Resources

Open Source Puppet — 6.17 (latest)

Sections

Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the actual state matches the desired state.

Resources contained in classes and defined types share the relationships of those classes and defined types. Resources are not subject to scope: a resource in any area of code can be referenced from any other area of code.

A *resource declaration* adds a resource to the catalog and tells Puppet to manage that resource's state. When Puppet applies the compiled catalog, it:

1. Reads the actual state of the resource on the target system.
2. Compares the actual state to the desired state.
3. If necessary, changes the system to enforce the desired state.
4. Logs any changes made to the resource. These changes appear in Puppet agent's log and in the run report, which is sent to the master and forwarded to any specified report processors.

If the catalog doesn't contain a particular resource, Puppet does nothing with whatever that resource described. If you remove a package resource from your manifests, Puppet doesn't uninstall the package; instead, it just ignores it. To remove a package, manage it as a resource and set `ensure => absent`.

You can delay adding resources to the catalog. For example, classes and defined types can contain groups of resources. These resources are managed only if you add that class or defined resource to the catalog. Virtual resources are added to the catalog only after they are realized.

+ Example configurations
+ References

+ Expand all

Resource declarations

At minimum, every resource declaration has a *resource type*, a *title*, and a set of *attributes*:

```
<TYPE> { '<TITLE>': <ATTRIBUTE> => <VALUE>, }
```

The resource title and attributes are called the resource body. A resource declaration can have one resource body or multiple resource bodies of the same resource type.

Resource declarations are expressions in the Puppet language — they always have a side effect of adding a resource to the catalog, but they also resolve to a value. The value of a resource declaration is an array of *resource references*, with one reference for each resource the expression describes.

A resource declaration has extremely low precedence; in fact, it's even lower than the variable assignment operator (=). This means that if you use a resource declaration for its value, you must surround it with parentheses to associate it with the expression that uses the value.

If a resource declaration includes more than one resource body, it declares multiple resources of that resource type. The resource body is a title and a set of attributes; each body must be separated from the next one with a semicolon. Each resource in a declaration is almost completely independent of the others, and they can have completely different values for their attributes. The only connections between resources that share an expression are:

- They all have the same resource type.
- They can all draw from the same pool of default values, if a resource body with the title `default` is present.

Resource uniqueness

Each resource must be unique; Puppet does not allow you to declare the same resource twice. This is to prevent multiple conflicting values from being declared for the same attribute. Puppet uses the resource `title` and the `name` attribute or *namevar* to identify duplicate resources — if either the title or the name is duplicated within a given resource type, catalog compilation fails. See the page about **resource syntax** for details about resource titles and namevars. To provide the same resource for multiple classes, use a class or a virtual resource to add it to the catalog in multiple places without duplicating it. See **classes** and **virtual resources** for more information.

Relationships and ordering

By default, Puppet applies unrelated resources in the order in which they're written in the manifest. If a resource must be applied before or after some other resource, declare a relationship between them to show that their order isn't coincidental. You can also make changes in one resource cause a refresh of some other resource. See the Relationships and ordering page for more information.

Otherwise, you can customize the default order in which Puppet applies resources with the ordering setting. See the configuration page for details about this setting.

Resource types

Every resource is associated with a *resource type*, which determines the kind of configuration it manages. Puppet has built-in resource types such as `file`, `service`, and `package`. See the resource type reference for a complete list and information about the built-in resource types.

You can also add new resource types to Puppet:

- Defined types are lightweight resource types written in the Puppet language.
- Custom resource types are written in Ruby and have the same capabilities as Puppet's built-in types.

Title

A resource's title is a string that uniquely identifies the resource to Puppet. In a resource declaration, the title is the identifier after the first curly brace and before the colon. For example, in this file resource declaration, the title is `/etc/passwd`:

```
file  { '/etc/passwd':
  owner => 'root',
  group => 'root',
}
```

Titles must be unique per resource type. You can have both a package and a service titled "ntp," but you can only have one service titled "ntp." Duplicate titles cause compilation to fail.

The title of a resource differs from the *namevar* of the resource. Whereas the title identifies the resource to Puppet itself, the namevar identifies the resource to the target system and is usually specified by the resource's `name` attribute. The resource title doesn't have to match the namevar, but you'll often want it to: the value of the namevar attribute defaults to the title, so using the name in the title can save you some typing.

If a resource type has multiple namevars, the type specifies whether and how the title maps to those namevars. For example, the `package` type uses the `provider` attribute to help determine uniqueness, but that attribute has no special relationship with the title. See each type's documentation for details about how it maps title to namevars.

Attributes

Attributes describe the desired state of the resource; each attribute handles some aspect of the resource. For example, the `file` type has a `mode` attribute that specifies the permissions for the file.

Each resource type has its own set of available attributes; see the **resource type reference** for a complete list. Most resource types have a handful of crucial attributes and a larger number of optional ones. Attributes accept certain data types, such as strings, numbers, hashes, or arrays. Each attribute that you declare must have a value. Most attributes are optional, which means they have a default value, so you do not have to assign a value. If an attribute has no default, it is considered required, and you must assign it a value.

Most resource types contain an `ensure` attribute. This attribute generally manages the most basic state of the resource on the target system, such as whether a file exists, whether a service is running or stopped, or whether a package is installed or uninstalled. The values accepted for the `ensure` attribute vary by resource type. Most accept `present` and `absent`, but there are variations. Check the reference for each resource type you are working with.

> **Tip:** Resource and type attributes are sometimes referred to as parameters. Puppet also has properties, which are slightly different from parameters: properties correspond to something measurable on the target system, whereas parameters change how Puppet manages a resource. A property always represents a concrete state on the target system. When talking about resource declarations in Puppet, parameter is a synonym for attribute.

Namevars and `name`

Every resource on a target system must have a unique identity; you cannot have two services, for example, with the same name. This identifying attribute in Puppet is known as the *namevar*.

Each resource type has an attribute that is designated to serve as the namevar. For most resource types, this is the `name` attribute, but some types use other attributes, such as the `file` type, which uses `path`, the file's location on disk, for its namevar. If a type's namevar is an attribute other than `name`, this is listed in the type reference documentation.

Most types have only one namevar. With a single namevar, the value must be unique per resource type. There are a few rare exceptions to this rule, such as the `exec` type, where the namevar is a command. However, some resource types, such as `package`, have multiple namevar attributes that create a composite namevar. For example, both the `yum` provider and the `gem` provider have `mysql` packages, so both the `name` and the `provider` attributes are namevars, and Puppet uses both to identify the resource.

The namevar differs from the resource's *title*, which identifies a resource to Puppet's compiler rather than to the target system. In practice, however, a resource's namevar and the title are often the same, because the namevar usually defaults to the title. If you don't specify a value for a resource's namevar when you declare the resource, Puppet uses the resource's title.

You might want to specify different a namevar that is different from the title when you want a consistently titled resource to manage something that has different names on different platforms. For example, the NTP service might be `ntpd` on Red Hat systems, but `ntp` on Debian and Ubuntu. You might title the service "ntp," but set its namevar --- the `name` attribute --- according to the operating system. Other resources can then form relationships to the resource without the title changing.

Metaparameters

Some attributes in Puppet can be used with every resource type. These are called *metaparameters*. These don't map directly to system state. Instead, metaparameters affect Puppet's behavior, usually specifying the way in which resources relate to each other.

The most commonly used metaparameters are for specifying order relationships between resources. See the documentation on **relationships and ordering** for details about those metaparameters. See the full list of available metaparameters in the **metaparameter reference**.

Resource syntax

You can accomplish a lot with just a few resource declaration features, or you can create more complex declarations that do more.

Basic syntax

The simplified form of a resource declaration includes:

- The resource type, which is a word with no quotes.
- An opening curly brace `{`.
- The title, which is a string.
- A colon (`:`).
- Optionally, any number of attribute and value pairs, each of which consists of:
  - An attribute name, which is a lowercase word with no quotes.
  - A `=>` (called an arrow, "fat comma," or "hash rocket").
  - A value, which can have any [data type][datatype].
  - A trailing comma.
- A closing curly brace (`}`).

You can use any amount of whitespace in the Puppet language.

This example declares a file resource with the title `/etc/passwd`. This declaration's `ensure` attribute ensures that the specified file is created, if it does not already exist on the node. The rest of the declaration sets values for the file's `owner`, `group`, and `mode` attributes.

```
file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  mode   => '0600',
}
```

Complete syntax

By creating more complex resource declarations, you can:

- Describe many resources at once.
- Set a group of attributes from a hash with the `*` attribute.
- Set default attributes.
- Specify an abstract resource type.
- Amend or override attributes after a resource is already declared.

The complete generalized form of a resource declaration expression is:

- The resource type, which can be one of:
  - A lowercase word with no quotes, such as `file`.
  - A resource type data type, such as `File`, `Resource[File]`, or `Resource['file']`. It must have a type but not a title.
- An opening curly brace (`{`).
- One or more resource bodies, separated with semicolons (`;`). Each resource body consists of:
  - A title, which can be one of:
    - A string.
    - An array of strings, which declares multiple resources.
    - The special value `default`, which sets default attribute values for other resource bodies in the same expression.
  - A colon (`:`).
  - Optionally, any number of attribute and value pairs, separated with commas (`,`). Each attribute/value pair consists of:
    - An attribute name, which can be one of:
      - A lowercase word with no quotes.
      - The special attribute `*`, called a "splat," which takes a hash and sets other attributes.

        - A =>, called an arrow, a "fat comma," or a "hash rocket".
        - A value, which can have any data type.
    - Optionally, a trailing comma after the last attribute/value pair.
- Optionally, a trailing semicolon after the last resource body.
- A closing curly brace (})

```
<TYPE> { default: * => <HASH OF ATTRIBUTE/VALUE PAIRS>, <ATTRIBUTE> => <VALUE>, ; '<
```

Resource declaration default attributes

If a resource declaration includes a resource body with a title of `default`, Puppet doesn't create a new resource named "default." Instead, every other resource in that declaration uses attribute values from the `default` body if it doesn't have an explicit value for one of those attributes. This is also known as "per-expression defaults."

Resource declaration defaults are useful because it lets you set many attributes at once, but you can still override some of them.

This example declares several different files, all using the default values set in the `default` resource body. However, the `mode` value for the the files in the last array (`['ssh_config', 'ssh_host_dsa_key.pub'....`) is set explicitly instead of using the default.

```
file {
  default:
    ensure => file,
    owner  => "root",
    group  => "wheel",
    mode   => "0600",
  ;
  ['ssh_host_dsa_key', 'ssh_host_key', 'ssh_host_rsa_key']:
    # use all defaults
  ;
  ['ssh_config', 'ssh_host_dsa_key.pub', 'ssh_host_key.pub', 'ssh_host_rsa_key.pub',
    # override mode
    mode => "0644",
  ;
}
```

The position of the `default` body in a resource declaration doesn't matter; resources above and below it all use the default attributes if applicable.You can only have one `default` resource body per resource declaration.

Setting attributes from a hash

You can set attributes for a resource by using the splat attribute, which uses the splat or asterisk character `*`, in the resource body.

The value of the splat (`*`) attribute must be a hash where:

- Each key is the name of a valid attribute for that resource type, as a string.
- Each value is a valid value for the attribute it's assigned to.

This sets values for that resource's attributes, using every attribute and value listed in the hash.

For example, the splat attribute in this declaration sets the `owner`, `group`, and `mode` settings for the file resource.

```
$file_ownership = {
  "owner" => "root",
  "group" => "wheel",
  "mode"  => "0644",
}

file { "/etc/passwd":
  ensure => file,
  *      => $file_ownership,
}
```

You cannot set any attribute more than once for a given resource; if you try, Puppet raises a compilation error. This means that:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if `mode` is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, since the splat itself is an attribute.

To use some attributes from a hash and override others, either use a hash to set per-expression defaults, as described in the section on resource declaration defaults, or use the merging operator, `+` to combine attributes from two hashes, with the right-hand hash overriding the left-hand one.

Abstract resource types

Because a resource declaration can accept a resource type data type as its resource type , you can use a `Resource[<TYPE>]` value to specify a non-literal resource type, where the `<TYPE>` portion can be read from a variable.That is, the following three examples are equivalent to each other:

```
file { "/tmp/foo": ensure => file, } File { "/tmp/foo": ensure => file, } Resource[F
```

```
$mytype = File
Resource[$mytype] { "/tmp/foo": ensure => file, }
```

```
$mytypename = "file"
Resource[$mytypename] { "/tmp/foo": ensure => file, }
```

This lets you declare resources without knowing in advance what type of resources they'll be, which can enable transformations of data into resources.

Arrays of titles

If you specify an array of strings as the title of a resource body, Puppet creates multiple resources with the same set of attributes. This is useful when you have many resources that are nearly identical.

For example:

```
$rc_dirs = [
  '/etc/rc.d',       '/etc/rc.d/init.d','/etc/rc.d/rc0.d',
  '/etc/rc.d/rc1.d', '/etc/rc.d/rc2.d', '/etc/rc.d/rc3.d',
  '/etc/rc.d/rc4.d', '/etc/rc.d/rc5.d', '/etc/rc.d/rc6.d',
]

file { $rc_dirs:
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}
```

If you do this, you must let the namevar attributes of these resources default to their titles. You can't specify an explicit value for the namevar, because it applies to all of the resources.

Adding or modifying attributes

Although you cannot declare the same resource twice, you can add attributes to an resource that has already been declared. In certain circumstances, you can also override attributes. You can amend attributes with either a resource reference, a collector, or from a hash using the splat (`*`) attribute.

To amend attributes with the splat attribute, see the section about setting attributes from a hash.

To amend attributes with a resource reference, add a resource reference attribute block to the resource that's already declared. Normally, you can only use resource reference blocks to add previously unmanaged attributes to a resource; it cannot override already-specified attributes. The general form of a resource reference attribute block is:

- A resource reference to the resource in question
- An opening curly brace
- Any number of attribute => value pairs
- A closing curly brace

For example, this resource reference attribute block amends values for the `owner`, `group`, and `mode` attributes:

```
file {'/etc/passwd':
  ensure => file,
}

File['/etc/passwd'] {
  owner => 'root',
  group => 'root',
  mode  => '0640',
}
```

You can also amend attributes with a collector.
The general form of a collector attribute block is:

- A **resource collector** that matches any number of resources
- An opening curly brace
- Any number of attribute => value (or attribute +> value) pairs
- A closing curly brace

For resource attributes that accept multiple values in an array, such as the relationship metaparameters, you can add to the existing values instead of replacing them by using the "plusignment" (+>) keyword instead of the usual hash rocket (=>). For details, see appending to attributes in the **classes** documentation.

This example amends the `owner`, `group`, and `mode` attributes of any resources that match the collector:

```
class base::linux {
  file {'/etc/passwd':
    ensure => file,
  }
  ...}

include base::linux

File <| tag == 'base::linux' |> {
 owner => 'root',
 group => 'root',
 mode => '0640',
}
```

CAUTION: Be very careful when amending attributes with a collector. Test with `--noop` to see what changes your code would make.

- It can override other attributes you've already specified, regardless of class inheritance.
- It can affect large numbers of resources at one time.
- It implicitly realizes any virtual resources the collector matches.
- Because it ignores class inheritance, it can override the same attribute more than one time, which results in an evaluation order race where the last override wins.

Local resource defaults

Because resource default statements are subject to dynamic scope, you can't always tell what areas of code will be affected. Generally, do not include classic resource default statements anywhere other than in your site manifest (`site.pp`). See the **resource defaults documentation** for details. Whenever possible, use resource declaration defaults, also known as per-expression defaults.

However, resource default statements can be powerful, allowing you to set important defaults, such as file permissions, across resources. Setting local resource defaults is a way to protect your classes and defined types from accidentally inheriting defaults from classic resource default statements.

To set local resource defaults, define your defaults in a variable and re-use them in multiple places, by combining resource declaration defaults and setting attributes from a hash.

This example defines defaults in a `$file_defaults` variable, and then includes the variable in a resource declaration default with a hash.

```
class mymodule::params {
  $file_defaults = {
    mode  => "0644",
    owner => "root",
    group => "root",
  }
  # ...
}

class mymodule inherits mymodule::params {
  file { default: *=> $mymodule::params::file_defaults;
    "/etc/myconfig":
      ensure => file,
    ;
  }
}
```

What should we know about this page?

Email Address:

Where can we reach you to follow up?

Submit

If you leave us your email, we may contact you regarding your feedback. For more information on how Puppet uses your personal information, see our **privacy policy**.

Use Cases

Continuous Delivery

Continuous Compliance

Incident Remediation

Configuration Management

Knowledge & Support

Get Support

Knowledgebase

Product Documentation

Open Source at Puppet

Forge: Modules and Plugins

Connect

About

Community

Puppet Blog

Work With Us

Press and News

Contact Puppet

Shop

puppet

Puppet automates the delivery and operation of the software that powers some of the biggest brands in the world.