

id: intro title: Introduction to Puppet allowDiscussion: None subject: description: An Introduction to the Puppet Framework. contributors: creators: jesse effectiveDate: 2005/08/10 10:29:59.140 GMT-5 expirationDate: None language: en rights: creation_date: 2005/07/29 11:22:40.919 GMT-5 modification_date: 2005/08/10 10:29:59.142 GMT-5 layout: document_view Content-Type: text/x-rst

Introduction

Puppet is a system configuration tool. Currently it contains a library and a language for manipulating that library. It has a stub in place for both a client and a server, but there is currently no network support.

The library is entirely responsible for all action, and the language is entirely responsible for expressing configuration choices. Everything is developed so that the language operations can take place centrally on a single server (or bank of servers), and all library operations will take place on each individual client. Thus, there is a clear demarcation between language operations and library operations, as this document will mention.

Language

The language is declarative, and is modeled somewhat after cfengine's concept of actions, action instances, and instance parameters. Here is how a file action would look in puppet:

```
file { "/etc/passwd":  
  owner => root,  
  group => root,  
  mode => 644  
}
```

Each type must have some kind of 'name' parameter defined (although this might change), and this is what goes between the brackets. Any string that doesn't match `\w+` or `\d+` needs to be quoted (e.g., file names).

There are two ways for types to be available in the language: They'll either be **primitive types** included in the Puppet framework itself, or they'll be **defined types** resulting from definitions or server classes.

Definitions

Because experience with cfengine showed that objects of different types are often related, puppet focuses on making creation of associations between objects of any type easier than making many objects of the same type. The fundamental unit of association is a 'definition':

```
# currently, the variables don't have '$' attached in the definition proto-  
type  
# this might change  
define sudo(source,group) {  
  package { sudo:  
    version => "1.6.7"  
  }  
  file { "/etc/sudoers":  
    source => $source, # parameterization; copy the file from where?  
    mode => 440 # the 'file' type converts this appropriately to  
              # octal  
  }  
  file { "/usr/sbin/sudo":
```

```

    owner => root,
    group => $group
  }
}

```

A definition has its own scope, so variables defined within it are only available within the definition itself.

This definition can now be treated as a type:

```

sudo {
  group => root,
  source => "http://fileserver/files/sudoers?v=production"
}

```

One significant difference between definitions and types is that types generally have many associated parameters, most of which are optional, while a given definition will (at this point) always require all parameters. You can manage as little or as much of a file as you want, for instance, but once you wrap it in a definition you have to pass every specified parameter to instances of the definition.

It is appropriate to think of definitions as declarative functions; referring to the name of a definition results in the associated declarative code being executed with the provided parameters.

I will likely soon add the ability to provide defaults to definition parameters; any other feature requests are welcome.

Classes

In addition to definitions, the language supports server classes. At this point, the only syntactical or functional difference between the two is that server classes can have a parent class:

```

class base {
  sudo { ... } # an existing definition
}

class server inherits base {
  ssh { ... } # an existing definition
}

if $hostname == "myhost" {
  server{} # the braces are mandatory at this point
} elsif $hostname == "otherhost" {
  base{}
}

```

The classes also support parameterization, just like definitions. Notice that the mechanism for associating nodes with servers is stupid. I considered the following syntax:

```

node host {
  server {}
}

```

But that doesn't seem much better. Again, recommendations are accepted.

Having a parent class means that that parent's objects are all operated on before any of the subclass's are. So, in this case, each member of 'server' would first verify 'sudo' is set up correctly and then check 'ssh'.

Variables

Because it is assumed that strings will be used far more than variables, simple strings don't have to be quoted or otherwise marked, but variables must have the `$` attached:

```
$group = "root"

file { "/etc/sudoers":
  group => $group
}
```

Strings and booleans (`true` and `false`) are the only data types; even numbers are converted to strings. Arrays are supported, although their behaviour has not been characterized for all cases. One particular use of arrays is for implicit iteration:

```
$files = ["/etc/passwd", "/etc/group", "/etc/fstab"]

file { $files:
  owner => root,
  group => root
}
```

This implicitly iterates across the file list and performs all of the appropriate checks.

Currently, `puppet` collects as much information as it can (using the `Facter` library) and sets all of it as top-level variables. So, you can expect variables like `$operatingsystem` and `$ipaddress`.

Also, because `puppet` is a declarative language, reassigning a variable within the same scope is currently an error. The language is written such that this could be disabled (please let me know if you need this feature), but it is currently always enabled. You can override defaults in a lower scope:

```
$var = default

define test {
  $var = override
  ...
}
```

It is expected that parameter passing to definitions will be the primary mechanism for variable assignment, so it seems unlikely that this limitation will cause much trouble.

Importing

Files can be imported using the `import` command:

```
import "filename"
```

There is currently no search path or anything; files are looked for in the same directory as the file doing the importing. Each file should have its own lexical scope, but I haven't yet figured out how to do that.

Control Structures

There are currently two basic control structures, one meant to return a value and the other just a normal 'if/elsif/else' structure.

Selectors

One of the primary goals of Puppet is to simplify building a single configuration that works across multiple machines and machine classes. One mechanism for this is currently called a 'selector'; it is similar to the trinary operator `?:`:

```
$value = $variable ? {
  value1 => "setvalue1",
  value2 => "setvalue2"
}
```

This sets the variable `$value` depending on the value of `$variable`. If it is `value1`, then `$value` gets set to `setvalue1`, etc.

The brackets can be in either part of the expression, or not at all:

```
$value = $variable ? "value1" => "setvalue1"
```

I think the results of selector statements are currently undefined if the value of the variable does not match a value within the selector.

These structures are useful for simplistic abstraction across platforms:

```
file { "/etc/sudoers":
  owner => root,
  group => $operatingsystem ? {
    SunOS => root,
    Linux => root,
    FreeBSD => wheel
  }
}
```

If

Puppet currently supports a normal If/Elsif/Else structure, and each statement introduces a new lexical scope. Things that can be tested are currently extremely limited (just simple comparisons, e.g., `'=='`, `'<='`, etc.); it should be easy to add `&&`, `||`, and parentheses for extended tests, but I don't yet know how I'll handling testing more complex expressions.

I expect that this structure will go away, to be replaced with a more declarative structure.

Library

This section discusses some aspects of the internals of the Puppet library. This information can be useful but is not critical for use and understanding of Puppet.

The library is composed of two fundamental types of objects: Types and states (yes, the terminology is atrocious). States are things that can be configured to change one aspect of an object (e.g., a file's owner), types are essentially named collections of states. So, there is a File type, and it is a collection of all of the states available to modify files.

In addition to states, which necessarily modify 'the bits on disk', as it were, types can also have non-state parameters which modify how that type instance behaves:

```
file { "/bin":
  owner => bin,
  recurse => true
}
```

The `recurse` parameter to `file` does not modify the system itself, it modifies how the `file` type manages `/bin`.

Not all types as expressed in the language are types defined in the library; some language types are actually definitions or server classes. Types that are defined in the language itself are termed **primitive types** or just **primitives**, and types that generate other types are termed **metatypes**.

At this point, all types have a unique name (within the current scope, but it probably should actually be global). This name is set using a class instance variable:

```
# in the library
class Mytype < Puppet::Type
  @name = :mytype
  ...
end

# in the language
mytype { "yay": ... }
```

The states similarly have unique names, although this uniqueness is only per-type. When a type is defined, all states and parameters are also listed as class instance variables; states are listed by class, and parameters with symbols. Here is the `File` type declaration:

```
@states = [
  Puppet::State::FileCreate,
  Puppet::State::FileUID,
  Puppet::State::FileGroup,
  Puppet::State::FileMode,
  Puppet::State::FileSetUID
]

@parameters = [
  :path,
  :recurse
]
```

`Puppet::Type` gets notified of a subclass, and it iterates over the states retrieving each name and making a hash from the name to the class.

Lastly, each type must either provide a state or parameter of `':name'`, or it must set `'@namevar'` so that the system knows what is considered the name:

```
@namevar = :path
```

With this declaration, `file { "/tmp/file": }` is basically equivalent to `file { path => "/tmp/file" }`.