Puppet Essentials

Felix Frank

Chapter No. 1
"Writing Your First Manifests"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Writing Your First Manifests"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Felix Frank** has used and programmed computers for most of his life. During and after his Computer Science diploma, he gained on-the-job experience as a system administrator, server operator, and open source software developer. Of his 10-year career, he spent 5 years as a Puppet power user. For almost a year, he intensified his learning by contributing source code and actively participating in several conferences. This is his first foray into writing books and is a work of great effort and sacrifice.

For More Information:

www.packtpub.com/networking-and-servers/puppet-essentials
Acknowledgments

I'd like to thank my editors, Sam Wood, Sweny M. Sukumaran, and Mrunal M. Chavan, for their ongoing support and for making this title possible in the first place. This book would not be what it is without the amazing feedback from my reviewers, Ger Apeldoorn, Brian Moore, Josh Partlow, and Thomas Dao.

I'd also like to thank a number of Puppet Labs employees for their kind feedback and ongoing support—Andrew Parker, Adrien Thebo, Dawn Foster, Joshua Partlow, Josh Cooper, Henrik Lindberg, Charlie Sharpsteen, Kylo Ginsberg, Ethan Brown, Rob Reynolds, Jeff McCune, and Eric Sorenson. Special thanks to Luke Kanies for creating Puppet and dedicating so many resources to this amazing community.

Further thanks to Waltraut Niepraschkh and the entire data center staff at DESY Zeuthen for getting me on track with the central management of Unix servers. Also, thanks to MPeXnetworks for giving me the opportunity to learn more about Puppet and ultimately helping me write this very book.

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
Puppet Essentials

The software industry is changing and so are its 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 evermore workflows. Developers set up and maintain their own environments, and operations raise automation to new levels and translate whole infrastructures to code.

A steady stream of new technologies allows for more efficient organizational principles. One of these newcomers is Puppet. Its fresh take on server configuration management caused 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 the software industry. 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 might require the navigation of some intricacies that stem from Puppet's unique modeling approach.

This book aims at teaching you all that is required to tap not only the basics of Puppet, but also the very ideas and principles of Puppet-based designs. Sophisticated tooling is presented in order to enable efficient and productive use. You are introduced to and familiarized with a range of Puppet-centric technologies.

What This Book Covers

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

Chapter 2, The Master and Its Agents, provides you with a quick how-to guide in order to set up all the components required for a distributed Puppet infrastructure. It will teach you how to create the master, an agent, and set up Passenger with Apache or Nginx.

Chapter 3, A Peek Under the Hood – Facts, Types, and Providers, gives you a summary of the core components that give Puppet its flexibility as well as its ease of use and extension. It will help you understand the principles of Puppet's function.

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
Chapter 4, *Modularizing Manifests with Classes and Defined Types*, teaches you the most important structural elements of Puppet manifests. You will learn how to use them for best results.

Chapter 5, *Extending Your Puppet Infrastructure with Modules*, gives you a tour of downloadable extensions along with providing you with a guide to create your own. You will learn how to extend Puppet for your specific needs.

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

Chapter 7, *Separating Data from Code Using Hiera*, provides you with an introduction to the powerful configuration data storage that comes with Puppet. You will learn how to naturally model your infrastructure in a hierarchy.

Chapter 8, *Configuring Your Cloud Application with Puppet*, gives you a superposition of the skills you have acquired. It will help you gain some specialized insights into how to take centralized control of your cloud through Puppet.

For More Information: 
Over the last few years, configuration management has become increasingly significant to the IT world. Server operations in particular are 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

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
Writing Your First Manifests

Getting started
Installing Puppet is easy. On large Linux distributions, you can just install the Puppet package via `apt-get` or `yum`.

Puppet moves a lot faster than most distributions. To be more up to date, you can install current packages directly from the Puppet Labs repositories. You can visit https://docs.puppetlabs.com/guides/install_puppet/pre_install.html for more details.

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. (I avoided the term "execute" on purpose—manifests cannot be executed. More details will follow around the middle of this chapter.)

```
root@puppetagent# puppet apply hello_world.pp
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.05 seconds
Notice: Hello, world!
Notice: /Stage[main]/Main/Notify[Hello, world!]/message: defined 'message' as 'Hello, world!'
Notice: Finished catalog run in 0.07 seconds
```

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
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. Assume 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_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 screenshot:

```
root@puppetagent:~ # puppet apply puppet_service.pp
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.14 seconds
Notice: /Stage[main]/Main/Service[puppet]/enable: enable changed 'true' to 'false'
Notice: Finished catalog run in 0.42 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@puppetagent:~ # puppet apply puppet_service.pp
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.15 seconds
Notice: Finished catalog run in 0.27 seconds
```

You will often get this output 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`.

If you are following along, you might get a different output depending on your version of Puppet. All examples in this book use the 3.6.2 release from Puppet Labs' repository.

For More Information:

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, 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 I 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:

```plaintext
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.

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, it will perform the following tasks:

- Test whether the resource is already in sync with the target state
- If the resource is not in sync, trigger a sync action

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
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.

**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 look at the informational messages though. Apply the service.pp manifest once more:

```
rroot@puppetagent# puppet apply puppet.service.pp
  Notice: Compiled catalog for puppetagent.example.net in environment production in 0.15 seconds
  Notice: Finished catalog run in 0.27 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:

```
rroot@puppetagent# puppet apply -e 'service { "puppet": enable => true }'
  Notice: Compiled catalog for puppetagent.example.net in environment production in 0.14 seconds
  Notice: /Stage[main]/Main/Service[puppet]/enable: enable changed 'false' to 'true'
  Notice: Finished catalog run in 0.43 seconds
```

I 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, if at least for convenience.

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
You instructed Puppet to perform yet another change upon the Puppet service. The output reflects the exact change that is 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 log output that indicates what will change without the switch. This is useful to determine whether a manifest would possibly break anything on your system:

```
root@puppetagent# puppet apply puppet_service.pp --noop
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.14 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: Finished catalog run in 0.29 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.

---

**For More Information:**

The additional notices about triggered refreshes are usually not important and can be ignored. 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 along with 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 equivalent 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 counterpart 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 ? {
```
In more complex manifests, this syntax will impede readability. Puppet Labs recommend to use it only in variable assignments.

Using variables

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

```bash
$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 to make 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.

Under specific circumstances, it is possible to amend values through concatenation. You might encounter statements such as for $variable += 'value'. This should be used with care, or avoided altogether.

Variable types

As of Puppet 3.x, there are only three variable types: strings, arrays, and hashes. Puppet 4 introduces a rich type system, but this is out of the scope of this book. The three variable types work much like their respective counterparts in other languages. Depending on your background, you might be familiar with associative arrays or dictionaries as semantic equivalents to Puppet’s hash type:

```bash
$a_string = 'This is a string value'
$an_array = [ 'This', 'forms', 'an', 'array' ]
$a_hash   = {
    'subject' => 'Hashes',
}```

For More Information:

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 obviously 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 this way, but it is a good practice in some cases.
Writing Your First Manifests

Controlling the order of evaluation

With what you've seen thus far, you might have gotten 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 software resources, including files, packages, and cron jobs, among others.

The whole paradigm is different from that of scripting languages. Whereas Ruby or Perl are imperative languages, which are based around statements that will be evaluated in a strict order, the Puppet DSL is declarative: the manifest declares a set of resources that are expected to have certain properties.

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:

```perl
package {  
    'haproxy': ensure => 'installed',
}
file {  
    '/etc/haproxy/haproxy.cfg':  
        owner => 'root',  
        group => 'root',  
        mode => '644',  
        source =>  
            'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
service {  
    'haproxy': ensure => 'running',
}
```

The manifest has Puppet make sure that:

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

To make this work, it is important that the necessary steps are performed in order. A configuration file cannot usually be installed before the package, because there is not yet a directory to contain it. 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.

For More Information:

www.packtpub.com/networking-and-servers/puppet-essentials
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. 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 that is a simple ASCII arrow between two resources:

```plaintext
package {
  'haproxy': ensure => 'installed',
}
->
file {
  '/etc/haproxy/haproxy.cfg':
    owner => 'root',
    group => 'root',
    mode  => '644',
    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 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.

---

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
A more generic and flexible way to declare dependencies are specialized **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), but for now, let's concentrate on `require` and `before`. Here is the **HAproxy** manifest, ordered using the `require` metaparameter:

```puppet
package {
    'haproxy': ensure => 'installed',
}
file {
    '/etc/haproxy/haproxy.cfg':
        owner   => 'root',
        group   => 'root',
        mode    => '644',
        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':
        owner   => 'root',
        group   => 'root',
        mode    => '644',
        source  =>
            'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
        before => Service['haproxy'],
}
service {
    'haproxy': ensure => 'running',
}
```

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
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 a 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).

The `before` metaparameter is still outright necessary in certain situations and can make the manifest code more elegant and straightforward for others. Familiarity with both variants is advisable.

Let's see an example of dependencies that do not form a straight chain. In the following code, Puppet manages the configuration directory explicitly, so the config file can be deployed independently of the package. The service's requirements are passed in an array:

```puppet
package {
    'haproxy': ensure => 'installed',
}

file {
    '/etc/haproxy':
        ensure => 'directory',
        owner => 'root',
        group => 'root',
        mode => '644';
    '/etc/haproxy/haproxy.cfg':
        owner => 'root',
        group => 'root',
        mode => '644',
        source =>
            'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}

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

Puppet will automatically make the config file require the containing directory if it is part of your manifest. There is no need to add the metaparameter explicitly. This is a special function of file resources.

The manifest saves lines by declaring both file resources in one block, separated by a semicolon.

**Error propagation**

Defining requirements serves another important purpose. I have used the term dependency in this context before. This wording was deliberate—aside from defining a mandatory order of evaluation, the require and before parameters bond the involved resources into a unidirectional failure pair. For example, a file resource will fail if the URL of the source file is broken:

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

Puppet will report that the resource could not be synchronized:

```
root@agent# puppet apply typo.pp
Notice: Compiled catalog for agent.example.net in environment production in 0.25 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: Finished catalog run in 0.13 seconds
```

In this screenshot, the Error line describes the error caused by the broken URL. The error propagation is represented by the Notice and Warning keywords below that.

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.

---

For More Information:  
www.packtpub.com/networking-and-servers/puppet-essentials
This is another important reason to make consequent use of resource dependencies. Remember that the chaining arrow and the before metaparameter imply the same semantics.

**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:

```ruby
file {
  '/etc/haproxy':
    ensure => 'directory',
    owner => 'root',
    group => 'root',
    mode => '644';
  '/etc/haproxy/haproxy.cfg':
    owner => 'root',
    group => 'root',
    mode => '644',
    source =>
      'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}

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

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'] autorequires the parent directory, File['/etc/haproxy']
- The parent directory, File['/etc/haproxy'], requires Service['haproxy'], due to the latter's before metaparameter
- The Service['haproxy'] requires the File['/etc/haproxy/haproxy.cfg'] config

For More Information:  
www.packtpub.com/networking-and-servers/puppet-essentials
Granted, the 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:

```bash
root@puppetagent1# puppet apply circle.pp
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.25 seconds
Error: Could not apply complete catalog: Found 1 dependency cycle:
(File[/etc/haproxy/haproxy.cfg] => Service[haproxy] => File[/etc/haproxy/haproxy] => File[/etc/haproxy/haproxy.cfg])
Try the '--graph' option and opening the resulting '.dot' file in OmniGraffle or GraphViz
Notice: Finished catalog run in 0.05 seconds
```

The output helps you to 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:

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

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

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

**Implementing resource interaction**

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

For More Information:

www.packtpub.com/networking-and-servers/puppet-essentials
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 analogy to 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 the ability 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.

```bash
root@puppetagent# puppet apply puppet_service.pp --noop
Notice: Compiled catalog for puppetagent.example.net in environment production in 0.14 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: Finished catalog run in 0.23 seconds
```

file {

'/etc/haproxy/haproxy.cfg':
  owner => 'root',
  group => 'root',
  mode  => '644',
  source =>
    'puppet:///modules/etc/haproxy/haproxy.cfg',
  require => Package['haproxy'],
}

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

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
If the `notify` metaparameter is to be used instead, it must be specified for the resource that emits the event:

```puppet
file {      
    '/etc/haproxy/haproxy.cfg':
          owner => 'root',
          group => 'root',
          mode  => '644',
          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 `subscribe` 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 `->`:

```puppet
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 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':
        content => '# Managed by Puppet!

drbd
',
}
```

The double quotes allow expansion of escape sequences such as `\n`.

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';
    '/etc/apache2/sites-enabled/002-wordpress':
        ensure => '../sites-available/wordpress',
}
```

Using the link target as the `ensure` value is possible but not recommended.

The next type 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 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':
        provider => 'dpkg',
        source   => '/opt/packages/haproxy-1.5.1_amd64.dpkg',
}
```

For More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
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 the service type, the most important attributes of which you already know as well. 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 similar. With enough information, the base provider for the service type will manage simple background processes for you:

```
service {
  'count-logins':
    provider => 'base',
    ensure   => 'running',
    binary   => '/usr/local/bin/cnt-logins',
    start    => '/usr/local/bin/cnt-logins --daemonize',
    subscribe => File['/usr/local/bin/cnt-logins'],
}
```

Puppet will not only restart the script if it is not running for some reason, but also will restart it whenever the content 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 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, but also group memberships:

```
group {
  'proxy-admins':
    ensure => present,
    gid    => 4002,
}
```

For More Information:

www.packtpub.com/networking-and-servers/puppet-essentials
user {
    'john':
        uid        => 2014,
        home       => '/home/john'
        managehome => true,
        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 it cannot be added before the group exists, this need not be expressed in the manifest. The user automatically requires all necessary groups, similar to file autorequiring its parent directory.

Note that Puppet will also happily manage your LDAP user accounts.

The exec resource type

There is one oddball resource type in the Puppet core. Remember my 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 is capable of altering your system to meet the defined goal. This is why you declare a user and a group, for example, 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 conceivable particulars of any supported system. Say 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 More Information:
www.packtpub.com/networking-and-servers/puppet-essentials
Writing Your First Manifests

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:

```plaintext
evac {
    'tar cjm /opt/packages/homebrewn-3.2.tar.bz2':
    cwd => '/opt',
    path => '/bin:/usr/bin',
    creates => '/opt/homebrewn-3.2',
}
```

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

```plaintext
evac {
    '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.

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

```plaintext
evac {
    '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, which are only to be used if all other alternatives have been exhausted.
Let's briefly discuss two more types that are supported out of the box. They allow the management of cron jobs and mounted partitions and shares, respectively, which are 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:

```puppet
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 about mountable filesystems for you—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:

```puppet
mount {
    '/media/gluster-data':
        ensure   => 'mounted',
        device   => 'gluster01:/data',
        fstype   => 'glusterfs',
        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, but Puppet can also just make sure the entry is present in the \texttt{fstab} file, or absent from the system altogether.

\section*{Summary}

After installing Puppet on your system, you can use it by writing and applying manifests. These are programs that are written in Puppet's DSL. 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 \texttt{require} and \texttt{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 \texttt{synchronizing} a resource or property.

The ordering parameters, require and before, are of further importance because they establish dependency of one resource upon one or more others. This allows Puppet to skip parts of the manifest 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 \texttt{exec} resources, but this should be done sparingly.

In the next chapter, we will have a look at the master/agent setup.

\footnotesize{For More Information: \url{www.packtpub.com/networking-and-servers/puppet-essentials}}
Where to buy this book

You can buy Puppet Essentials from the Packt Publishing website:
https://www.packtpub.com/networking-and-servers/puppet-essentials

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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