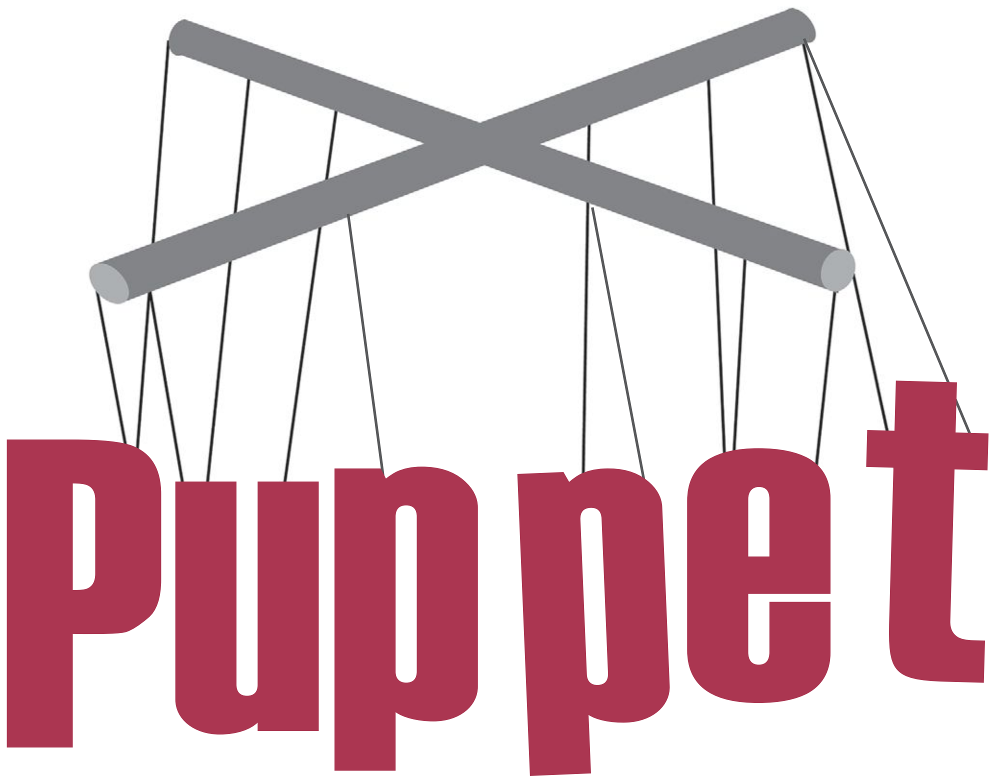
SEE
TEXT
ONLY



Puppet on FreeBSD

PRIOR TO PUPPET THE MAIN CONFIGURATION MANAGEMENT TOOL WAS ONE CALLED CFENGINE THAT STARTED IN 1993. PUPPET WAS FIRST RELEASED IN 2005 AND HELPED KICKSTART SOMETHING OF A REVIVAL IN THE CONFIGURATION MANAGEMENT SPACE. NOW IT IS WIDELY CONSIDERED TO BE ONE OF THE MOST POPULAR CONFIGURATION MANAGEMENT TOOLS OUT THERE.

By Brad Davis and Andrew Fengler

Programmatic Configuration

The main strength of Puppet, and any configuration management tool, is the ability to control configuration elements in a programmatic manner. Puppet accomplishes this with an object-oriented system. The most basic element is a resource definition, where a resource could be a file, a package, a command to be executed, or any number of similar things. And if the built-in resource types are not enough, it is possible to define custom resources. Resource definitions along with conditional structures are used to construct classes. These classes are the primary elements giving control on a per-host basis. Each host, or node as Puppet calls them, has a node definition where configuration for that specific host is controlled. Although it is possible to configure all of the resources here, it makes more sense to create classes that can be reused: for instance, an indefinite number of webservers can easily be brought up to an identical state by simply including a class that defines how the webservers should be configured.

Applying Configurations

There are two ways to run Puppet. The first is to use `puppet apply` to install from a local set of configurations. This is the simplest, but it requires some method of copying the configurations to each node. It also lacks the ability to use PuppetDB. The other method is to run a dedicated Puppetmaster. In this configuration, the nodes will all connect into the central Puppetmaster and fetch the configs from there. The other advantage of this system is that the nodes will send all their facts to the Puppetmaster. Facts are snippets of information about the node, such as the OS, netmask, number of cpu cores, etc. These facts can be used as variables in conditionals allowing control of resources: for instance, it is possible to select a package repository based on the OS version. With all the facts on the Puppet master, a utility such as Puppetboard will provide a view of what's happening on all the servers. It will show breakdowns of facts, the results of Puppet executions, and details on what's changing or has failed. Facts are provided by a library that Puppet depends on called 'Facter'. This tool can be run from the command line to inspect the results on a given machine. The Puppet-specific facts can be viewed by running:

```
# puppet facts
```

Running on FreeBSD

Puppet, Puppetserver, and PuppetDB can all be installed from the ports tree, making getting started on FreeBSD very simple. For example, to install the current version of Puppet as of the time of this writing, use:

```
# pkg install puppet4
```

Other versions are available to be backwards compatible, as long as they are supported by the Puppet project.

The latest version of PuppetDB can be installed by:

```
# pkg install puppetdb4
```

PuppetDB is not required, but does provide some advanced features to a Puppet installation. For example, the ability to use facts from other nodes on a different node. We will not be covering PuppetDB in this article, but want to mention its availability for advanced users.

From there, simply write a few resource definitions in a class, import that in a node definition, then point the Puppet agent at the server or configurations—depending on which method you picked—and Puppet is up and running! Then comes the fun part of managing everything. There are surprisingly few gotchas for FreeBSD, and as of Puppet 4, pkgng is a supported package provider out of the box. Configuration files can be easily managed by using templating. Puppet uses a templating system from Ruby called ERB, which allows embedding of Ruby code itself into the files and has access to all variables Puppet knows about the environment.

Example Using agentless

Puppet in so-called agentless mode is a good way to test recipes before applying them to lots of machines. It is very handy to just spin up a jail or VM quickly and run apply to test the configuration.

For example, if we create a file called `'test.pp'` and it contains the following recipe to install nginx:

```
package { 'nginx':
    ensure => installed,
}
```

it can easily be executed by running:

```
# puppet apply test.pp
```

For more debugging information to see what might not be working, enabling full debugging mode and verbose mode can be very helpful:

```
# puppet apply -d -v test.pp
```

Example Using a master/agent

After installing Puppet, on the master, enable it with:

```
# sysrc puppet_master_enable=YES
```

Then start up the master:

```
# service puppet_master start
```

On the client (which could be the same machine):

```
# sysrc puppet_enable=YES
```

Modify the puppet.conf to point to the IP or hostname of the master. In this case, we will be pointing to localhost:

```
master: 127.0.0.1
```

And start up the client:

```
# service puppet start
```

Once the master and agent are started, the agent will attempt to connect to the master. They will then create an internal Certificate Authority to authenticate each other. The agent will generate a client certificate and upload it to the master. You will be able to see the agent waiting for the master to accept its certificate in the logs. This certificate can be viewed on the master by running:

```
# puppet ca list
```

Then you should see the certificate from your client. It will look like:

```
"hostname" [sha256 checksum of certificate request]
```

Now sign the certificate:

```
# puppet ca sign hostname
```

The agent should receive the certificate and continue on with its run. However, it's not going to do very much, and will spit out a big yellow warning:

```
ERROR: Could not fetch my node definition, but the Puppet agent run will continue
```

Looks like we'll need to create a node definition.

Creating a Simple Configuration

Let's start by going to our Puppet configuration directory, /usr/local/etc/puppet/ by default. Our configuration environments go in the "environments" directory. You can have multiple environments, and the Puppet agent can be configured to select which environment it uses. This means you can have separate branches for development and so on, but, for now, the default "production" environment should be perfect for testing. Each environment has a "manifests" and a "modules" directory. Site-wide manifests, such as common configuration and node definitions, go under manifests, and modules holds reusable blocks of configuration.

Puppet files end in .pp, so let's create our node definition. A node definition matches the hostname of the node, either exactly, or a regex match. Create localhost.pp with the following content:

```
node 'localhost.example.com' {
   $host_desc = "My computer"
   $colorscheme = "desert"

   file { "/etc/test":
      ensure    => present,
      user      => "root",
      group     => "wheel",
      mode      => "0644",
      content   => "Hello World! This is ${host_desc}\n",
   }
}
```

And that's a simple node definition. We set two variables to use later and created your first file. You can see that the file has a number of options set, namely its owner, group, permissions (mode), and what goes in the file. Save this, and do another Puppet run. If you started the Puppet service, it will run every so often, but if you don't want to wait, you can trigger a run with the `--test` or `-t` flag:

```
# puppet agent -t
```

And that will set off a run. When it's finished, you'll have a shiny new file.

Now let's create our first module. Let's say we want our favorite editor and all its settings on our computer. Under the modules directory, create a directory called "vim". Inside that, a number of subdirectories are used to hold different parts of the module. Create a "manifests" directory; this will hold all the Puppet configuration for this class. For modules, there must always be a class that has the same name as the module in a file called "init.pp". This is the starting point of the module, and we can add more files with extra classes to include if needed. Create init.pp:

```
class vim {
   package { "vim-lite":
      ensure => latest,
      name   => "editors/vim-lite",
      provider => "pkgng",
   }
   file { "/home/beastie/.vimrc":
      ensure => present,
      user     => "beastie",
      group  => "beastie",
      mode   => "0644",
      source => "puppet:///modules/vim/vimrc",
   }
}
```

We've managed two more resources: a package that will install vim-lite using pkgng, and a file that will install our vimrc. Notice how we used `"source"` instead of `"content"`. Instead of placing some text in a file, this will get a file from the Puppetmaster. These files also go in the `"vim"` module, but under a `"files"` subdirectory instead of in the manifests directory. The format for the file URLs is `"puppet:///modules/<module name>/<file name>"`. So, our `"puppet:///modules/vim/vimrc"` URL will look for `"modules/vim/files/vimrc"` under the current environment.

Every resource type offers a different set of options. For files, there are a wide variety of options, one of which is `"ensure"`. This determines what state we want the file to be in. When we're done with that file, you can clean it up by setting ensure to `"absent"`.

Now that we have a class, just add the line `"include vim"` to our node definition, and it will add the entire vim class. We can add it to as many nodes as we want with just a single line. But let's say we need to change a setting on some servers. This is easy to do with a template.

Templating

Templating is a very powerful feature in any automation system. The templating system has access to all the variables from Facter, so it knows all about the machine. It allows for very complex configuration across many machines to be expressed simply. For example, to template a configuration file where it is necessary to do something like set the IP address to bind to, first create a resource for the file like:

```
file { '/usr/local/etc/nginx/nginx.conf':
   ensure     => 'file',
   owner      => 'root',
   group      => 'wheel',
   mode       => '644',
   content    => template('/usr/local/etc/puppet/templates/nginx.conf'),
   require    => Package['nginx'],
}
```

The templating engine works on template code found between "<%=" and "%>". Now here is a snippet of what the template could contain:

```
http {
   server {
      listen <%= @ipaddress %>:80;
      server_name <%= @fqdn %>;
      location / {
         root /usr/local/www/nginx;
         index index.html;
   }
}
```

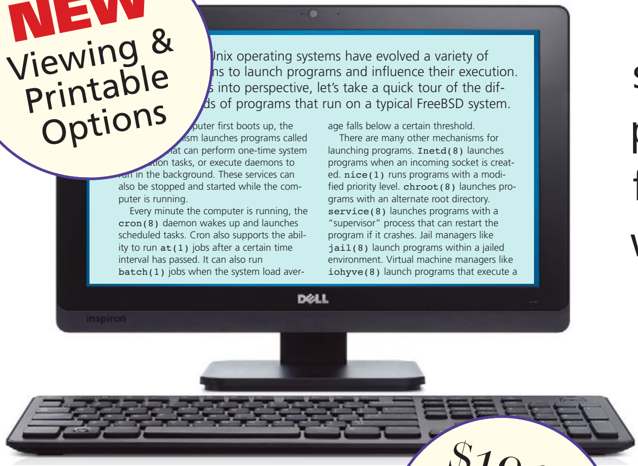
Note that the variables begin with an "@" symbol. This will give us a result like:

```
http {
   server {
      listen 192.168.11.10:80;
      server_name test.example.com;
      location / {
      root /usr/local/www/nginx;
      index index.html;
   }
}
```

There are many more powerful things that can be done with templating including using Ruby functions to create more advanced functionality. As a slightly more complicated example of what can be done with templating, one time it was used to modify one of the variables using the Ruby gsub function

FreeBSD™ JOURNAL

NEW
Viewing & Printable Options


Unix operating systems have evolved a variety of ns to launch programs and influence their execution. s into perspective, let's take a quick tour of the different ds of programs that run on a typical FreeBSD system.

puter first boots up, the ism launches programs called hat can perform one-time system tion tasks, or execute daemons to run in the background. These services can also be stopped and started while the computer is running.

Every minute the computer is running, the cron(8) daemon wakes up and launches scheduled tasks. Cron also supports the ability to run at(1) jobs after a certain time interval has passed. It can also run batch(1) jobs when the system load average falls below a certain threshold.

There are many other mechanisms for launching programs. Inetd(8) launches programs when an incoming socket is created. nice(1) runs programs with a modified priority level. chroot(8) launches programs with an alternate root directory. service(8) launches programs with a "supervisor" process that can restart the program if it crashes. Jail managers like jail(8) launch programs within a jailed environment. Virtual machine managers like iohyve(8) launch programs that execute a

The DE, like the App, is an individual product. You will get an email notification each time an issue is released.

$19.99 YEAR SUB
$6.99 SINGLE COPY

The Browser-based Edition (DE) is now available for viewing as a PDF with printable option.

The Broswer-based DE format offers subscribers the same features as the App, but permits viewing the *Journal* through your favorite browser. Unlike the Apps, you can view the DE as PDF pages and print them.

To order a subscription, or get back issues, and other Foundation interests, go to

www.freebsdfoundation.org

and then the result was embedded in the file.

Dependencies

One issue you can run into is that some resources will require other resources in order to function. Starting a service before you have its config file in place will not likely do what you want. Dependencies are a lot better in Puppet 4, as resources are now applied in the order you place them in files. Previously, this was not the case as they were sorted prior to execution. However, you may still need to specify a dependency order.

The primary way is with the `"require"` keyword. There are two ways to use it. You can require an entire class:

```
require nginx
```

This works exactly like `"include nginx"`, but will ensure the entire class is applied before continuing. The other way is to require another resource:

```
package { 'zsh':
   ensure     => 'installed',
}
   user { "beastie":
   ensure     => "present",
   comment    => "Beastie",
   groups     => ["beastie", "wheel"],
   shell      => "/usr/local/bin/zsh",
   require    => Package['zsh'],
}
file { "/home/beastie":
   ensure     => "directory",
   mode       => "0644",
   owner      => "beastie",
   group      => "beastie",
}
```

This user will depend on the Z Shell package being installed, and because the files depend on the owner and group, those will automatically depend on the user.

Node Interaction

You may need servers to be aware of each other, whether to whitelist IPs, or to know where to listen for a heartbeat. The difficulty is that normally this would require hardcoding long lists of addresses. Puppet has a feature called "exported resources". These resources are much like normal resources, but instead of being installed on the host, they are stored on the Puppetmaster for any node to collect. Resources can also be tagged, making it easy for a server to pick up an entire directory of configs with a single line.

We can export a file like this:

```
@@file { "/usr/local/etc/nagios/puppet.d/jail_${hostname}.dns.cfg":
     ensure     => present,
     content => template('nagios/server.cfg.erb'),
     tag        => "nagios_config",
}
```

And then pick up any files tagged `"nagios_config"` on another host:

```
   File <<| tag == "nagios_config" |>> {
}
```

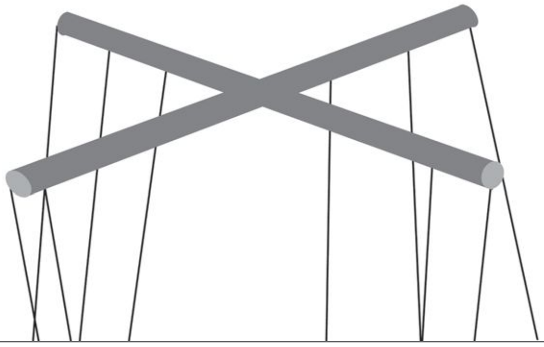
This will create the file on the server that picks it up. Note that we use the hostname in the filename. This way if we export configs from several hosts, they won't collide.

Conclusion

There are a few different ways to view the documentation, but one of the best sources is the Puppet documentation website available here: https://docs.puppet.com/puppet/latest/. Additionally, for help for specific Puppet commands, from the cli:

```
      puppet help <cmd>
```

This provides a good foundation to getting started with Puppet and making life easier for any Systems Administrator or anyone else interested in automating building of their environments. •



ANDREW FENGLER is a Unix administrator working for ScaleEngine Inc. in Hamilton, Canada, where he manages a large number of Unixen with Puppet. He has two years' experience administering FreeBSD systems at scale.

Professionally, BRAD DAVIS has been infrastructure systems architect, developer, and is now a consultant. He has been a FreeBSD committer for over 10 years and involved in many different areas of the Project. Initially starting out as a documentation committer, he used that to launch into helping the Cluster Administration and Postmaster teams. After retiring from those teams, he dabbled in pkg and poudriere. These days he is working on various ports, packaging the FreeBSD base system and a project called RaspBSD to nicely package up FreeBSD and some extra tools for ARM boards like the BeagleBone Black and the Raspberry Pi.