

id: big-picture title: Puppet: The Big Picture allowDiscussion: None subject: description: An Overview of the Puppet framework with brief examples. contributors: creators: jesse effectiveDate: None expirationDate: None language: en rights: creation\_date: 2005/07/29 10:50:27.437 GMT-5 modification\_date: 2005/08/02 11:36:05.600 GMT-5 layout: document\_view Content-Type: text/x-rst

## Big picture

Puppet is a **declarative language** for expressing system configuration, a **client and server** for distributing it, and a **library** for realizing the configuration.

Rather than approaching server management by automating current techniques, Puppet reframes the problem by providing a language to express the relationships between servers, the services they provide, and the primitive objects that compose those services. Rather than handling the detail of how to achieve a certain configuration or provide a given service, Puppet users can simply express their desired configuration using the abstractions they're used to handling, like **service** and **node**, and Puppet is responsible for either achieving the configuration or providing the user enough information to fix any encountered problems.

This document is a supplement to the [Introduction](#) and it is assumed that readers are familiar with the contents of that document.

## Less Detail, More Information

A correct configuration must obviously provide the appropriate details, but a configuration tool should understand that the details are generally the easy part. The hard part of system configuration is understanding the complex relationships between services and the objects that comprise those services. One of the primary goals of Puppet is to allow users to push the details into lower levels of the configuration and pull the relationships into the foreground.

For instance, take a typical Apache web server deployment. Puppet allows one to encapsulate all of the primitives necessary for successful deployment into one reusable object, and that object can even be abstracted to support multiple apache versions. Here's how a simple apache definition might look for a Debian server (Debian uses **apache** for 1.x versions and **apache2** for 2.x versions):

```
define apache(version, conf, user, group) {
  # abstract across apache1 and apache2
  $name = $version ? {
    1 => "apache",
    2 => "apache2"
  }
  package{ $name:
    install => true
  }

  file { $conf:
    user => $user,
    group => $group,
    source => $conf
  }

  # we want the service to restart if the config file changes
  # or if the package gets upgraded
  service { $name:
    running => true,
    requires => [file[$conf], package[$name]]
  }
}
```

```
    }  
  }  
}
```

Now, with this configuration, one can easily set multiple servers up to run different versions of apache. The key benefit here is that the information necessary to run apache correctly is separated from the decision to do so on a given host. For example:

```
# import our apache definition file  
import "apache"  
  
node server1 {  
  # use a locally-available config file  
  apache {  
    version => 1,  
    conf => "/nfs/configs/apache/server1.conf",  
    user => www-data,  
    group => www-data  
  }  
}  
  
node server 2 {  
  # use a config that we pull from elsewhere  
  apache {  
    version => 2,  
    conf => "http://configserver/configs/server2/httpd.conf"  
    user => www-data,  
    group => www-data  
  }  
}
```

Notice that our node configuration only specifies 1) that a given server is running Apache, and 2) the information necessary to differentiate this instance of apache from another instance. If a given detail is going to be the same for all apache instances (such as the fact that the service should be running and that it should be restarted if the configuration file changes), then that detail does not have to be specified every time an Apache instance is configured.

## Describing Configuration

Puppet's declarative language separates the "what" of the configuration from the "how".

### Language Example: Class Hierarchy using Inherit

A Site Configuration is made of high level declarative statements about policy. A Policy of having all hosts monitored (to ensure uptime, report performance/load, or audit log files) can be implemented with inheritance (as with "server\_host") or with composition (as in workstation):

```
define monitored_component (report_server) {  
  #(registers w/ monitoring software)...  
}  
  
class monitored_host {  
  $report_server = "somehost.domain.com"  
  monitored_component { "mon_comp" :  
    report_server => $report_server  
  }  
}
```

```

}
class server_host inherits monitored_host {
    #(other definitions/classes specific to servers)....
}
#... or ...
class workstation_host {
    $salt_report_server = "someotherhost.domain.com"
    monitored_component ( "workstation_mon" :
        report_server => $salt_report_server
    #(other definitions specific to workstations)....
}

```

## Language Example: Type Composition using Function Definitions

The following example describes the services (objects) and relationships (definitions) for a simple web site.:

```

define webserver (port, htmldoc, default-config-file) {
    #...(configures httpd to serve files at path "htmldoc")...
}
define firewall_port (external-port, internal-port) {
    #...(stores the port aspects for a "firewall" object to
    #later build a firewall config)...
}

```

By defining the Webserver/Firewall objects, the details of behavior not relevant to a simple website (such as the default configuration of a httpd server or the process of registering the firewall port aspect can be enclosed in an abstraction.

One model for a Simple website defines the relationship between two components: a webserver to respond to the HTTP requests, and a firewall to limit traffic and protect the webserver host. They require two parameters: The external port to receive HTTP requests, and the collection of files that the webserver will offer as webpages.:

```

define simple_web_site_service (external-port,data-filepath) {
    $internal-port = 80
    $default_config_file = "...httpd.conf"
    webserver { "simple_webserver" :
        port => $internal-port
        htmldoc => $data-filepath,
        default-config-file => $default-config-file }

    firewall_port { "simple_firewall_port" :
        external-port => $external-port,
        internal-port => $internal-port }

    requires ( simple_webserver, simple_firewall_port )
}

```

This definition of the service declares the components as related objects. The Web Site Service requires a webserver and an open port on the firewall component.

As this configuration is deployed to different operating systems (RedHat linux, solaris, etc.), the appropriate webserver, web-data, firewall object class can be called by the Puppet framework. The lower level details of how each OS specific component implements its configuration is separate from site policy.

You can read more about the Puppet language in the [Introduction](#). (Add link to user manual, when it's written)

## Distributing Configuration

The Puppet framework Library consists of a client and server.

(picture/diagram: client, server, site-config -> host-config)

A Puppet server is aware of the full configuration. As some component's configuration aspects depend on the configuration of other components (e.g. the firewall config includes the ports used by webservers), generating configuration for a component requires being aware of full configuration.

A Puppet client that runs on a specific host (or perhaps the same host as its Puppet Server) is generally only concerned with the components to be configured on that host.

Puppet Clients normally request or "pull" configuration from their server. The Server processes the configuration request for the host using a pre-generated tree model of the classes and definitions from the site-config.

When configuration needs to be "pushed" to the clients, the Server can be asked to attempt to trigger each client to request "pull" a new host configuration.

Example: Puppet server, library

You can read more about the Puppet language in the [Introduction](#). (Add link to user manual, when it's written)

## Realizing the Configuration

The Puppet Client Library contains the component knowledge of how to reach desired states and configurations for several objects: File, Package, etc.

Example: Puppet Client, library:

```
...todo: file: transfer, mode, ownership...
```

You can read more about the Puppet language in the [Introduction](#). (Add link to user manual, when it's written)

Some components such as the webserver and firewall, from the simple-website example, require additional code to reach "closure". "Closure" is when the component is entirely responsible for implementing its own configuration.

Much as database applications abstract the mechanics of storing, indexing, and searching their data, a component ideally should abstract the specifics of how to store, confirm, and implement its requested configuration.

To learn more, review other Puppet [Documentation](#) and sample [Configurations](#).