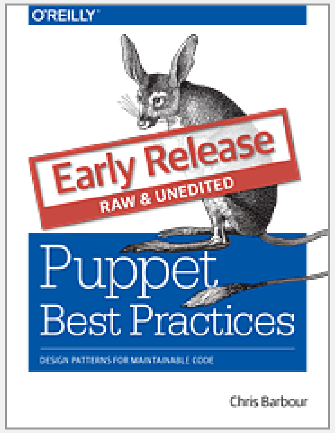
The Puppet design philosophy

Explore the declarative, idempotent, and stateless Puppet DSL.

by Brian Anderson | @branderlog | April 21, 2015



Buy *Puppet Best Practices*.

It can be very easy to get started with Puppet, but scaling it effectively can be a challenge. In this ***early release excerpt*** *from* Puppet Best Practices*, author Chris Barbour discusses the philosophy behind using Puppet effectively.*

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 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.

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) verbless. 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 is ideal for configuration baselining 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 5.

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 workflow. 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 your understanding how to read a map or use a GPS device, Puppet's declarative model relies on its 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, its declarative equivalent in Puppet, and the benefits of using Puppet rather than a shell script for this task.

Imperative / 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 1-1. 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 1-2. 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 1-3. 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 1-4. 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\ JPNV
ngkd2R9rA0vQ7Qx/XrZ0hgGpBA99AkxEnMSuFrD/E5TunvRHIczaI9Hy0IMXc= \ alice@localhost" >>
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 1-5. 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 1-6. 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 1-7. Ensure that a user is absent using Puppet

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

group { 'examplegroup':
    ensure => 'absent', # (1)
}

user { 'alice':
    ensure      => 'absent', # (2)
    gid         => 'examplegroup',
    groups      => 'wheel',
    managehome  => true,
}

ssh_authorized_key { 'alice@localhost':
    ensure  => 'absent', # (3)
    user    =>'alice',
    type    =>'ssh-rsa',
    key     => $ssh_key,
}

Ssh_authorized_key['alice@localhost'] -> # (4)
User['alice'] -> # (5)
Group['examplegroup']
```

(1), (2), (3) Ensure values are changed from Present to Absent.

(4), (5) 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 5.

You may notice the addition of resource ordering in this example when it wasn't 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 1-6.

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 Internet 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 factor and plugin sync 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 passphrase, 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 its declarative resource management model, however Puppet also enforces a number of rules on its variable handling, iterators, and conditional logic to maintain its 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 filesystem 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 1-8. 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 1-9. 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 1-10. 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 1-11. 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 1-12. 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 1-13. 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 its 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.

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 5.

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 1-14. 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 its 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 1-15. 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 only if properly will help produce safe and robust exec resources. We will explore this in depth in Chapter 5.

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

Example 1-16. 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 model of change becomes just as important as whether or not the resources are functionally idempotent. Remember that before performing any configuration change, Puppet will first determine whether or not the resource currently conforms to policy. Puppet will only make a change if resources are in an inconsistent 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 its second invocation on a fresh system, your Puppet codebase 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 1-17. 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 1-17, 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 its 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 1-18. 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.

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

Example 1-18 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 its 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 has 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 its 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 datacenter 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 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 its state as part of its 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

Filebucketing is an interesting and perhaps underappreciated feature of the File resource type. If a filebucket 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 filebucketing to display the contents of managed files.

Filebuckets can also be used for content distribution, however using a filebucket 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 datacenters, when rebuilding masters. Use of filebucketing 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 Chapter 2. For the time being, just be aware that exported resources introduce a source of state into your infrastructure.

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

Example 1-19. Declaring state with an exported resource

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

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

Example 1-20. 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.

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 Chapter 6.

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.

There are plugins 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. 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 its 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

---

Editor's note: Puppet 4 was recently released. If you're interested in seeing what it takes to upgrade an existing Puppet installation, don't miss Chris Barbour's Rehabilitating an Existing Puppet Site webcast at 10AM PT on Wednesday, April 22, 2015.

tags: best practices, devops, DSL, Puppet