Puppet 4 Essentials
Second Edition

Puppet 4 Essentials, Second Edition gets you started quickly as you’ll put Puppet’s tools to work right away. It also highlights the changes associated with performance improvements, as well as the new language features in Puppet 4.

We’ll start with a quick introduction to Puppet to quickly get you managing your IT systems. You will then learn about the Puppet Agent, which comes with an all-in-one (AIO) package and can run on multiple systems. Next, we’ll show you the Puppet Server for high-performance communication. As you progress through the book, the innovative structure and approach of Puppet will be explained with powerful use cases. The difficulties that are inherent in a complex and powerful tool will no longer be a problem for you as you discover Puppet’s fascinating intricacies.

By the end of the book, you will not only know how to use Puppet, but also its companion tools Facter and Hiera, and will be able to leverage the flexibility and expressive power implemented by their toolchain.

Who this book is written for
If you’re an experienced IT professional and a new Puppet user, this book will provide you with all you need to know to go from installation to advanced automation in no time at all. Get a rapid introduction to the essential topics and then tackle Puppet for advanced automation.

What you will learn from this book
- Write and employ individual Puppet manifests
- Understand how Puppet implements system abstraction
- Deploy the Puppet master and agents
- Leverage and extend Facter to summarize your systems
- Create modular and reusable Puppet code
- Extend your code base with publicly available Puppet modules
- Use the modern features of Puppet 4
- Separate logic from data using Hiera

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Writing Your First Manifests'
- A synopsis of the book’s content
- More information on Puppet 4 Essentials Second Edition
About the Authors

**Felix Frank** has used and programmed computers for most of his life. During and after working on his computer science diploma, he gained experience on the job as a systems administrator, server operator, and open source software developer. He spent 6 years of his 11-year career as a Puppet power user. In parallel, he spent about two years intensifying his studies through ongoing source code contributions and active participation in several conferences.

**Martin Alfke** is co-founder and CEO of example42 GmbH. He is a Puppet and automation enthusiast since 2007. In the past, he would have said he is a "system administrator". Nowadays, he likes the term "infrastructure engineer". The big difference is that system administrators sshs into systems to fix them, whereas infrastructure engineers fix their automation.

Martin is serving as a Puppet Labs' training partner in Germany since 2011, conducting official Puppet training. In his spare time, he likes reading books and visiting interesting locations.
The software industry is changing and so are the related fields. Old paradigms are slowly giving way to new roles and shifting views on what the different professions should bring to the table. The DevOps trend pervades more and more workflows. Developers set up and maintain their own environments. Operations raise automation to new levels and translate whole infrastructures to code.

A steady stream of new technologies allows for ever-evolving, more efficient organizational principles. One of these is Puppet. Its fresh take on server configuration management caused a rapid adoption and distribution throughout the industry. In the few years of its existence, Puppet has managed to rally thousands of users who employ it in numerous contexts to achieve manifold results. While it is not the only configuration management system available, it is certainly the most widespread by now.

From its specialized language, to the system that makes it work, Puppet has innovated and rapidly conquered. Its extendible structure, paired with a large and helpful community, has made Puppet a powerhouse of easy configuration. The more well known a software is, the greater the chance that Puppet will deploy and configure it out of the box.

Puppet's own learning curve is not sharp, and the available documentation is not only extensive, but also of high quality. Nevertheless, even experienced programmers and administrators can face difficulties at some point. Advanced use may require the navigation of some intricacies that stem from Puppet's unique modeling approach.

This book aims to impart all the knowledge required to tap into not only the basics of Puppet, but also its core. The very ideas and principles of Puppet-based designs are explored and explained. Sophisticated tooling is presented to enable efficient and productive use. The reader is rapidly introduced to and familiarized with a whole range of Puppet-centric technology.
Preface

What this book covers

Chapter 1, Writing Your First Manifests, is an introduction to the core concepts of Puppet, including a syntax tutorial. You will learn to write and use Puppet manifests within a few pages.

Chapter 2, The Master and Its Agents, is a quick how-to guide to setting up all the required components for a distributed Puppet infrastructure. In this chapter, you will learn to create the master and an agent, and set up puppetserver.

Chapter 3, A Peek under the Hood Facts, Types, and Providers, covers a summary of the core components that give Puppet its flexibility, as well as its ease of use and extension. You will also be able to understand the principles of Puppet’s function.

Chapter 4, Modularizing Manifests with Classes and Defined Types, helps you learn the most important structural elements of Puppet manifests. Find out how to use them for best results.

Chapter 5, Extending Your Puppet Infrastructure with Modules, takes you on a tour of downloadable extensions, with a guide to making your own. Extend Puppet to your specific needs.

Chapter 6, Leveraging the Full Toolset of the Language, teaches you to interpret and use language features beyond the basics.

Chapter 7, New Features from Puppet 4 helps you learn about the new Type system and data processing features. It also helps you to understand the deprecations and what to change to make use of the new features.

Chapter 8, Separating Data from Code Using Hiera, gives you an introduction to the powerful configuration data storage that comes with Puppet. In this chapter, you will learn to model your infrastructure naturally in a hierarchy.
Writing Your First Manifests

Over the last few years, configuration management has become increasingly significant to the IT world. Server operations in particular is hardly even feasible without a robust management infrastructure. Among the available tools, Puppet has established itself as one of the most popular and widespread solutions. Originally written by Luke Kanies, the tool is now distributed under the terms of Apache License 2.0 and maintained by Luke's company, Puppet Labs. It boasts a large and bustling community, rich APIs for plugins and supporting tools, outstanding online documentation, and a great security model based on SSL authentication.

Like all configuration management systems, Puppet allows you to maintain a central repository of infrastructure definitions, along with a toolchain to enforce the desired state on the systems under management. The whole feature set is quite impressive. This book will guide you through some steps to quickly grasp the most important aspects and principles of Puppet.

In this chapter, we will cover the following topics:

- Getting started
- Introducing resources and properties
- Interpreting the output of the puppet apply command
- Adding control structures in manifests
- Using variables
- Controlling the order of evaluation
- Implementing resource interaction
- Examining the most notable resource types
Getting started
Installing Puppet is easy. On large Linux distributions, you can just install the Puppet package via `apt-get` or `yum`.

The installation of Puppet can be done in the following ways:

- From default Operating System repositories
- From Puppet Labs

The former way is generally simpler. *Chapter 2, The Master and Its Agents*, provides simple instructions to install the Puppet Labs packages. A platform-independent way to install Puppet is to get the `puppet` Ruby gem. This is fine for testing and managing single systems, but is not recommended for production use.

After installing Puppet, you can use it to do something for you right away. Puppet is driven by manifests, the equivalent of scripts or programs, written in Puppet's domain-specific language (DSL). Let's start with the obligatory *Hello, world!* manifest:

```
# hello_world.pp
notify { 'Hello, world!': }
```

To put the manifest to work, use the following command. (We avoided the term "execute" on purpose—manifests cannot be executed. More details will follow around the middle of this chapter.):

```
root@puppetmaster:~# puppet apply hello_world.pp
```

Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.45 seconds
Notice: Hello, world!
Notice: /Stage[main]/Main/Notify[Hello, world!]/message: defined 'message' as 'Hello, world!'
Notice: Applied catalog in 0.03 seconds
Before we take a look at the structure of the manifest and the output from the puppet apply command, let's do something useful, just as an example. Puppet comes with its own background service. Let's assume that you want to learn the basics before letting it mess with your system. You can write a manifest to have Puppet make sure that the service is not currently running and will not be started at system boot:

```puppet
# puppet_service.pp
service { 'puppet':
    ensure => 'stopped',
    enable => false,
}
```

To control system processes, boot options, and the like, Puppet needs to be run with root privileges. This is the most common way to invoke the tool, because Puppet will often manage OS-level facilities. Apply your new manifest with root access, either through sudo, or from a root shell, as shown in the following transcript:

```
root@puppetmaster:~# puppet apply puppet_service.pp
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.61 seconds
Notice: /Stage[main]/Main/Service[puppet]/ensure: ensure changed 'running' to 'stopped'
Notice: Applied catalog in 0.15 seconds
```

Now, Puppet has disabled the automatic startup of its background service for you. Applying the same manifest again has no effect, because the necessary steps are already complete:

```
root@puppetmaster:~# puppet apply puppet_service.pp
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.62 seconds
Notice: Applied catalog in 0.07 seconds
```

This reflects a standard behavior in Puppet: Puppet resources are idempotent—which means that every resource first compares the actual (system) with the desired (puppet) state and only initiates actions in case there is a difference (configuration drift).

You will often get this output, as shown previously, from Puppet. It tells you that everything is as it should be. As such, this is a desirable outcome, like the all clean output from git status.
Introducing resources and properties

Each of the manifests you wrote in the previous section declared one respective resource. Resources are the elementary building blocks of manifests. Each has a type (in this case, notify and service, respectively) and a name or title (Hello, world! and puppet). Each resource is unique to a manifest, and can be referenced by the combination of its type and name, such as Service["puppet"]. Finally, a resource also comprises a list of zero or more attributes. An attribute is a key-value pair, such as "enable => false".

Attribute names cannot be chosen arbitrarily. Each resource type supports a specific set of attributes. Certain parameters are available for all resource types (metaparameters), and some names are just very common, such as ensure. The service type supports the ensure property, which represents the status of the managed process. Its enabled property, on the other hand, relates to the system boot configuration (with respect to the service in question).

Note that we have used the terms attribute, property, and parameter in a seemingly interchangeable fashion. Don't be deceived—there are important distinctions. Property and parameter are the two different kinds of attributes that Puppet uses. You have already seen two properties in action. Let's look at a parameter:

```
service { 'puppet':
  ensure   => 'stopped',
  enable   => false,
  provider => 'upstart',
}
```

The provider parameter tells Puppet that it needs to interact with the upstart subsystem to control its background service, as opposed to systemd or init. If you don't specify this parameter, Puppet makes a well-educated guess. There is quite a multitude of supported facilities to manage services on a system. You will learn more about providers and their automatic choosing later on.

The difference between parameters and properties is that the parameter merely indicates how Puppet should manage the resource, not what a desired state is. Puppet will only take action on property values. In this example, these are ensure => 'stopped' and enable => false. For each such property, Puppet will perform the following tasks:

- Test whether the resource is already in sync with the target state
- If the resource is not in sync, it will trigger a sync action
A property is considered to be in sync when the system entity that is managed by
the given resource (in this case, the upstart service configuration for Puppet) is in
the state that is described by the property value in the manifest. In this example, the
ensure property will be in sync only if the puppet service is not running. The enable
property is in sync if upstart is not configured to launch Puppet at system start.

As a mnemonic concerning parameters versus properties, just remember that
properties can be out of sync, whereas parameters cannot.

Puppet also allows you to read your existing system state by using the puppet
resource command:

root@puppetmaster:~# puppet resource user root
user { 'root':
   ensure => 'present',
   comment => 'root',
   gid => '0',
   home => '/root',
   password => '$6$17/7FtU/$TvYEDtFgGr0SaS7xOVloWXVTqQxuDugH.eBKJ7bgHJ.hdoc03Xrvm2ru0HFKpu1QSvVW/7o.rLdk/9MZANEGt/',
   password_max_age => '99999',
   password_min_age => '0',
   shell => '/bin/bash',
   uid => '0',
}

Please note that some resource types will return read-only attributes (for example,
the file resource type will return mtime and ctime). Refer to the appropriate
type documentation.

**Interpreting the output of the puppet apply command**

As you have already witnessed, the output presented by Puppet is rather verbose.
As you get more experienced with the tool, you will quickly learn to spot the crucial
pieces of information. Let’s first take a look at the informational messages though.
Apply the service.pp manifest once more:

root@puppetmaster:~# puppet apply puppet_service.pp
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.48 seconds
Notice: Applied catalog in 0.05 seconds

Puppet took no particular action. You only get two timings: one, from the compiling phase of the manifest, and the other, from the catalog application phase. The catalog is a comprehensive representation of a compiled manifest. Puppet bases all its efforts concerning the evaluation and syncing of resources on the content of its current catalog.

Now, to quickly force Puppet to show you some more interesting output, pass it a one-line manifest directly from the shell. Regular users of Ruby or Perl will recognize the call syntax:

```bash
# puppet apply -e'service { "puppet": enable => true, }'
```

Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.62 seconds
Notice: /Stage[main]/Main/Service[puppet]/enable: enable changed 'false' to 'true'
Notice: Applied catalog in 0.12 seconds.

We prefer double quotes in manifests that get passed as command-line arguments, because on the shell, the manifest should be enclosed in single quotes as a whole.

You instructed Puppet to perform yet another change on the Puppet service. The output reflects the exact change that was performed. Let's analyze this log message:

- The **Notice:** keyword at the beginning of the line represents the log level. Other levels include **Warning**, **Error**, and **Debug**.
- The property that changed is referenced with a whole path, starting with `Stage[main]`. Stages are beyond the scope of this book, so you will always just see the default of `main` here.
- The next path element is `Main`, which is another default. It denotes the class in which the resource was declared. You will learn about classes in *Chapter 4, Modularizing Manifests with Classes and Defined Types*.
- Next, is the resource. You already learned that `Service[puppet]` is its unique reference.
- Finally, `enable` is the name of the property in question. When several properties are out of sync, there will usually be one line of output for each property that gets synchronized.
The rest of the log line indicates the type of change that Puppet saw fit to apply. The wording depends on the nature of the property. It can be as simple as created, for a resource that is newly added to the managed system, or a short phrase such as changed false to true.

Dry-testing your manifest
Another useful command-line switch for puppet apply is the --noop option. It instructs Puppet to refrain from taking any action on unsynced resources. Instead, you only get a log output that indicates what will change without the switch. This is useful in determining whether a manifest would possibly break anything on your system:

```
root@puppetmaster:~# puppet apply puppet_service.pp --noop
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.63 seconds
Notice: /Stage[main]/Main/Service[puppet]/enable: current_value true, should be false (noop)
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Applied catalog in 0.06 seconds
```

Note that the output format is the same as before, with a (noop) marker trailing the notice about the sync action. This log can be considered a preview of what will happen when the manifest is applied without the --noop switch.

The additional notices about triggered refreshes will be described later and can be ignored for the moment. You will have a better understanding of their significance after finishing this chapter and Chapter 4, Modularizing Manifests with Classes and Defined Types.

Adding control structures in manifests
You have written three simple manifests while following the instructions in this chapter so far. Each comprised only one resource, and one of them was given on the command line using the -e option. Of course, you would not want to write distinct manifests for each possible circumstance. Instead, just as how Ruby or Perl scripts branch out into different code paths, there are structures that make your Puppet code flexible and reusable for different circumstances.
The most common control element is the if/else block. It is quite similar to its equivalents in many programming languages:

```puppet
if 'mail_ldap' in $needed_services {
    service { 'dovecot': enable => true }
} else {
    service { 'dovecot': enable => false }
}
```

The Puppet DSL also has a case statement, which is reminiscent of its counterparts in other languages as well:

```puppet
case $role {
    'imap_server': {
        package { 'dovecot': ensure => 'installed' }
        service { 'dovecot': ensure => 'running' }
    }
    /_webserver$/: {
        service { ['apache', 'ssh']: ensure => 'running' }
    }
    default: {
        service { 'ssh': ensure => running }
    }
}
```

A variation of the case statement is the selector. It's an expression, not a statement, and can be used in a fashion similar to the ternary if/else operator found in C-like languages:

```puppet
package { 'dovecot':
    ensure => $role ? {
        'imap_server' => 'installed',
        /desktop$/   => 'purged',
        default      => 'removed',
    },
}
```

It should be used with caution, because in more complex manifests, this syntax will impede readability.
Using variables

Variable assignment works just like in most scripting languages. Any variable name is always prefixed with the $ sign:

```puppet
$download_server = 'img2.example.net'
$url = "https://${download_server}/pkg/example_source.tar.gz"
```

Also, just like most scripting languages, Puppet performs variable value substitution in strings that are in double quotes, but no interpolation at all in single-quoted strings.

Variables are useful for making your manifest more concise and comprehensible. They help you with the overall goal of keeping your source code free from redundancy. An important distinction from variables in imperative programming and scripting languages is the immutability of variables in Puppet manifests. Once a value has been assigned, it cannot be overwritten.

Why is it called a variable at all if it is a constant? One should never look at Puppet as a tool that manages a single system. For a single system, a Puppet variable might look like a constant. But Puppet manages a multitude of systems with different operating systems. Across all these systems, variables will be different and not constants.

Variable types

As of Puppet 3.x, there are only four variable types: Strings, Arrays, Hashes, and Boolean. Puppet 4 introduces a rich type system. The new type system will be explained in Chapter 7, New Features from Puppet 4. The basic variable types work much like their respective counterparts in other languages. Depending on your background, you might be familiar with using associative arrays or dictionaries as semantic equivalents to Puppet's hash type:

```puppet
$a_bool = true
$a_string = 'This is a string value'
$an_array = [ 'This', 'forms', 'an', 'array' ]
$a_hash = {
    'subject'   => 'Hashes',
    'predicate' => 'are written',
    'object'    => 'like this',
    'note'      => 'not actual grammar!',
    'also note' => [ 'nesting is',
                    { 'allowed' => 'of course' } ],
}
```
Accessing the values is equally simple. Note that the hash syntax is similar to that of Ruby, not Perl’s:

```perl
$x = $a_string
$y = $an_array[1]
$z = $a_hash['object']
```

Strings can be used as resource attribute values, but it’s worth noting that a resource title can also be a variable reference:

```perl
package { $apache_package:  
  ensure => 'installed'
}
```

It's intuitively clear what a string value means in this context. But you can also pass arrays here to declare a whole set of resources in one statement. The following manifest manages three packages, making sure that they are all installed:

```perl
packages = ['apache2', 
            'libapache2-mod-php5',
            'libapache2-mod-passenger', ]
package { $packages: 
  ensure => 'installed'
}
```

You will learn how to make efficient use of hash values in later chapters.

The array does not need to be stored in a variable to be used, but it is a good practice in some cases.

### Controlling the order of evaluation

With what you’ve seen this far, you might have got the impression that Puppet’s DSL is a specialized scripting language. That is actually quite far from the truth—a manifest is not a script or program. The language is a tool to model a system state through a set of resources, including files, packages, and cron jobs, among others.

The whole paradigm is different from that of scripting languages. Ruby or Perl are imperative languages that are based around statements that will be evaluated in a strict order. The Puppet DSL is declarative, which means that the manifest declares a set of resources that are expected to have certain properties. These resources are put into a catalog, and Puppet then tries to build a path through all declared resources. The compiler parses the manifests in order, but the configurator applies resources in a very different way.
In other words, the manifests should always describe what you expect to be the end result. The specifics of what actions need to be taken to get there are decided by Puppet.

To make this distinction more clear, let's look at an example:

```yaml
package { 'haproxy':
   ensure => 'installed',
}
file {'/etc/haproxy/haproxy.cfg':
   ensure => file,
   owner  => 'root',
   group  => 'root',
   mode   => '0644',
   source => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
service { 'haproxy':
   ensure => 'running',
}
```

With this manifest, Puppet will make sure that the following state is reached:

1. The **HAproxy** package is installed.
2. The **haproxy.cfg** file has specific content, which has been prepared in a file in `/etc/puppet/modules/`.
3. **HAproxy** is started.

To make this work, it is important that the necessary steps are performed in order.

1. A configuration file cannot usually be installed before the package, because there is not yet a directory to contain it.
2. The service cannot start before installation either. If it becomes active before the configuration is in place, it will use the default settings from the package instead.

This point is being stressed because the preceding manifest does not, in fact, contain cues for Puppet to indicate such a strict ordering. Without explicit dependencies, Puppet is free to put the resources in any order it sees fit.
The recent versions of Puppet allow a form of local manifest-based ordering, so the presented example will actually work as is. The manifest-based ordering can be configured in the `puppet.conf` configuration file as follows:

```
ordering = manifest.
```

This setting is default for Puppet 4. It is still important to be aware of the ordering principles, because the implicit order is difficult to determine in more complex manifests, and as you will learn soon, there are other factors that will influence the order.

### Declaring dependencies

The easiest way to bring order to such a straightforward manifest is resource chaining. The syntax for this is a simple ASCII arrow between two resources:

```
package { 'haproxy':
   ensure => 'installed',
}
->
file { '/etc/haproxy/haproxy.cfg':
   ensure => file,
   owner  => 'root',
   group  => 'root',
   mode   => '0644',
   source => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
->
service {'haproxy':
   ensure => 'running',
}
```

This is only viable if all the related resources can be written next to each other. In other words, if the graphic representation of the dependencies does not form a straight chain, but more of a tree, star, or any other shape, this syntax is not sufficient.

Internally, Puppet will construct an ordered graph of resources and synchronize them during a traversal of that graph.

A more generic and flexible way to declare dependencies is through special **metaparameters**—parameters that are eligible for use with any resource type. There are different metaparameters, most of which have nothing to do with ordering (you have seen `provider` in an earlier example).
For resource ordering, puppet offers the metaparameters, require and before. Both take one or more references to a declared resource as their value. Puppet references have a special syntax, as was previously mentioned:

**Type['title']**

* e.g.

**Package['haproxy']**

Please note that you can only build references to resources which are declared in the catalog. You cannot build and use references to something that is not managed by Puppet, even when it exists on the managed system.

Here is the **HAproxy** manifest, ordered using the **require** metaparameter:

```puppet
package { 'haproxy':
  ensure => 'installed',
}
file {'/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  require => Package['haproxy'],
}
service {'haproxy':
  ensure  => 'running',
  require => File['/etc/haproxy/haproxy.cfg'],
}
```

The following manifest is semantically identical, but relies on the **before** metaparameter rather than **require**:

```puppet
package { 'haproxy':
  ensure => 'installed',
  before => File['/etc/haproxy/haproxy.cfg'],
}
file {'/etc/haproxy/haproxy.cfg':
  ensure => file,
  owner => 'root',
  group => 'root',
  mode => '0644',
  source => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  before => Service['haproxy'],
```
The manifest can also mix both styles of notation, of course. This is left as a reader exercise with no dedicated depiction.

The require metaparameter usually leads to more understandable code, because it expresses the dependency of the annotated resource on another resource. The before parameter, on the other hand, implies a dependency that a referenced resource forms upon the current resource. This can be counter-intuitive, especially for frequent users of packaging systems (which usually implement a require style dependency declaration).

Sometimes, it might be difficult to decide whether to use require or before. In simple cases, most people prefer require. In some cases, it is easier to use before. Think of services that have multiple configuration files. Keeping information about the configuration file and the requirement in a single place reduces errors caused by forgetting to also adopt changes to the service, when adding or removing additional configuration files. Take a look at the following example code:

```
file { '/etc/apache2/apache2.conf'
  ensure => file,
  before => Service[apache2],
}

file { '/etc/apache2/httpd.conf'
  ensure => file,
  before => Service[apache2],
}

service { 'apache2':
  ensure => running,
  enable => true,
}
```

In the example, all dependencies are declared within the file resource declarations. If you use the require parameter instead, you will always need to touch at least two resources in case of changes:

```
file { '/etc/apache2/apache2.conf':
  ensure => file,
}

file { '/etc/apache2/httpd.conf':
  ensure => file,
}
```
Will you remember to update the service resource declaration whenever you add a new file to be managed by Puppet?

Consider another simpler example:

```puppet
if $os_family == 'Debian' {
  file { '/etc/apt/preferences.d/example.net.prefs':
    content => '...
    before => Package['apache2'],
  }
}
package { 'apache2':
  ensure => 'installed',
}
```

The file in the preferences.d directory only makes sense for Debian-like systems. That's why the package cannot safely require it. If the manifest is applied on a different OS, such as CentOS, the `apt preferences` file will not appear in the catalog thanks to the if clause. If the package had it as a requirement regardless, the resulting catalog would be inconsistent, and Puppet would not apply it. Specifying before in the file resource is safe, and semantically equivalent.

The before metaparameter is outright necessary in situations like this one, and can make the manifest code more elegant and straightforward in other scenarios. Familiarity with both before and require is advisable.

**Error propagation**

Defining requirements serves another important purpose. References on declared resources will only be validated as successful references if the depended upon resource was finished successfully. This can be seen like a stop point inside Puppet DSL code, when a required resource is not synchronized successfully.
For example, a file resource will fail if the URL of the source file is broken:

```
file { '/etc/haproxy/haproxy.cfg':
  ensure => file,
  source => 'puppet:///modules/haproxy/etc/haproxy.cfg',
}
```

One path segment is missing here. Puppet will report that the file resource could not be synchronized:

```
root@puppetmaster:~# puppet apply typo.pp
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.62 seconds
Error: /Stage[main]/Main/File[/etc/haproxy/haproxy.cfg]: Could not evaluate: Could not retrieve information from environment production source(s) puppet:///modules/haproxy/etc/haproxy.cfg
Notice: /Stage[main]/Main/Service[haproxy]: Dependency File[/etc/haproxy/haproxy.cfg] has failures: true
Warning: /Stage[main]/Main/Service[haproxy]: Skipping because of failed dependencies
Notice: Applied catalog in 0.06 seconds
```

In this example, the Error line describes the error caused by the broken URL. The error propagation is represented by the Notice and Warning lines below it.

Puppet failed to apply changes to the configuration file; it cannot compare the current state to the nonexistent source. As the service depends on the configuration file, Puppet will not even try to start it. This is for safety: if any dependencies cannot be put into the defined state, Puppet must assume that the system is not fit for application of the dependent resource.

This is another important reason to make consequent use of resource dependencies. Remember that both the chaining arrow and the before metaparameter imply error propagation as well.

**Avoiding circular dependencies**

Before you learn about another way in which resources can interrelate, there is an issue that you should be aware of: dependencies must not form circles. Let's visualize this in an example:

```
file { '/etc/haproxy':
  ensure => 'directory',
  owner => 'root',
}
```
The dependency circle in this manifest is somewhat hidden (as will likely be the case for many such circles that you will encounter during regular use of Puppet). It is formed by the following relations:

- The `File['/etc/haproxy/haproxy.cfg']` auto-requires the parent directory, `File['/etc/haproxy']`. This is an implicit, built-in dependency.
- The parent directory, `File['/etc/haproxy']`, requires `Service['haproxy']` due to its `before` metaparameter.
- The `Service['haproxy']` service requires the `File['/etc/haproxy/haproxy.cfg']` config.

Implicit dependencies exist for the following resource combinations, among others:

- If a directory and a file inside the directory is declared, Puppet will first create the directory and then the file.
- If a user and his primary group is declared, Puppet will first create the group and then the user.
- If a file and the owner (user) is declared, Puppet will first create the user and then the file.

Granted, the preceding example is contrived—it will not make sense to manage the service before the configuration directory. Nevertheless, even a manifest design that is apparently sound can result in circular dependencies. This is how Puppet will react to that:

```
rroot@puppetmaster:~# puppet apply circle.pp
Notice: Compiled catalog for puppetmaster.example.net in environment production in 0.62 seconds
```
Writing Your First Manifests

Error: Failed to apply catalog: Found 1 dependency cycle:
(File[/etc/haproxy/haproxy.cfg] => Service[haproxy] => File[/etc/haproxy]
=> File[/etc/haproxy/haproxy.cfg])

Try the '--graph' option and opening the resulting '.dot' file in
OmniGraffle or GraphViz

The output helps you locate the offending relation(s). For very wide dependency
circles with lots of involved resources, the textual rendering is difficult to analyze.
Therefore, Puppet also gives you the opportunity to get a graphical representation
of the dependency graph through the --graph option.

If you do this, Puppet will include the full path to the newly created .dot file in its
output. Its content looks similar to Puppet's output:

digraph Resource_Cycles {
  label = "Resource Cycles"
  "File[/etc/haproxy/haproxy.cfg]" -> "Service[haproxy]" -> "File[/etc/
  haproxy]" -> "File[/etc/haproxy/haproxy.cfg]"
}

This is not helpful by itself, but it can be fed directly into tools such as dot to
produce an actual diagram.

To summarize, resource dependencies are helpful in keeping Puppet from acting
upon resources in unexpected or uncontrolled situations. They are also useful in
restricting the order of resource evaluation.

Implementing resource interaction

In addition to dependencies, resources can also enter a similar yet different mutual
relation. Remember the pieces of output that we skipped earlier. They are as follows:

root@puppetmaster:--# puppet apply puppet_service.pp --noop
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.62 seconds
Notice: /Stage[main]/Main/Service[puppet]/ensure: current_value running,
should be stopped (noop)
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Applied catalog in 0.05 seconds
Puppet mentions that refreshes would have been triggered for the reason of an event. Such events are emitted by resources whenever Puppet acts on the need for a sync action. Without explicit code to receive and react to events, they just get discarded.

The mechanism to set up such event receivers is named in an analogy of a generic publish/subscribe queue—resources get configured to react to events using the subscribe metaparameter. There is no publish keyword or parameter, since each and every resource is technically a publisher of events (messages). Instead, the counterpart of the subscribe metaparameter is called notify, and it explicitly directs generated events at referenced resources.

One of the most common practical uses of the event system is to reload service configurations. When a service resource consumes an event (usually from a change in a config file), Puppet invokes the appropriate action to make the service restart.

If you instruct Puppet to do this, it can result in brief service interruptions due to this restart operation. Note that if the new configuration causes an error, the service might fail to start and stay offline.

The following code example shows the relationships between the haproxy package, the corresponding haproxy configuration file, and the haproxy service:

```yaml
file { '/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  require => Package['haproxy'],
}

service { 'haproxy':
  ensure    => 'running',
  subscribe => File['/etc/haproxy/haproxy.cfg'],
}
```

If the notify metaparameter is to be used instead, it must be specified for the resource that emits the event:

```yaml
file { '/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
}
```
source => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
require => Package['haproxy'],
notify  => Service['haproxy'],
}

service { 'haproxy':
  ensure  => 'running',
}

This will likely feel reminiscent of the before and require metaparameters, which offer symmetric ways of expressing an interrelation of a pair of resources just as well. This is not a coincidence—these metaparameters are closely related to each other:

- The resource that subscribes to another resource implicitly requires it
- The resource that notifies another is implicitly placed before the latter one in the dependency graph

In other words, subscribe is the same as require, except for the dependent resource receiving events from its peer. The same holds true for notify and before.

The chaining syntax is also available for signaling. To establish a signaling relation between neighboring resources, use an ASCII arrow with a tilde, ~>, instead of the dash in ->

    file { '/etc/haproxy/haproxy.cfg': ... }
    ->
    service { 'haproxy': ... }

The service resource type is one of the two notable types that support refreshing when resources get notified (the other will be discussed in the next section). There are others, but they are not as ubiquitous.

Examining the most notable resource types

To complete our tour of the basic elements of a manifest, let's take a closer look at the resource types that you have already used, and some of the more important ones that you have not yet encountered.

You probably already have a good feeling for the file type, which will ensure the existence of files and directories, along with their permissions. Pulling a file from a repository (usually, a Puppet module) is also a frequent use case, using the source parameter.
For very short files, it is more economic to include the desired content right in the manifest:

```puppet
file { '/etc/modules':
  ensure => file,
  content => '# Managed by Puppet:
drbd
',}
```

The double quotes allow expansion of escape sequences such as 

Another useful capability is managing symbolic links:

```puppet
file { '/etc/apache2/sites-enabled/001-puppet-lore.org':
  ensure => 'link',
  target => '../sites-available/puppet-lore.org',}
```

The next type that you already know is package, and its typical usage is quite intuitive. Make sure that packages are either installed or removed. A notable use case that you have not yet seen is to use the basic package manager instead of `apt` or `yum/zypper`. This is useful if the package is not available from a repository:

```puppet
package { 'haproxy':
  ensure => present,
  provider => 'dpkg',
  source => '/opt/packages/haproxy-1.5.1_amd64.dpkg',}
```

Your mileage usually increases if you make the effort of setting up a simple repository instead, so that the main package manager can be used after all.

Last but not least, there is a service type, the most important attributes of which you already know. It’s worth pointing out that it can serve as a simple shortcut in cases where you don’t wish to add a full-fledged init script or something similar. With enough information, the base provider for the service type will manage simple background processes for you:

```puppet
service { 'count-logins':
  provider => 'base',
  ensure => 'running',
  binary => '/usr/local/bin/cnt-logins',
  start => '/usr/local/bin/cnt-logins --daemonize',
```
Puppet will not only restart the script if it is not running for some reason, but will also restart it whenever the content of the referenced configuration file changes. This only works if Puppet manages the file content and all changes propagate through Puppet only.

If Puppet changes any other property of the script file (for example, the file mode), that too will lead to a restart of the process.

Let's take a look at some other types you will probably need.

**The user and group types**

Especially in the absence of central registries such as LDAP, it is useful to be able to manage user accounts on each of your machines. There are providers for all supported platforms, however, the available attributes vary. On Linux, the `useradd` provider is the most common. It allows the management of all fields in `/etc/passwd`, such as `uid` and `shell`, and also group memberships:

```puppet
user { 'john':
    ensure => present,
    uid    => 2014,
    home   => '/home/john',
    managehome => true, # <- adds -m to useradd
    gid    => 1000,
    shell  => '/bin/zsh',
    groups => [ 'proxy-admins' ],
}
```

As with all resources, Puppet will not only make sure that the user and group exist, but also fix any divergent properties, such as the `home` directory.

Even though the user depends on the group, (because they cannot be added before the group exists) it need not be expressed in the manifest. The user automatically requires all necessary groups, similar to a file auto requiring its parent directory.
Note that Puppet will also happily manage your LDAP user accounts.

It was mentioned earlier that there are different attributes available, depending on the Operating System. Linux (and the `useradd` provider) support setting a password, whereas on HP-UX (using the `hp-ux` provider) the user password cannot be set via Puppet.

In this case, Puppet will only show a warning saying that the user resource type is making use of an unsupported attribute, and will continue managing all other attributes. In other words, using an unsupported attribute in your Puppet DSL code will not break your Puppet run.

### The exec resource type

There is one oddball resource type in the Puppet core. Remember our earlier assertion that Puppet is not a specialized scripting engine, but instead, a tool that allows you to model part of your system state in a compelling DSL, and which is capable of altering your system to meet the defined goal. This is why you declare `user` and `group`, instead of invoking `groupadd` and `useradd` in order. You can do this because Puppet comes with support to manage such entities. This is vastly beneficial because Puppet also knows that on different platforms, other commands are used for account management, and that the arguments can be subtly different on some systems.

Of course, Puppet does not have knowledge of all the conceivable particulars of any supported system. Say that you wish to manage an OpenAFS file server. There are no specific resource types to aid you with this. The ideal solution is to exploit Puppet’s plugin system and to write your own types and providers so that your manifests can just reflect the AFS-specific configuration. This is not simple though, and also not worthwhile in cases where you only need Puppet to invoke some exotic commands from very few places in your manifest.

For such cases, Puppet ships with the `exec` resource type, which allows the execution of custom commands in lieu of an abstract sync action.

For example, it can be used to unpack a tarball in the absence of a proper package:

```puppet
exec { 'tar cjf /opt/packages/homebrewn-3.2.tar.bz2':
  cwd     => '/opt',
  path    => '/bin:/usr/bin',
  creates => '/opt/homebrewn-3.2',
}
```
Writing Your First Manifests

The `creates` parameter is important for Puppet to tell whether the command needs running—once the specified path exists, the resource counts as synchronized. For commands that do not create a telltale file or directory, there are the alternative parameters, `onlyif` and `unless`, to allow Puppet to query the sync state:

```puppet
exec { 'perl -MCPAN -e "install YAML"':
  path   => '/bin:/usr/bin',
  unless => 'cpan -l | grep -qP ^YAML\b',
}
```

The query command’s exit code determines the state. In the case of `unless`, the `exec` command runs if the query fails. This is how the `exec` type maintains idempotency. Puppet does this automatically for most resource types, but this is not possible for `exec`, because synchronization is defined so arbitrarily. It becomes your responsibility as the user to define the appropriate queries per resource.

Finally, the `exec` type resources are the second notable case of receivers for events using `notify` and `subscribe`:

```puppet
exec { 'apt-get update':
  path        => '/bin:/usr/bin',
  subscribe   => File['/etc/apt/sources.list.d/jenkins.list'],
  refreshonly => true,
}
```

You can even chain multiple `exec` resources in this fashion so that each invocation triggers the next one. However, this is a bad practice and degrades Puppet to a (rather flawed) scripting engine. The `exec` resources should be avoided in favor of regular resources whenever possible. Some resource types that are not part of the core are available as plugins from the Puppet Forge. You will learn more about this topic in Chapter 5, `Extending Your Puppet Infrastructure with Modules`.

Since `exec` resources can be used to perform virtually any operation, they are sometimes abused to stand in for more proper resource types. This is a typical antipattern in Puppet manifests. It is safer to regard `exec` resources as the last resort that is only to be used if all other alternatives have been exhausted.
All Puppet installations have the type documentation built into the code, which is printable on command line by using the puppet describe command:

```
puppet describe <type> [-s]
```

In case you are unsure whether a type exists, you can tell Puppet describe to return a full list of all available resource types:

```
puppet describe --list
```

Let's briefly discuss two more types that are supported out of the box. They allow the management of cron jobs, mounted partitions, and shares respectively, which are all frequent requirements in server operation.

### The cron resource type

A cron job mainly consists of a command and the recurring time and date at which to run the command. Puppet models the command and each date particle as a property of a resource with the `cron` type:

```
cron { 'clean-files':
    ensure      => present,
    user        => 'root',
    command     => '/usr/local/bin/clean-files',
    minute      => '1',
    hour        => '3',
    weekday     => [ '2', '6' ],
    environment => 'MAILTO=felix@example.net',
}
```

The `environment` property allows you to specify one or more variable bindings for cron to add to the job.

### The mount resource type

Finally, Puppet will manage all aspects of mountable filesystems for you—including their basic attributes such as the source device and mount point, the mount options, and the current state. A line from the `fstab` file translates quite literally to a Puppet manifest:

```
mount { '/media/gluster-data':
    ensure => 'mounted',
    device => 'gluster01:/data',
    fstype => 'glusterfs',
}
```
Writing Your First Manifests

```perl
(options => 'defaults,_netdev',
    dump => 0,
    pass => 0,
}
```

For this resource, Puppet will make sure that the filesystem is indeed mounted after the run. Ensuring the unmounted state is also possible, of course; Puppet can also just make sure the entry is present in the `/etc/fstab` file, or absent from the system altogether.

### Summary

After installing Puppet on your system, you can use it by writing and applying manifests. These manifests are written in Puppet’s DSL and contain descriptions of the desired state of your system. Even though they resemble scripts, they should not be considered as such. For one thing, they consist of resources instead of commands. These resources are generally not evaluated in the order in which they have been written. An explicit ordering should be defined through the `require` and `before` metaparameters instead.

Each resource has a number of attributes: parameters and properties. Each property is evaluated in its own right; Puppet detects whether a change to the system is necessary to get any property into the state that is defined in the manifest. It will also perform such changes. This is referred to as **synchronizing** a resource or property.

The ordering parameters, `require` and `before`, are of further importance because they establish dependency of one resource on one or more others. This allows Puppet to skip parts of the catalog if an important resource cannot be synchronized. Circular dependencies must be avoided.

Each resource in the manifest has a resource type that describes the nature of the system entity that is being managed. Some of the types that are used most frequently are file, package, and service. Puppet comes with many types for convenient system management, and many plugins are available to add even more. Some tasks require the use of `exec` resources, but this should be done sparingly.

In the next chapter, we will introduce the master/agent setup.
Where to buy this book

You can buy Puppet 4 Essentials Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.