')

# $files == [ '/tmp/test0.txt', '/tmp/test1.txt', '/tmp/test02.txt' ]

$defaults = {
  ensure  => 'file',
  content => 'example',
}

create_resources('file', $files, $defaults)
```

range is useful for specifying intervals, such as with cron.

Example 4-39. Iterating over a flat hash using each

```
$command  = '/usr/bin/git --git-dir=/var/www/site/ pull'
$minutes  = range(fqdn_rand(14), 59, 15)
```

```
cron { 'pull_website':
  command => $command,
  minute  => $minutes,
}
```

In this example, we use `fqdn_rand()` to specify a pseudo-random starting minute between 0 and 14, and range to create an array containing 15 minute increments of that value. This approach is very useful if we have hundreds of web-servers that might otherwise overwhelm our infrastructure if they all synchronized at the same minute [9]. This approach is somewhat more flexible than using cron step values, though using a `fqdn_rand()` with a cron range and a step value might produce comparable results in this case.

Data Transformation

Data Transformation is the act of manipulating data in one or more ways. We've already seen a few examples of data transformation, such as Example 4-38 where we converted an array of integers into a list of filenames.

Puppet does not transform data in place; instead data may be transferred when passing it between classes, or transformed when defining a new variable.

Best practices with Puppet is to avoid transforming data unless necessary. When data must be transformed, I recommend doing so in the same class that initially defines the data, or within the init manifest of the module that consumes the data.

Take the following example:

Example 4-40. Some YAML data

```
files:
  - foo.txt
  - bar.txt
  - baz.txt
directory: /tmp
```

Example 4-41. Some YAML data

```
class roles::myfiles {
  $files     = hiera('files')
  $directory = hiera('directory')

  class { 'profiles::myfiles':
    files     => $files,
    directory => $directory,
  }
```

9. often referred to as a *thundering heard*

```
}

class profiles::myfiles (
  $files,
  $directory,
) {

  $apnfiles = prefix($files, $directory)

  class { 'myfiles':
    files => $apnfiles,
  }
}

class myfiles (
  $files,
) {
  file { $files:
    ensure => 'present',
  }
}
```

This somewhat confusing chunk of code defines data using Hiera, passes it between 3 classes, and transforms the data as it's passed through `profiles::myfiles`. Imaigne for a moment that the `file` resource contained in the class `myfiles` was producing an unusual result. Our first reaction would be to look at the class `myfiles`, and then perhaps to look at the data in Hiera that feeds `myfiles`. Unfortunately, the developer would not be seeing the full picture, because the data is transformed in `profiles::myfiles`. In this case, debugging will go slowly because we have to follow the path the data transverses our code-base to identify the step where the transformation takes place.

This example could be improved significantly using one of several approaches:

1. Store the data in its final form.
2. Transform the data in Hiera using interpolation tokens.
3. Transform the data in the class `myfiles`

In this case, the correct solution would be to store the absolute pathname in Hiera. When that is not feasible, the next best solution is usually to pass the data verbatim to the module, transforming the data in the same class that applies it.

While this approach may violate the DRY principle, it's much simpler to debug.

There are a number of cases where data must be transformed inside the module. One such example are cases where the default of one parameter should be based on the value supplied to another parameter. We'll see examples of this in Chapter 5.

Whenever possible, it's best to avoid passing data needlessly through multiple classes. The Hiera data terminus can be very helpful this way. See [Link to Come] for an in-depth discussion of the pros and cons of various approaches relating to Hiera.

Templates

Templating is a traditional way of separating *presentation logic* from *business logic*. In the case of Puppet, templateing allows us to easily interpolate data from Puppet into our configuration files, while maintaining a layer of separation between the source configuration files and the logic used to populate those files.

ERB Templates

The ERB templateing language is built into Ruby, and has been supported since the very early days of Puppet. ERB allows ruby code to be embedded into configuration files, and allows the use of Ruby iterators and function calls within those templates.

There are a few best practices relating to the use of ERB template:

Never source a template from another module. This violates the separation of concerns, and often results in problems down the line when someone changes the template in the other module without realizing your module will be affected. If you need to pass templates between modules, render the template in the source module, pass the template in its rendered form as parameter data.

Try to avoid referencing out of scope variables within your template, even when those variables are declared in other classes of the same module. Absolutely avoid referencing variables from other modules. Instead, copy out of scope variables into the local scope, and reference the local scope variables. The reason for this recommendation is that it's often hard to tell what variables are being used by a template, especially when reviewing Puppet code. Referencing only local variables guarantee that any variable used by your template is defined in the same class that declares the template. This also helps avoid subtle bugs created when out of scope variables change and are treated as `nil` within the template rather than throwing an exception.

Example 4-42. Referencing an out of scope variable using ERB

```
class example::alpha {
  inline_template('<%= scope["::example::beta::foo"] %>')
}
```

Variables local to the template function call are available as Ruby class variables within the template. Out of scope variables must be referenced using the `scope.lookupvar()` function call, or the `scope[]` array (recommended for Puppet 3+.)

Besides creating maintenance problems, out of scope variables lookup methods have inconsistent behaviors when tested as boolean value, evaluating to true in unexpected situations.

EPP Templates

Embedded Puppet Programming Language (EPP) templates were introduced with the future parser, and are available to all Puppet 4 users.

EPP is stylistically similar to ERB templateing, however it uses Puppet iterators and interpolation tokens, rather than native Ruby code.

The huge advantage of using EPP templates over ERB templates is that EPP templates permit you to parameterize your template and to pass an explicit list of input values to the template using the EPP function call. This is a **huge** maintainability win.

Example 4-43. Declaring an EPP template

```
$epp_args {
  'arg_a' => 'Value 1',
  'arg_b' => 'Value 2',
}

$example_content = epp('example_module/template.epp', $epp_args),

file { '/tmp/example.txt':
  ensure  => 'file',
  content => $example_content,
}
```

Example 4-44. EPP template with input parameters

```
<%-| $arg_a, $arg_b |-%>

arg_a is <%= $arg_a %>
arg_b is <%= $arg_b %>
```

Best practices is to always explicitly pass variables into your EPP templates using this syntax.

I strongly recommend against declaring EPP input parameters inside a resource declaration; instead either assign the output of the epp template to a variable and pass that to the resource, or assign the variables to a hash and pass that to the EPP statement (both approaches are demonstrated in Example 4-43)

The general recommendations for ERB templates also apply; do not reference out of scope variables, and never render templates stored outside the scope of your module.

EPP vs. ERB

The ability to explicitly pass input parameters to EPP templates is a significant advantage over the ERB templateing engine. If you do not need to support Puppet 3 clients, you should seriously consider adopting EPP templates for this reason alone.

Otherwise, the choice between the two comes down to whether you need to support older agents, and personal preference.

ERB is a very common templateing language, and as a result enjoys a lot of tooling that isn't available with EPP templates. This may or may not be an issue for you.

Template abuse

Because templates are simply rendered into strings within Puppet, it's often tempting to use ERB syntax to do things that might be more difficult in the Puppet DSL, such as appending strings or variables.

Try to avoid this temptation; it's often a sign that you need to re-think your data inputs, or that your code can be re-factored to be more declarative.

`inline_template()` is especially prone to such abuse.

The puppet::file defined type

Here's an implementation of the `puppet::file` defined type mentioned in "Don't Repeat Yourself (the DRY principle)" on page 52. It is placed here to demonstrate how splitting your logic between manifests can create readability problems.

Example 4-45. An implementation of the puppet::file defined type

```
define puppet::file {
  $basename = basename($title)

  file { $title:
    ensure => 'file',
    mode   => '0440',
    owner  => 'root',
    group  => 'puppet',
    source => 'puppet://modules/puppet/${basename}',
  }
}
```

Although this code is very DRY, having to flip back and forth between files can be painful when debugging code. This simple example only involves two files; the `init.pp` manifest and the `file.pp` manifest containing this defined type. In more complex cases it's easy to end up with a problem space spanning many more files than this. Imagine if you were debugging a problem involving your profiles, hieradata, your site.pp file, your ENC, 2

custom functions, 3 manifests, and a few DRY defined types such as this one thrown in the mix. Would saving a few lines of code be justified in such an example?

Other Language Features

- Resource relationships, exported resources, meta-parameters, and virtual resources will be discussed in Chapter 6.
- Classes and defined types will be discussed in Chapter 5
- Node statements will be explorded in (to come).
- We will look at run-stages in [Link to Come].

Summary

This chapter discussed a number of coding practices, and provided an overview of useful function calls. Applying clean coding practices as discussed in this chapter will help make your code easier to understand, and thus easier to maintain. In many cases, simple and clean code will often result in fewer defects as well.

Recommendations from this chapter:

- Apply common development principles to improve the quality of your code
- Reduce the amount of code in your Puppet manifests
- Separate your code from your resource declarations
- Use clearly named variables to clarify the purpose of your code.
- Separate your code into modules for re-usability
- Avoid creating additional scope needlessly
- Use Puppet's built-in and stdlib function calls to enhance your code
- Be extremely careful with scope when using templates

CHAPTER 5
Puppet Module Design

In this chapter, we discuss practices related to designing Puppet Modules.

Puppet modules are self-contained bundles of code and data. Being self-contained allows modules to be portable; the module is placed into the module path; it's code is auto-loaded when requested, and the data contained within the module may be referenced by the module name rather than its location on the host machine.

Because modules must be auto-loaded, they have a fairly strict structure. However, modules are very flexible in terms of their contents; a module may extend puppet with new facts, functions, types and providers. It may also include native Puppet DSL code, files relating to the module, meta-data and documentation, or data completely unrelated to Puppet.

For the purposes of this chapter, we will focus on Modules containing native Puppet DSL code in the form of Puppet manifests, templates, files, meta-data, and tests. We will discuss modules containing native code in [Link to Come] and the creation of modules for roles and profiles in [Link to Come]. Distribution and deployment of modules is discussed in [Link to Come].

Good module design relates very strongly to good code design. We touched on a number of coding principles and practices in Chapter 4, including *separation of concerns*, the *single responsibility principle, KISS,* and *interface driven design.* We apply those principles in this chapter.

I also recommend reading [Link to Come] for information on how to build a development environment in which to author Puppet modules. Use of available tools and a good development environment tends to promote good coding practices, will help identify and eliminate code defects, and will improve the overall quality of your modules.

Design modules for public consumption

All modules (other than your roles and profiles) should be designed as if they are going to be released to the public.

This is not to say that you should release all of your modules, or that your modules are generally available. It means that the design patterns for the creation good public modules are the design patterns for creating good modules in general.

Your modules should seek to be well defined, well documented, well tested. They should minimize dependencies, and contain the application specific data needed for generic cases. When building a module, ask "is this data or dependency specific to my site?" If the answer to this question is "yes" you may be dealing with domain logic or site specific data that should be shifted to Hiera or your profiles.

Modules absolutely do need to meet the requirements of your site. The design considerations that went into creating the *puppetlabs/apache* module creates a lot of complexity. The complexity means that *puppetlabs/apache* is very heavy, and isn't suitable for bulk hosting.

Designing modules for portability ultimately makes them simpler to support and extend. It helps to eliminate technical debt. The design patterns for creating public modules encourage re-use. Re-use means that you won't have to re-write the module from scratch every time your requirements or environment change.

Design every module for public consumption, even if it will never be released to the public.

Using public modules

Before you begin the process of developing your own modules, it's a very good idea to check what's already available on the Puppet Forge module repository. In many cases, someone may already have developed a module suitable for your site.

Modules from the forge have a number of benefits:

- You don't have to spend time developing the module
- Public modules tend to be maintained
- Public modules tend to see community reporting of bugs
- Public modules often include good documentation and test-cases

Picking good modules

To paraphrase Sturgeon's law: 90% of everything is crud.

There are a lot of great modules on the forge, but there are many-many poor modules as well. The forge is not a curated list; anyone can write and publish a module, regardless of quality.

There are thousands of modules available on the forge, and for the most part Puppet Labs does a good job of highlighting the best modules. Download statistics, ratings, platform support, test results, and module documentation is published. Puppet Labs provides first party support for their own modules, and highlights approved modules which have undergone a review process.

Module checklist

When considering a module from the forge, it's a very good idea to review the module. There are a few key points to check:

- Is the module well documented?
- Does the module include a clean interface?
- Does the module include spec tests, and do those tests pass in your environment?
- Does the module follow the Puppet Labs style guide?
- Does the module conform to module development best practices?
- Does the module pull in a lot of dependencies? Are those dependencies warranted?
- Does the module license meet your requirements?
- Is the module hosted publicly? On GitHub perhaps?
- Is the author responsive to pull requests and issues?
- Is the module actively maintained?

Module Applicability

Often otherwise good modules simply won't be suitable for your site, or your specific needs. It's important to carefully consider your requirements when selecting a module.

Platform support

Does the module support the platforms you have deployed?

If you are a CentOS only site, there's a high probability any published module will work on your site. However, if you must also support Solaris or Windows nodes, you choices may be somewhat more limited. While there are great multi-platform modules available, many published modules only support Debian and Enterprise Linux platforms. If the module doesn't support your platform, consider how difficult it would be to extend the module. Perhaps you can fork the module or submit a pull request to the original author?

Scaling

Will the module scale?

In some cases, the actual implementation of a module can limit the scalability of that module. The `puppetlabs/apache` module is a prime example; it's hugely powerful, but the extensive internal use of concat resources to build configuration files means that it's not suitable for large scale hosting. In those cases, you'll need to use another module, or write your own vhost template.

Modules that rely on exported resources are another common example. Many Nagios modules simply won't scale to support huge sites.

Features

Does the module support the application features you require?

Most modules only manage the more common configuration features of an application or service. While some of the more popular modules, such as MySQL and apache can be incredibly comprehensive, you may find some limitations in less popular modules.

If you find a module that meets most of your requirements, consider how difficult it would be to extend the module, and consider contributing your extensions to the up-stream author.

Embracing and extending modules

If you find a module that meets most but not all of your requirements, remember that you can always fork and extend the module. R10k is very helpful in these cases; With R10k you can simply fork the upstream module and maintain it as an independent repository, maintaining your own features as a branch to the original module. From that point, it's fairly straight-forward to re-base and integrate your changes with updates provided by the original author.

Of course, it's best to contribute up-stream when possible. Code accepted into the mainline branch is code that can be maintained by the original author, and general public. Merged code is a much lower risk of breaking in future releases.

See [Link to Come] for more information regarding R10k and module management.

Contributing Modules

If you've written a new and interesting module, or if you've improved significantly upon modules already available on the forge, consider publishing your module. Assuming you've created documentation and meta-data, the `puppet module` utility can easily package a module for distribution on the puppet forge.

Planning and scoping your module

Before you begin writing your module, it's important to first determine your module's scope. For many of us, our first instinct is to write a jumbo module that installs and manages all of its dependencies. Unfortunately, this approach tends to create problems down the line; such modules are often inflexible and become difficult to maintain. They can create compatibility problems when they manage a dependency that's needed by another module.

Design modules using "Seperation of Concerns" on page 49 and "The Single Responsibility Principle" on page 48 as a guideline. As a rule of thumb, if a resource in your module might be declared in any other module it should probably be it's own module.

Dependencies outside the scope of your module should be externalized into their own modules. Domain logic should be handled using roles and profiles [1].

In many cases you can rely on your package manager to handle dependencies.

The Java dependency for tomcat is a classic example of a dependency that should be externalized as domain logic. By separating the management of Java from the management of Tomcat, the process of upgrading Java is simplified, and potential conflicts with other modules also attempting to deploy Java are eliminated.

Even if you never plan on ever distributing the module, it should be designed as if you were. By designing your modules to be portable, they can be adapted to new requirements and new environments with minimal effort.

Basic Module Layout

A puppet module has a fairly standardized structure made up of a number of *optional* components.

- `manifests/` containing Puppet manifests written in the Puppet DSL.
- `templates/` containing ERB and EPP templates.
- `files` containing files made available through the `modules` Puppet file server mountpoint.

1. see [Link to Come]

- `tests/` containing test manifests used for system testing, experimentation, and demonstration
- `spec/` containing rspec unit tests and Beaker acceptance tests
- `docs/` containing additional documentation if available (ignored by Puppet.)
- `lib/` native Ruby extensions for Puppet
- `README.md` containing documentation describing your module and its interfaces.
- `Gemfile` containing gem dependencies for testing and extending Puppet
- `Rakefile` containing rake tasks for validating and testing the module
- `metadata.json` containing meta-data about the module for the Puppet Forge and for the `puppet module` tool.
- `.fixtures.yaml` containing dependencies for `puppet apply`, rspec and Beaker testing.
- Additional meta-data for CI, editing, RVM, GIT, and other tools.

Every one of these components is optional. Although many of these will be automatically generated using the `puppet module` tool, they should be removed if not used in order to simplify the module layout and clarify the design and intent of your module.

For example, if you are writing a module that contains only native Puppet extensions, there's no need to have a `manifests` directory or an `init.pp` file. Likewise, if your module contains no files or templates, you should remove those directories.

Finally, while I strongly encourage you to document your code, it's best to remove the auto-generated `README.markdown` file, as well as any meta-data that you don't intend to use or maintain. In most cases, incorrect data is worse than no data, so this bit of cleanup will help to avoid confusion, and will quickly focus attention on what your module does provide.

manifests/init.pp; the module entry point

`manifests/init.pp` is the Puppet manifest containing the root class of your module. This manifest should be present in most modules written in the Puppet DSL, but may be omitted in special cases, such as modules that don't contain DSL code, or the `roles` and the `profiles` modules (see [Link to Come].)

As the entry point, init.pp is the go to manifest to see what input the module accepts, to understand the basic layout of the module, to perform input validation, and to handle user input and transformation of input data.

As a general rule, I recommend moving resources to subclasses, and performing all subclass inclusion and relationship handling in init. This approach makes the module easier to understand, and centralizes the flow and features of the module.

While the init manifest should be the main entry point into your module, this recommendation does not exclude having defined types or other subclasses bundled with your module that might be declared or referenced from outside your module.

An example init class

Let's take a look at the init class for an Apache module. This particular example is extremely simplified; it was designed as an instructional tool. A real Apache module would likely have many more input parameters.

Example 5-1. An example init class for a simple Apache module

```
class apache ( # ❶
  $ensure       = 'installed',
  $config_file  = $::apache::params::config_file,
  $documentroot = $::apache::params::documentroot,
  $errorlog     = $::apache::params::errorlog,
  $group        = $::apache::params::group,
  $listen       = 80,
  $servername   = $::fqdn,
  $user         = $::apache::params::user,
) inherits ::apache::params { # ❷

  validate_absolute_path($config_file)
  validate_absolute_path($documentroot)
  validate_absolute_path($errorlog)

  # group should match debian's useradd filter # ❸
  validate_re($group, '^[_.A-Za-z0-9][-\@_.A-Za-z0-9]*\$?$')

  unless is_numeric($listen) {
    fail("${module_name}: listen ${listen} must be an integer")
  }

  unless $listen >= 0 and $listen <= 65534 {
    fail("${module_name}: listen ${listen} must be a port between 0 and 65534")
  }

  # package must be one or more printable characters
  validate_re($package, '^[[:print:]]+$')

  # user should match debian's useradd filter
  validate_re($user, '^[_.A-Za-z0-9][-\@_.A-Za-z0-9]*\$?$')

  # service must only contain printable characters, and length > 1
  validate_re($service, '^[[:print:]]+$')
```

```
  # servername must match apache's specifications
  # http://httpd.apache.org/docs/2.2/mod/core.html#servername
  validate_re($servername, '^([a-z]+:\/\/)?[\w\-\.]+(:[\d]+)?$')

  class { '::apache::install': # ❹
    package => $package,
    ensure  => $ensure,
  }

  class { '::apache::config':
    config_file  => $config_file,
    documentroot => $documentroot,
    errorlog     => $errorlog,
    group        => $group,
    listen       => $listen,
    servername   => $servername,
    user         => $user,
  }

  class { '::apache::service':
    service => $service,
  }

  contain apache::install # ❺
  contain apache::config
  contain apache::service

  Class['::apache::install'] -> # ❻
  Class['::apache::config']  ~>
  Class['::apache::service']
}
```

❶ As of Puppet 3.7.5 class declarations cannot use fully qualified class names.

❷ Inheritance is used to enforce ordering between the init class and the params class. This is one of the few remaining use cases for class inheritance, and doesn't actually use any inheritance features.

❸ Some input validation such as validate_absolute_path is self-documenting. This regular expression is not, so it's a good idea to add a comment regarding it's purpose.

❹ I prefer to use resource style class declaration for submodules. Other approaches will be discussed in "Subclasses" on page 99

❺ Containment is critical so that we can create resource relationships between modules. See "Subclass containment" on page 101

❻ Using class relationships is dramatically simpler than maintaining resource relationships. We use chaining arrows since they are easier to read at a glance, and simpler to wrap in conditional logic if necessary.

Although simplified, this example demonstrates a layout usable for much larger and more complex modules.

The class begins by accepting its set if input parameters. Defaults that are universal for all platforms are declared here, and platform specific defaults are pulled from the `::apache::params` sub-class.

Input is validated using simple conditional logic and functions from the puppetlabs/stdlib module. We could also include debugging and logging statements here.

If necessary, any data transformation would happen after input validation. None was needed in this example, but if one parameter default was based on another, it could be handled here.

The validated input is then passed to sub-classes using class parameters, and containment of the sub-classes is declared using the `contain()` function.

Finally, relationships are established between classes.

Parameterizing your module

If your module accepts any kind of input, it should be parameterized [2]. Class parameters are a well defined module interface that permit you to declare the class with data. This data can be passed directly to your module's resources or it can alter the behavior of your module using conditional logic.

Example 5-2. A NTP class with a parameter

```
class ntp (
  $servers = 'pool.ntp.org',
) {
  # Resources go here
}
```

If your module has special case input needs, such as to lookup data using `hiera_hash()` [3], the best approach is to still define an input parameter and to set the default value of that parameter so that the lookup you wish to use is performed automatically if no value is explicitly supplied.

Example 5-3. A NTP class with a parameter

```
class ntp (
  $servers = hiera_array('ntp::servers', 'pool.ntp.org'), # ❶
) {
```

2. This statement seems obvious now, but in the past it was common to use global variables to pass data into modules

3. with Puppet 4 you should use `lookup()` instead.

```
  # Resources go here
}
```

❶ Note that we still supply an optional default value in our Hiera lookup to be used in case Hiera is not available.

Automatic parameter lookups take precedence over parameter defaults. Example 5-3 contains a subtle bug; with parameter lookups enabled, ntp::servers if defined would perform a conventional lookup via the data bindings rather than an array merge lookup. The workaround is to perform this lookup in the profile that declares the class instead.

The approach demonstrated in Example 5-3 has several major advantages over alternatives:

1. It avoids creating an explicit dependency on Hiera
2. It allows you to declare the class with an explicit value
3. It facilitates debugging by embedding the result of the lookup in your catalog.

Parameter defaults

It's a good idea to supply default values for all parameters, even if those defaults aren't necessarily going to be useful in a real world environment. In many cases, a default of `undef` is perfectly valid and simple value.

There are two main reasons for this recommendation:

1. It simplifies experimentation with your module.
2. It avoids creating nasty Hiera dependencies during testing

There are many situations where one may wish to test or experiment with one or many modules. This is common when deciding whether or not a module from the Puppet Forge is suitable for a site.

In these cases, It's ideal to be able to test the module by installing it into a temporary module path, testing the module with the apply command.

```
$ mkdir -p example
$ puppet module install --modulepath=example puppetlabs/ntp
Notice: Preparing to install into /home/vagrant/example ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/vagrant/example
|--- puppetlabs-ntp (v3.3.0)
```

```
   |--- puppetlabs-stdlib (v4.6.0)
$ sudo puppet apply ./example/ntp/tests/init.pp -modulepath='./example' --noop
Notice: Compiled catalog for localhost in environment production in 0.74 seconds
Notice: /Stage[main]/Ntp::Config/File[/etc/ntp.conf]/content: current_value
{md5}7fda24f62b1c7ae951db0f746dc6e0cc, should be
{md5}c9d83653966c1e9b8dfbca77b97ff356 (noop)
Notice: Class[Ntp::Config]: Would have triggered 'refresh' from 1 events
Notice: Class[Ntp::Service]: Would have triggered 'refresh' from 1 events
Notice: /Stage[main]/Ntp::Service/Service[ntp]/ensure: current_value stopped,
should be running (noop)
Notice: Class[Ntp::Service]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 2 events
Notice: Finished catalog run in 0.67 seconds
```

The `puppetlabs/ntp` module includes a test manifest and supplies sane defaults for all of its parameters. There is no need to read the documentation to determine what inputs are mandatory, and there's no trial and error involved in a simple test application of the module. As a result, it's very easy to test, evaluate, and deploy this module.

Another case where sane defaults are useful is during rspec testing. If your module has mandatory parameters and may be invoked via an include() call from another module, you've implicitly created a dependency on Hiera (or another data binding.) This can complicate the setup for your rspec test cases, because your Hiera data will not be available in this context.

====== Parameter default complexity

Puppet allows fairly complex generation of parameter defaults. The default may be a function call, nested function, selector, or a complex data structure.

As a general recommendation, if your default value is more than one line long, it should probably me moved into your `params.pp` subclass, even if is not dynamic.

While you can also embed selectors and other complex logic in your parameter defaults, doing so makes the default somewhat difficult to comprehend. The params.pp pattern is a way to move such complex logic outside the parameter default block without losing the benefits of parameterization. See "params.pp pattern" on page 96 later in this chapter.

====== Parameter default limitations

A major limitation of parameter defaults is that a parameter default cannot contain the value of another parameter. For example, the following code is sadly invalid:

Example 5-4. An example of invalid inter-parameter references

```
class chroot (
  $root_dir = '/chroot',
  $prefix   = "${root_dir}/usr",
  $bindir   = "${prefix}/bin",
) { }
```

The workaround to this problem is to use the `_real` variable pattern. With this pattern, we default variables to `undef`, and declare a new variable depending on the value of the original variable.

Example 5-5. An example of parameter interpolation using pick and real variables

```
class chroot (
  $root_dir = '/chroot',
  $prefix   = undef,
  $bindir   = undef,
) {
  $_root_dir = $root_dir                    ❶
  $_prefix   = pick($prefix, "${root_dir}/usr")
  $_bindir   = pick($bindir, "${prefix}/usr")
}
```

❶ Note that while redefining $root_dir is not necessary, declaring this variable makes your code more consistent, and thus easier to read and maintain. It's much simpler to prefix pathing variables consistently than it is to try to remember which variables have been redefined and which haven't.

We would then use the underscored variable names in place of the original variable names.

The `pick()` function is from puppetlabs/stdlib and is almost ideal for this purpose. It checks to see if the leftmost variable is defined. If so, it returns that value, otherwise it returns the right side variable.

Why is it called the *_real variable* pattern?

This pattern gets its name because it's common to use the word *real* in new variables. E.g. `$prefix` vs `$prefix_real` I personally feel that prefixing the redefined variable with _ is cleaner. You're welcome to use any convention you prefer.

====== Parameter naming conventions

A Good parameter name has a few properties:

- It should be unambiguous
- It should be are self-descriptive
- Its purpose is fairly obvious
- It should be memorable

A good rule of thumb is to name the parameter after whatever will consume it's value. For example, if your parameter will supply a `DocumentRoot` value for an Apache con-

figuration file, the most intuitive name for your parameter will be `documentroot`. If your parameter provides a source for a package resource, the most intuitive name will be `source` or source prefixed with the package name when necessary to disambiguate multiple sources.

For example, If I was writing an apache module, the following parameter names would be fairly obvious:

- `serverroot`
- `documentroot`
- `listen`
- `servername`
- `errorlog`

If I were writing a module for a relocatable service, I might follow the GNU coding standards and use the directory naming conventions:

- `prefix`
- `bindir`
- `sysconfidir`
- `localstatedir`

In all of these cases, we attempt to conform to pre-existing naming conventions, with minimal transformation to meet Puppet's variable naming requirements.

When a parameter is to be passed to a puppet resource, we might simply re-use the resource's parameter name, possibly prepended with the resource name or resource title to remove ambiguity.

- `ensure`
- `source`
- `package_httpd_source`

Input validation

Input validation is the act of testing input supplied to your module to ensure that it conforms to a specification and doesn't contain any nefarious data.

Available input validation functions and techniques are discussed in "Input validation functions" on page 68.

The design of input validation is very important to consider, and is extremely environment specific. It is important to access the goals of validating input, and the risks associated with invalid input.

In many cases, data supplied to your module will come from a trusted source, such as from Hiera data committed into your sites control repository. In these cases, the goal of input validation is not so much about security as it is about about generating useful failure messages when the input is malformed, ands protecting your systems from invalid or dangerous input.

Input validation should be designed to provide useful troubleshooting information, and overly restrictive validation should be carefully avoided. Specifically, when designing input validation, be cautious that your tests don't reject inputs that are otherwise perfectly valid, such as non-fully qualified paths when your application will happily accept them. For example, with apache, unqualified paths are valid in many cases, and become relative to the ServerRoot directive. With Puppet, variable interpolation can be used in pathnames, allowing many paths to be relative to `$confdir` and other base directories.

The most common source of external untrusted data are `facts` generated by a managed node when working in a master/agent environment. In many cases, the facts are simply interpolated into data returned by the client, or used to provide platform specific defaults. However, there is a risk of privilege escalation if your site allows non-privileged users to supply arbitrary facts to privileged Puppet runs.

When using exported resources or other forms of shared data, facts from one node may be used to attack other nodes on the network. In these cases, input should absolutely be validated to help protect nodes from each other. The best way to protect against privilege escalation is to avoid sharing untrusted data between nodes, and to control access to the Puppet agent so that unprivileged users may never modify it's environment, or configuration and so that it is never invoked with greater privileges than the users capable of modifying it's behavior, libraries, configuration, or environment.

params.pp pattern

The params.pp pattern is a design pattern designed to simplify the creation of platform specific parameter defaults by moving default definition into a dedicated manifest. For those familiar with Chef, the *params.pp* pattern is somewhat analogis to cookbook attributes, with a somewhat less flexible interface.

This pattern is still a common practice as of early release of Puppet 4, but is likely to be deprecated in favor of module data sources in the near future.

With the params.pp pattern, default module parameters are defined in a subclass using conditional logic, and then referenced using their full path in the init class of your module. While Hiera makes it very simple to provide platform specific values to your

module, with Puppet 3.x there isn't a simple way for module authors to ship Hiera data inside their module in a portable way. Requiring your users to add specific data to their site-wide Hiera configuration can be difficult given the number of potential hierarchies and hierarchy back-ends, and it would remove many of the benefits of supplying parameter defaults in the first place.

The params.pp pattern violates several best practices. The pattern relies on the otherwise highly discouraged `inherits` feature of Puppet classes in order to enforce ordering between it and your modules init class. It also relies on fully qualified resource references from outside the init class scope, which is otherwise discouraged by this very book. The params.pp pattern is however far preferable to other alternatives such as embedding selectors in your parameter defaults.

Example 5-6. The init class declaration for a module using the params.pp pattern

```
class apache (
  $ensure       = 'installed',
  $config_file  = $::apache::params::config_file,
  $documentroot = $::apache::params::documentroot,
  $errorlog     = $::apache::params::errorlog,
  $group        = $::apache::params::group,
  $listen       = 80,
  $servername   = $::fqdn,
  $user         = $::apache::params::user,
) inherits ::apache::params { # ❶

  # Module contents go here

}
```

❶ Inherits is mandatory. Otherwise, Puppet will throw an error complaining that we've referenced variables from a class that has not been evaluated. This is a case in puppet where ordering does matter. Inherits is a convenient way to ensure the params class is evaluated before the apache class.

Example 5-7. The params class declaration for a module using the params.pp pattern

```
class apache::params {
  case $::osfamily { # ❶
    'RedHat': {
      $config_file  = '/etc/httpd/conf/httpd.conf'
      $errorlog     = '/var/log/httpd/error.log'
      $package      = 'httpd'
      $service      = 'httpd'
      $user         = 'apache'
      $group        = 'apache' #❷
      $documentroot = '/var/www/html'
    }
    'Debian': {
      $config_file  = '/etc/apache2/apache2.conf'
```

```
      $errorlog     = '/var/log/apache2/error.log'
      $package      = 'apache2'
      $service      = 'apache2'
      $user         = 'www-data'
      $group        = 'www-data'
      $documentroot = '/var/www'
    }
    default: {
      fail("${::osfamily} isn't supported by ${module_name}") # ❸
    }
  }
}
```

❶ In this case, we are only concerned with platform specific default values. A single case statement is sufficient to meet our needs. We could have other logic, such as to set defaults based on available memory, processor cores, or storage space.

❷ Repetition of user and group names would seem to violate the DRY principle, but actually doesn't in this case. Despite having the same value, the user name and group name are conceptually distinct, where re-using a key would imply the opposite.

❸ The default case should always be supplied to fail the module on unsupported platforms. Users can easily fork the module an add additional cases if needed. This process is discussed in [Link to Come]

The `params.pp` pattern isn't limited to platform specific defaults; you might have a set of defaults depending on whether or not the host is virtualize, the size and resource availability of the node, etc. etc.

You may also wish to place complex data structures and default values here to better organize your parameters list.

When using the *params.pp* pattern, it's important to keep in mind that the values should be application specific defaults, not site specific defaults. *params.pp* is application specific data. Try to use this pattern to provide values that would be true no matter where the module is deployed, and use Hiera or parameters from your roles and profiles to override these defaults based on your domain logic and site-specific needs.

Module data sources; the alternative to params.pp

Puppet 4 introduces module data sources, allowing the module to define it's own data terminus. This ability is expected to replace the params.pp pattern. I have yet to see a practical implementation of module data sources, but will update this section when one is available.

Module data sources will allow a module author to embed a hierarchy within their module, allowing default parameters to be stored as YAML data. This would solve all

the problems inherent in the params.pp pattern and eliminate one of the few remaining use cases for class inheritance.

Although there are other implementations of data-in-module for Puppet, all of them are to one degree or another a bit of a hack. I generally advise using the params.pp pattern until we have useful examples of module data sources to build from.

Subclasses

Subclasses are additional classes in your module; anything that isn't your module's root class in your modules `init.pp` manifest.

Subclass relationships

Subclasses are a useful way to organize data and establish relationships between resources. If we were to write a `puppetmaster` module for example, we could establish a notify relationship between `puppet.conf`, `auth.conf`, `route.yaml`, `puppetdb.conf` and the puppet services or we could place these resources into seperate subclasses and simply declare relationships between the subclasses.

Building resource relationships this way then becomes a huge maintainability win. If we add a new resource to our module, we simply place it in the appropriate class and Puppet automatically establishes relationships between it and the rest of our resources.

Example 5-8. Resource based relationships

```
file { '/etc/puppet/puppet.conf':
  notify => Service['puppet','puppetserver'],
}

file { '/etc/puppet/auth.conf':
  notify => Service['puppet','puppetserver'],
}

file { '/etc/puppet/puppetdb.conf':
  notify => Service['puppet','puppetserver'],
}

file { '/etc/puppet/routes.yaml':
  notify => Service['puppet','puppetserver'],
}

service { 'puppet':
  ensure => 'running'.
  enable => true,
}

service { 'puppetserver':
  ensure => 'running'.
```

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

Example 5-9. Class based relationships

```
class { '::puppet::config': }
class { '::puppet::service': }

contain ::puppet::config
contain ::puppet::service

Class['::suppet::config'] ~>
Class['::puppet::service']
```

Class based relationships are also a huge win if we wish to make our module uninstallable. Removing an application in most cases requires reversing all of the resource relationships and in some cases removing resources from the catalog completely. Establishing class based relationships is very simple, making it easy to reverse the relationships when necessary.

I strongly recommend using relationships as a guide to how to subclass your modules. If you are finding yourself declaring a lot of relationships between two sets of resources, consider using subclasses and class based relationships instead. It is not uncommon to see modules designed with 3 classes for 3 resources.

A final benefit of the subclass pattern is that it can be used to limit the number of variables available to our templates, better indicate what variables are used where, and normalize our variable names after the variables have been transformed.

Example 5-10. Class based relationships

```
class example (
  $prefix     => '/usr',
  $sysconfdir => undef,
)
  $_sysconfdir = pick($sysconfdir, "${prefix}/etc")

  class { '::example::config':
    sysconfdir => $_sysconfdir,
  }
}
```

In example <modules-subclasses-03>, the class `::example::config` can reference the variable `$sysconfdir `rather than the transformed variable name `$_sysconfdir`.

As a general rule, I recommend sub-classing any module that contains more than one resource, and I recommend against nesting subclasses without an extremely good reason. A two level module design where the init class declares subclasses allows init to provide a complete picture of the module layout and flow, while the subclasses provide

organization to your resources. When your subclasses themselves contain subclasses, you risk violating the single responsibility principle.

Subclass containment

Containment causes relationships with a parent class to be applied to subclasses. Consider the following example:

Example 5-11. Class based relationships

```
class java {
  package { 'openjdk':
    ensure => 'installed',
  }
}

class tomcat {
  notify { 'tomcat': }

  class { '::tomcat::package': } ~>
  class { '::tomcat::service': }
}

class tomcat::package {
  package { 'tomcat'
    ensure => 'installed',
  }
}

class tomcat::service {
  service { 'tomcat':
    ensure => 'running',
  }
}

include java
include tomcat

Class['java'] ->
Class['tomcat']
```

In this example, The class `::java` has a before relationship with the class `::tomcat` and the resource `Notify[tomcat]`. `::tomcat::package` and `::tomcat::service` counter-intuitively have no relationship with their parent class, `::tomcat` and thus no relationship with the `::java` class.

Although the submodules are defined in the class `::tomcat`, the relationships with tomcat apply only to resources declared directly in the `::tomcat` class and not to the resources in any of its subclasses. In the above example, the following relationships exist:

Example 5-12. Class based relationships

```
Class['::java'] ->
Paclage['openjdk'] ->
Class['::tomcat'] ->
Notify['tomcat']

Class['::tomcat::package'] ->
Package['tomcat'] ~>
Class['tomcat::service'] ~>
Service['tomcat']
```

To solve this problem, we need to either anchor or contain the subclasses.

Containment

The contain keyword includes a class and creates a relationship between the included class and its parent class. Here is an example of our previous module using containment.

Example 5-13. Class based relationships

```
class java {
  package { 'openjdk':
    ensure => 'installed',
  }
}

class tomcat {
  notify { 'tomcat': }

  contain ::tomcat::package
  contain ::tomcat::service

  Class['::tomcat::package'] ~>
  Class['::tomcat::service']
}

class tomcat::package {
  package { 'tomcat'
    ensure => 'installed',
  }
}

class tomcat::service {
  service { 'tomcat':
    ensure => 'running',
  }
}

include java
include tomcat
```

```
Class['java'] ->
Class['tomcat']
```

Using contains, the above example now has the following relationships:

Example 5-14. Class based relationships

```
Class['::java'] ->
Paclage['openjdk'] ->
Class['::tomcat'] ->
Class['::tomcat::package'] ->
Package['tomcat'] ~>
Class['tomcat::service'] ~>
Service['tomcat']
```

Notice that `Class['::tomcat']` now has a relationship with it's child `Class['::tomcat::package']`, where this relationship did not exist in Example 5-11

Although the `contain` function automatically includes the class being contained, it can be combined with Class style resource declarations. Doing so is parse order dependent, which gives this soltion the air of a hack. Regardless, this approach is currently the best practices solution to handling containment and is the officially recommended approach to building modules. See Example 5-14 for an example.

Example 5-15. Contain with resource style class declarations and chaining

```
class { '::tomcat::package':
  ensure => $ensure,
  source => $source,
}
class { '::tomcat::service':
  ensure => $ensure,
  enable => $enable,
}

contain '::tomcat::package'
contain '::tomcat::service'

Class['::tomcat::package'] ~>
Class['::tomcat::service']
```

The contain keyword was added to Puppet in version 3.4.0. If you are writing modules for modern releases of Puppet, it is recommended that you use the `contain` function in your classes rather than the anchor pattern.

Anchors

The anchor pattern is the original solution to the class containment problem. Anchors are a resource type provided by the *puppetlabs/stdlib* module. Anchors themselves perform no actions, but they do provide an anchor with which to establish class relation-

ships inside a modoule. They also pass along notify signals, so that notification works between modules as expected.

Here is `Class[::tomcat]` with anchors:

Example 5-16. Class[::tomcat] using the Anchor pattern

```
class tomcat {
  anchor { '::tomcat::begin'} ->
  class { '::tomcat::package': } ~>
  class { '::tomcat::service': } ~>
  anchor { '::tomcat::end'}
}
```

Although this seems to be somewhat simpler than our containment example, it carries a huge amount of extra complexity in ensuring that our resource relationships behave the way we expect. Example 5-16 contains a subtle bug; `Anchor[::tomcat::begin]` does not have a notify relationship with `Anchor[::tomcat::service']`. As a result, notifications sent to the Tomcat module would not cause the tomcat service to restart. This might be an issue if, for example, you updated Java to patch a vulnerability using Puppet, and the Tomcat service remained resident in memory running under the old release of Java because it's service resource never received a notification to restart.

Beyond that, the anchor pattern creates some ugly resource relationship graphs that can be painful to read when attempting to analyze Puppet's graph output.

Intentionally uncontained classes

There are some cases where you may wish to intentionally avoid containing resources. Consider a case where we need to insert the deployment of one application after it's dependent module has been installed and configured, but before the service from its dependent service has been started.

Example 5-17. Class[::tomcat] using the Anchor pattern

```
include ::tomcat
include ::my_tomcat_app

Class['::tomcat'] ->
Class['::my_tomcat_app'] ~>
Class['::tomcat::service']
```

This example will only work if `Class[::tomcat::service]` is *not* contained inside of `Class[::tomcat]`. Otherwise, a dependency loop would be created; `Class[::tomcat] -> Class[::tomcat::service] -> Class[::my_tomcat_app] -> Class[::tomcat::service]`

Internally, the rest of the tomcat module may have a notify relationship with `Class[::tomcat::service]` and a relationship loop will not be created. We could create such a module using this basic layout:

Example 5-18. Class[::tomcat] using the Anchor pattern

```
class tomcat {
  contain '::tomcat::install'
  contain '::tomcat::config'
  include '::tomcat::service'

  Class['::tomcat::install'] ->
  Class['::tomcat::config'] ~>
  Class['::tomcat::service']
}
```

This works because resource relationships do not require that one resource be evaluated immediately after another when a relationship is defined. The relationship simply sets general ordering requirements, and allows for other resources to be inserted into the order. Puppet maintains the relationship ordering internally using a dependency graph.

When using an approach such as this, remember that you lose the ability for the un-contained resources to generate notifications for the parent class; anything that wishes to subscribe to `Class[::tomcat::service]` must do so explicitly now.

Because such module designs do not conform to the typical design pattern for a module, it's critical to test and document this special behavior, and to treat the un-contained class as an interface into the module; not to be changed without planning for the break in compatibility.

Inter-module relationships like this are typically domain logic and should typically handled in your profiles rather than hard-coded into your modules.

This approach is useful when you need to complex relationships between arbitrary modules. In many cases, it's much better to use a defined type as an interface, as discussed in "Module Interfaces with defined types" on page 108.

Interfacing with subclasses.

There are three popular ways to pass data from your modules init class to it's subclasses:

1. Use a resource style class declaration with parameterized subclasses.
2. Use contain or include style class declaration with fully qualified variable references inside the subclass.
3. Use class inheritance.

Passing data via parameterized class declarations

This is my preferred approach for passing data from an init class to a subclass. This approach it makes variable handling extremely explicit and causes immediate failures in the event of a typo or hasty change.

Example 5-19. Passing data using parameterized subclasses and resource style class declaration

```
class apache (
  $ensure  => 'installed',
  $package => $::apache::params::package,
) {
  class { '::apache::install':
    ensure  => $ensure,
    package => $package,
  }
  contain ::apache::install
}

class apache::install (
  $ensure,
  $package,
) {
  package { $package:
    ensure => $ensure,
  }
}
```

With this approach, if either the `package` or `ensure` parameter are missing or if there is a typo, the declaration of `::apache::install` will throw an error. If the `::apache::install` class does not accept a parameter that's passed to it from the init class, an error is also thrown.

This pattern makes it very clear what variable is being used where, and explicitly states what will be affected by a parameter change.

The only downside of this class is the ugly combination of a class declaration and a `contain` function call.

Passing data via fully qualified variable references

With this approach, the `::apache::install` subclass pulls variables from the init class.

Example 5-20. Passing data using fully qualified variables

```
class apache (
  $ensure  => 'installed',
  $package => $::apache::params::package,
) {
  contain ::apache::install
}
```

```
class apache::install (
  $ensure  = $::apache::ensure,
  $package = $::apache::package,
) {
  package { $package:
    ensure => $ensure,
  }
}
```

The benefit of this approach is that inclusion of the `::apache::install` subclass and containment of the subclass are handled with a simple `contain` statement. The disadvantage of this approach is that the use of fully qualified variables is a bit more verbose, and the consumption of values from init is no longer visible within init, which somewhat mitigates the use of init to view the flow of a module.

Passing data via class inheritence

You can also pass variables using class inheritance. With this approach, variables local to the init class are available in the local scope of any class that inherits from the init class.

Example 5-21. Passing data using class inheritance

```
class apache (
  $ensure  => 'installed',
  $package => $::apache::params::package,
) {
  contain ::apache::install
}

class apache::install inherits ::apache {
  package { $package:
    ensure => $ensure,
  }
}
```

This approach carries the least initial development overhead. Variables are declared in the init module, and are automatically made available to any of the submodules as if they were declared locally. This effectively re-implements variable inheritance from the Puppet 2.x days, but does it in a more limited and controlled way.

The two major disadvantages of this approach are that you won't be able to see or control what variables are passed into a subclass, and that without it's own scope, you cannot re-use variable names declared in the parent classes.

While I do not personally recommend this pattern, it is used in a lot of official Puppetlabs modules, making it a fairly common and well understood approach for passing data into subclasses.

Defined Types

A defined type is a Puppet class that behaves similarly to a native Puppet resource.

Unlike a class, you can declare multiple instances of a defined type so long as those instances and their individual resources do not conflict with each other.

Defined types are commonly used in one of four ways within a module:

1. For iteration or the DRY principle
2. For creating interfaces into a module
3. As a service to the outside world
4. As the core purpose of a module

Iteration and DRY with defined types

Defined types may be used in a manor analogous to methods or functions from imperative languages. This use of defined types is fairly universal, and isn't really specific to module development. Both concepts are explored in Chapter 4; "Don't Repeat Yourself (the DRY principle)" on page 52 and "Iteration" on page 72

Module Interfaces with defined types

There are many situations where you may need to create complex relationships between modules. The principle of interface driven design [4] would strongly advise against accessing the data structures and referencing resources from other modules directly.

In "Intentionally uncontained classes" on page 104 we discuss the use of un-contained classes as an interface. When applicable, the use of defined types as an interface into your module is the preferable approach for handling inter-module dependencies. Defined types are often used as an interface into a module, making this the obvious choice for building interfaces. Defined types provide a clean parameterized interface that can be defined and tested, allowing the internal structure of the module to change without breaking dependent modules.

In "Interface Driven Design" on page 51 we give this example of the use of a defined type as an interface into an apache module:

Example 5-22. Interface Driven Design vhost template

```
class myapp {
  contain ::myapp::config
```

4. see: "Interface Driven Design" on page 51

```
  apache::custom_vhost { 'myapp':
    content => template('myapp/vhost.conf.erb'),
  }

  Class['::myapp::config'] ->
  Apache::Custom_vhost['myapp']
}
```

In Example 5-22, the class `::myapp` is interfacing with the Apache module using the `apache::custom_vhost` defined type. The internal structure of that defined type might look something like Example 5-23:

Example 5-23. Defined type vhost interface

```
define apache::custom_vhost (
  $servername = $title,
  $content    = undef,
) {
  $_content = pick($content, template('apache/custom_vhost.erb'))

  file { '/etc/httpd/conf.d/${title}.conf':
    ensure  => 'file',
    content => $_content,
  }

  Class['::apache::config'] ->
  Apache::Custom_vhost[$title] ~>
  Class['::apache::service']
}
```

Notice that all relationships and references to the internal structure of the apache module are entirely contained with the defined type `apache::custom_vhost`. We could completely re-write the apache module, and so long as the defined type exists and continues to provide the `servername` and `content` parameters the `::myapp` class will continue to work without change.

If we've documented `apache::custom_vhost` as an interface into our module and written spec tests for it, we can know without looking at any other module what changes are safe to make to the apache module, and what changes are not safe. This flexibility is the huge benefit of interface driven design. Without it, many sites are slow to adapt to new needs because changes to one module often break code elsewhere.

Providing services with defined types

In many cases your module may be able to provide a useful feature to the outside world via a defined type. The Apache module we've been looking at through this section is a great example of this; our module can provide a defined type that handles the OS specific implementation details of configuring a virtual host, and provides a useful template and set of properties for defining the host.

This defined type doesn't prevent other modules from implementing their own vhost templates, but it does offer a simple way to define a virtual host that's integrated nicely into our module, and handles most common use cases.

Example 5-24. A conveince defined type for managing Virtual Hosts

```
define apache::vhost (
  $ensure         = 'present',
  $documentroot   = undef,
  $errorlog       = undef,
  $port           = 80,
  $servername     = $title,
) {
  $vhost_dir = $::osfamily ? {
    'RedHat' => '/etc/httpd/conf.d',
    'Debian' => '/etc/apache2/sites-available',
    default  => '/etc/httpd/conf.d',
  }

  $link_ensure = $ensure ? {
    'absent' => 'absent',
    default  => 'link',
  }

  file { "${vhost_dir}/${title}.conf":
    ensure  => $ensure,
    content => template('apache/vhost.conf.erb'),
    group   => 'root',
    owner   => 'root',
    mode    => '0444',
  }

  if $::osfamily == 'Debian' {
    file { "/etc/apache2/sites-enabled/${title}.conf":
      ensure => $link_ensure,
      target => "../sites-available/${title}.conf",
    }
  }

  Class['::apache::install'] ->
  Apache::Vhost[$title] ~>
  Class['::apache::service']
}
```

As in "Module Interfaces with defined types" on page 108, the use of a defined type is much better than trying to manage virtual hosts as files, because the defined type can establish relationships into the apache module and access data from inside that module without violating the separation of concerns and the principles of interface driven design.

Defined types for simplifying complex tasks

There are often cases where a defined type can be used to provide a clean and simple interface for complex operations.

Here's an example of a module for managing network service names. It uses Augeas to manage `/etc/services` and a defined type to provide a clean interface around the Augeas resource.

Example 5-25. An Augeas resource wrapped in a defined type

```
define network_service_name (
  $port,
  $protocol      = 'tcp',
  $service_name  = $title,
  $comment       = $title,
) {
  $changes = [
    "set service-name[last()+1] ${service_name}",
    "set service-name[last()]/port ${port}",
    "set service-name[last()]/protocol ${protocol}",
    "set service-name[last()]/#comment ${comment}",
  ]

  $match = "service-name[port = '${port}'][protocol = '${protocol}']"
  $onlyif = "match ${match} size == 0"

  augeas { "service-${service_name}-${port}-${protocol}":
    lens    => 'Services.lns',
    incl    => '/etc/services',
    changes => $changes,
    onlyif  => $onlyif,
  }
}
```

The defined type Example 5-24 can now be used to declare service names without having to go through the somewhat error-prone process of reimplementing this Augeas statement each time.

This approach helps a mixed team leverage experience. A more experienced module creator can produce defined types such as this one to manage complex tasks, allowing less senior team members use much more friendly resource declarations. The above defined_type can now be declared very easily, as in Example 5-26.

Example 5-26. Declaration of a netwok_service_name defined type

```
network_service_name { 'example':
  port => '12345',
  protocol => 'tcp',
}
```

In an example like this, the defined type may be in the init manifest of the module, and the module may contain no other manifests.

These kinds of defined type modules provide convenience features and extensions similar to native Puppet resource types. You may chose in this case to utilize them directly in your other other modules, creating inter-module dependencies. In these cases, the features your defined types create may truly be part of your application logic rather than your domain logic.

Interacting with the rest of the module

When a defined type is included in a an application module, it often needs to interact with the rest of the module. Care must be taken when doing so to avoid dependency loops, parse order dependence, and to keep the module simple to understand and use.

Resource relationships

As a general rule, the best place to put relationships between the defined type and the rest of your module is inside the defined type. While you can use resource collectors inside the module to declare relationships, resource collectors also have the behavior of realizing virtual resources, which may not be what you want.

Including other classes

A somewhat dangerous pattern is for the defined type to include its parent init class. This pattern is commonly used in cases where the defined type depends on resources and variables defined in the init class. Unfortunately, this pattern can create parse-order bugs. s

Another issue with this pattern is that it often results in the defined type becoming the de-facto entry point to your module, rather than the init class. This can be confusing to debug, especially with more complex modules.

Example 5-27. A conveince defined type for managing Virtual Hosts

```
defined example::defined_type {
  include example # ❶

  # Do other stuff
}
```

❶ This can create parse order dependent conflict if `class {'example': }` appears elsewhere in your code-base.

Instead, it's usually best to keep your defined types as small and self-contained as possible. Try to avoid declaring other classes or referencing variables from other contexts if at all possible.

Documentation

Documentation is an investment in the future. When you are in the middle of writing a module, the behavior of the module, it's inputs, and it's quirks are self-evident. When you've moved on, it's easy to forget what each parameter does, what kind of input the parameters accept, and what quirks exist in your module.

Documentation of a module takes three forms:

1. Documentation committed to your module's repository
2. In-line documentation within your module
3. In-line comments

The `puppet module generate` command produces documentation templates that can be a useful starting point to write your own documentation.

Markdown

Github, Gitlab and most other GIT collaboration tools provide built-in rendering of Markdown and `.txt` documentation, making documentation committed to your software repository one of the most user friendly ways of documenting your module for your users.

It's a good idea to include a README.markdown file containing the following information:

1. The name of the module
2. Example invocations of the module for common use cases.
3. A description of each supported parameter for your module
4. Notes about bugs or known issues (if you are not using a public issue-tracker)
5. Any dependencies your module may have, either internal or external to Puppet
6. Contact information
7. The license for your module, if any

Usage documentation

Except for the most trivial modules, it's a good idea to show some common usage examples of your modules. This can help the user test the module, can provide ideas as to how to use your module, and can help the user if they get stuck deploying the module, perhaps due to syntax errors in their input.

Usage documentation is also useful if you wish to highlight the most commonly set or modified parameters for your module. It's very common for a module to have several dozen parameters; this is the place your user will look for the most important parameters.

If your module has any dependencies or requires any setup, it's a good idea to provide an example of how to satisfy those dependencies as well.

Parameter documentation

Documenting your input parameters is key to writing a usable module. Parameter names are often terse, tend to be numerous, and can often be confusing.

When documenting an input parameter, it's good to provide the following information:

1. The name of the parameter
2. A brief description of the parameters purpose.
3. The types of data accepted.
4. The structure of the data if structured data is accepted.
5. The default value of the parameter.
6. Any constraints on the data enforced by the application, module design, or input validation.

For example:

Example 5-28. Markdown documentation for a parameter

```
== Module parameters

=== document_root
The path to your site's document root. Must be a fully qualified path. Defaults to: '/var/www/html'
```

You should only document input parameters you wish to support long term. Unsupported or deprecated parameters can be marked as such.

Dependencies

If your module depends on other modules, code, or setup, it's a very good idea to note those dependencies in your documentation. If the dependencies are very specific, noting the modules required, providing a link to those modules, and noting version compatibility is extremely valuable.

If your dependencies are generic (this module requires a web-server,) it's a good idea to mention the dependencies, and show how a commonly available module can be used to satisfy the dependency.

Your fixtures file, meta-data, and test.pp file also track these dependencies.

License information

If you plan to publicly release your module, it's a good idea to attach a license your it.

Enterprise users may have constraints placed on the code that can be deployed to their site. A license for the module may be required by their internal policies in order to permit use of the module. Some companies restrict what software license are acceptable for use internally, and may not be able to use code under more restrictive licenses such as GPL 3.

If you are writing modules in house, you may wish to clarify with your management or legal team regarding the license or restrictions that should be placed on the code. Some companies are fairly generous with their modules releasing them publicly once scrutinized and sanitized. Others businesses will prefer to keep internally developed Puppet modules proprietary. You will save a lot of headache by making node of these constraints in the documentation.

If you plan to release the module to the public, a license is a good way of communicating what other users may do with your code. Most public licenses permit redistribution of code, however the license used may limit the ability for others to fork your code.

If you're not sure what license to use, I recommend the Apache 2.0, which permits liberal modification and redistribution of your code, but requires that your authorship of the module be recognized and prevents others from patenting your work.

I do recommend however researching licenses to identify the license that best suits your requirements/wishes. You may find you prefer a more restrictive license such as the AFPL.

In-line documentation

In-line documentation is documentation embedded into the code of your module. A number of Puppet tools including the `puppet doc` command (3.x), `puppet strings` command (4.x) and Geppetto can consume and display this documentation. A benefit of in-line documentation is that Puppet can search for the docs in your module path, and Geppetto can display the documentation as hover-text.

In most cases, you can reuse your markdown documentation as in-line documentation, modifying formatting as needed. It may also be a good idea to document your unsupported and internal parameters. This can simplify development; you can use the built-in tools to check the documentation rather than keeping the class open in an editor.

In-line documentation usually takes the forms of comment blocks at the top of your manifest, but you should check the documentation for the tool you are using.

Rake tasks

Puppetlabs provides a number of extremely useful Rake tasks that can dramatically simplify module development.

Rake is Ruby's variant of the Make command. Once defined, Rake tasks can be executed from your module's root directory and any of it's sub directories using the `rake` command. Among other things, the Rakefile generated by the `puppet module` tool can be used to:

- Quickly validate the syntax of your module
- Invoke the puppet-lint command to validate your module follows the style guide
- Compile your module using rspec-puppet
- Build a package for the Puppet Forge

Using these tasks is a great way to perform simple sanity checks prior to deploying your module. As a general rule, you should lint and validate your module prior to pushing changes to upstream repositories.

Testing

Testing, like documentation, is an investment in the future of your module.

Module testing takes two common forms:

1. Unit testing
2. Acceptance testing

Unit testing is your `puppet-rspec` tests. Unit tests may be run on your local workstation, and are a quick way to validate that your changes have not altered your modules interfaces, broken support for other platforms, or otherwise generally broken your module (E.g. it no longer compiles.) While the lint and validate commands will ensure that your module parses, unit testing actually exercises the functions, logic, and resources in your module.

Where validate may not catch a typo in the parameter of a resource type, unit testing will. The limitation for unit testing is that it tests the logic of your modules; it doesn't validate external dependencies.

Acceptance tests are your rspec-system [5] and beaker tests. These tests spin up a virtual machine and ensure that your module deploys correctly, that the packages, services, and

5. rspec-system is deprecated; use breaker

other resources in your module actually exist, that the syntax in your configuration files and templates are valid, and that the service your modules manage actually comes up and listens on the expected port. Integration testing is very slow, but and comprehensive; it will catch dependencies and ordering problems that cannot be identified in other ways.

Rspec

Rspec tests are performed entirely within the constructs of the ruby interpreter, with no impact on the host system other than the need to install ruby gem and puppet module dependencies.

Rspec tests are a quick and clean way to perform regression and smoke tests on your code, and with limited exceptions, need not be run on the platform your module targets. For example, puppet-rspec can test a module targeted for Debian and RedHat systems on a Apple OSX host without issue.

Unfortunately, Puppet hard codes several special behaviors for Windows hosts; UNIX tests will often fail on Windows workstations, and visa versa. Use a Vagrant box for rspec testing in these situations.

Ideally, your rspec tests should accomplish four things:

1. Validate that your modules inputs match your documentation
2. Validate that the output of your module matches your expectations.
3. Test input validation
4. Prevent regression.

Your rspec tests do not need to validate the internal state of your module, and need not be absolutely strict about the output. Overly strict testing with heavy focus on the internal structure of your module will neadlessly slow the development process.

Input testing with rspec

Input validation ensures compatibility with your documentation, and ensures that you do not break compatibility within a major release of your module. Writing good input validation tests provide the freedom to modify the module with the confidence of knowing that if you break compatibility, you will know.

I recommend performing the following tests for each documented parameter of your module:

1. Validate your modules behavior with the default parameter value.
2. Validate your modules behavior with a valid input supplied to the parameter.

3. Validate your modules behavior with an invalid input supplied to the parameter.

Example 5-29. Input validation

```
require 'spec_helper'

describe 'apache' do
  context "on RedHat" do
    let(:facts) do
      { :osfamily => 'RedHat' }
    end

    context 'with document_root => unset' do
      it { is_expected.to contain_file('/etc/httpd/conf/httpd.conf').with(
          { :content => %r{DocumentRoot /var/www/html}
        )
      }
    end

    context 'with document_root => /tmp' do
      it { is_expected.to contain_file('/etc/httpd/conf/httpd.conf').with(
          { :content => %r{DocumentRoot /tmp}
        )
      }
    end

    context "with document_root => false" do
      let :params do
        { :documentroot => false }
      end
      it { expect{ is_expected.not_to compile }.to
        raise_error(Puppet::Error, /not an absolute path/)
      }
    end
  end
end
```

Resource validation

In most cases, input validation will by it's nature check the output of the module. However, there are often resources in your module that are not affected by any of your input parameters. Those resources should be tested explicitly.

Example 5-30. Output validation

```
require 'spec_helper'

describe 'apache' do
  context "on RedHat" do
    let(:facts) do
      { :osfamily => 'RedHat' }
    end
```

```
    it { is_expected.to contain_service('httpd').with({
        :ensure => 'running',
        :enable => true,
      })
    }
  end

  context "on Debian" do
    let(:facts) do
      { :osfamily => 'Debian' }
    end

    it { is_expected.to contain_service('apache2').with({
        :ensure => 'running',
        :enable => true,
      })
    }
  end
end
```

Testing input validation

If your module performs input validation, I strongly advise testing your input validation to ensure that it behaves as expected. Nothing is more frustrating than having a valid parameter input rejected because of a poorly designed regular expression validation string.

In most cases, it's fine to design your module first and your tests after. For input validation, I strongly recommend using test-driven development; write test cases test both positive and negative outcome from your input validation scripts, and then write your validation. In many cases, it's much easier to design tests than it is to build a validation regular expression that handles every single case correctly on the first pass.

Example 5-31. Testing input validation

```
require 'spec_helper'

describe 'apache' do
  protocols = [ '', 'http://', 'ftp://' ]
  hostnames = [
    'localhost',
    'localhost.localdomain',
    '1.2.3.4',
  ]
  ports = [ '', ':80', ':8080' ]

  # Positive tests
  protocols.each do |protocol|
    hostnames.each do |hostname|
      ports.each do |port|
```

```
        servername = "#{protocol}#{hostname}#{port}"
        context "with servername => '#{servername}'" do
          let :params do
            { :servername => servername }
          end
          it { is_expected.to contain_file('/etc/httpd/conf/httpd.conf').with(
            { :content => /ServerName servername/ }
          )
        end
      end
    end
  end

  # Negative tests
  protocols = [ '', 'http:/', '://', 'http:', '1://', '//:' ]
  hostnames = [ '', 'localhost' ]
  ports     = [ '', ':', ':80' ]

  protocols.each do |protocol|
    hostnames.each do |hostname|
      ports.each do |port|
        servername = "#{protocol}#{hostname}#{port}"
          unless servername == 'localhost' || servername == 'localhost:8080'
            context "servername => '#{servername}'" do
            let :params do
              { :servername => servername }
            end
            it { expect{ is_expected.not_to compile }.to
              raise_error(Puppet::Error, /validate_re/)
            }
          end
        end
      end
    end
  end
end
```

Acceptance testing

It is a good idea to write acceptance tests for your modules. Acceptance tests provide a quick way to apply your module to a real running system, and can catch issues related to ordering, idempotence, undeclared dependencies, and external resources that cannot be reliably tested in any other way.

With acceptance tests, the functionality of the applied module and the return codes from applying the module. For example, with an `apache` module, you might wish to ensure that the apache package is present on the system, the service is running, and that Apache responds with a boilerplate site when queried using Curl.

It's also very valuable to ensure that Puppet returns an exit code indicating that it modified system state with no errors upon initial invocation, and that further invocations produced no change. These tests will quickly ensure that your module is idempotent when applied to a live system.

Example 5-32. Beaker Acceptance tests for an Apache module

```
require 'spec_helper_acceptance'

describe 'apache' do
  apply_manifest('include apache', :catch_failures => true)
  apply_manifest('include apache', :catch_changes => true)

  describe package('httpd') do
    it { is_expected.to be_installed }
  end

  describe service('httpd') do
    it { is_expected.to be_running }
  end

  describe port(80) do
    it { should be_listening }
  end
end
```

Beaker supports multiple back-ends for acceptance testing. The most commonly used back-end is Vagrant, and is compatible with your existing boxes. However, Beaker can build and invoke tests on VMWare, AWS, OpenStack, and other infrastructure as well.

When writing beaker tests, it's important to begin with a clean system image to ensure that your module is not implicitly dependent on existing system state to run.

If your site uses a custom base image, I advise using a clean copy of that image as a base for testing; often local security policy can create restrictions that are not present on public boxes. You will also benefit if the base box has pre-configured local package repositories for your site.

If you intend to publicly release the module, ideally the module should be tested against a clean publicly available system image in order to ensure maximum compatibility.

Module testing best practices

As a best practice, module tests should be self contained within the module. I recommend against using site-wide Hiera data to build a test environment. Instead, declare your modules parameters inside your test cases, or use manifests inside your module's `tests` directory where applicable.

The `puppetlabs_spec_helper` gem can handle the retrieval of your module dependencies, and the creation of a test modules directory.

Testing outside the scope of your overall site is a good way to identify assumptions about your module, and to catch unexpected dependencies.

We will explore end to end testing of your site logic and Hiera data in [Link to Come].

Continuous Integration

With automated testing in place, it's fairly straightforward to perform CI testing on your modules.

It's a good idea to build several gemsets for your module so that you can test on different rubies and with different versions of Puppet. Even if your site currently uses Puppet 3, rspec testing your modules with Puppet 4 is a good way to catch and resolve compatibility problems early in your development cycle. Resolving problems in this iterative way is a great way to simplify the upgrade process and to avoid massive upgrade projects when the time finally comes to upgrade.

Dependencies

In many cases, your module will have dependences on other modules. For example, `puppetlabs/stdlib` is a nearly universal dependency due to the number of useful function calls it provides.

Module dependencies are typically created with other modules that extend Puppet; modules containing custom functions, custom types and providers, and in some cases useful defined types.

In these situations, the dependencies should be documented in your modules `README.markdown` file [6], listed in your modules meta-data, and inserted into your modules `.fixtures` file for easy deployment.

The `puppetlabs_spec_helper` gem provides a number of useful rake tasks for managing and deploying module dependencies in such a way that your module can be easily tested, without the need to clone and deploy everything in your site. This clean approach helps document your module, and allows flexibility in compatibility testing between module versions.

I strongly recommend including a Gemfile and RVM meta-data in your module to simplify the deployment of these and other useful gems. The `puppet module gener`

6. see "Documentation" on page 113

`ate` tool will create the appropriate Gemfile and meta-data for you when generating a new module.

Summary

In this chapter we learned how to write clean modules that confirm to the best practices discussed in Chapter 4

Takeaways from this chapter:

- Carefully design your module to limit its scope
- Structure your module to make it easy to understand and simple to deploy
- Design interfaces into your module and use them
- Document your module for future maintenance
- Use test cases in order to prevent regressions and ensure compatibility

CHAPTER 6

Built-in Types