
Puppet System

Configuration Management

What is Puppet?



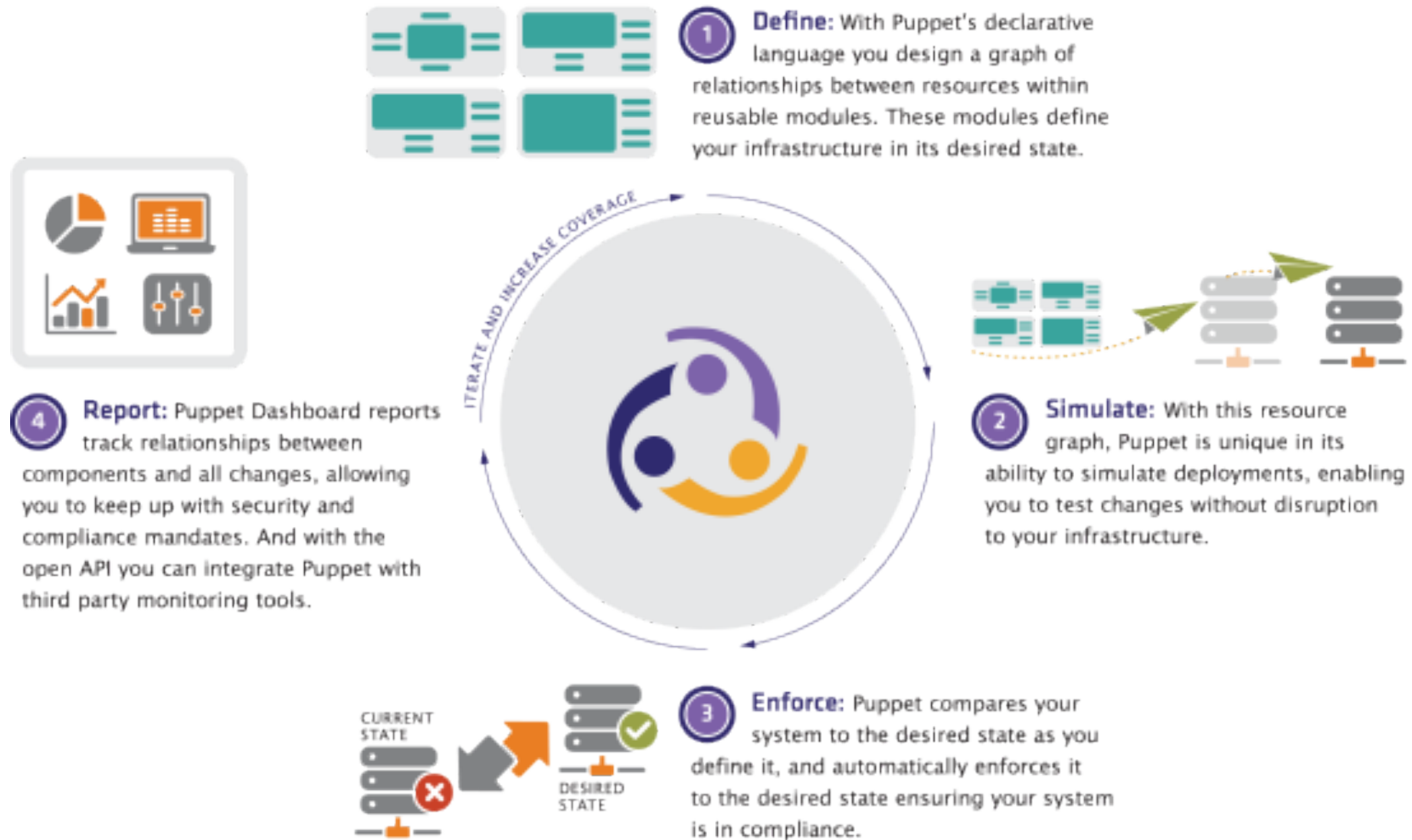
Configuration Management

State Maintenance

Policy Enforced Consistency and
Auditing

Open-Source Framework that
centrally manages IT Systems

How Puppet Works

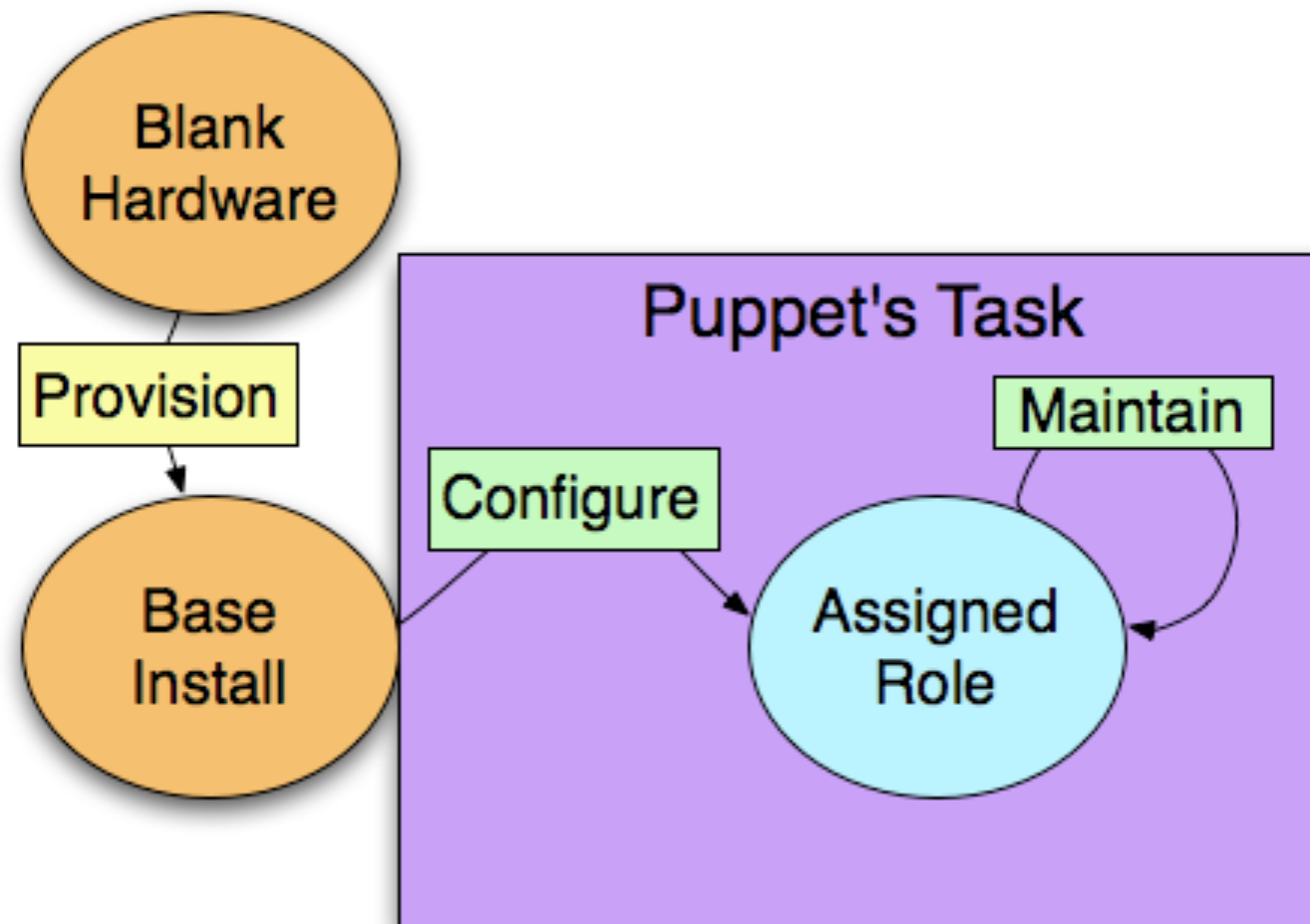


Puppet's Benefits

- Centrally Manage Thousands of Systems
- Ensure Systems are in a “Blessed” State
- Eliminate Inconsistency Between Hosts
- Security and Regulation Compliance
- Less Time managing System Drift

What Puppet Does for “Us”

Assigns a Machine’s Desired Role



Maintains That Role

The Major Working Parts of Puppet

- Puppet Server
 - Resources
 - Classes
 - Manifests
 - Modules
 - Facter Facts
- Puppet Client
 - Facter

Puppet Server

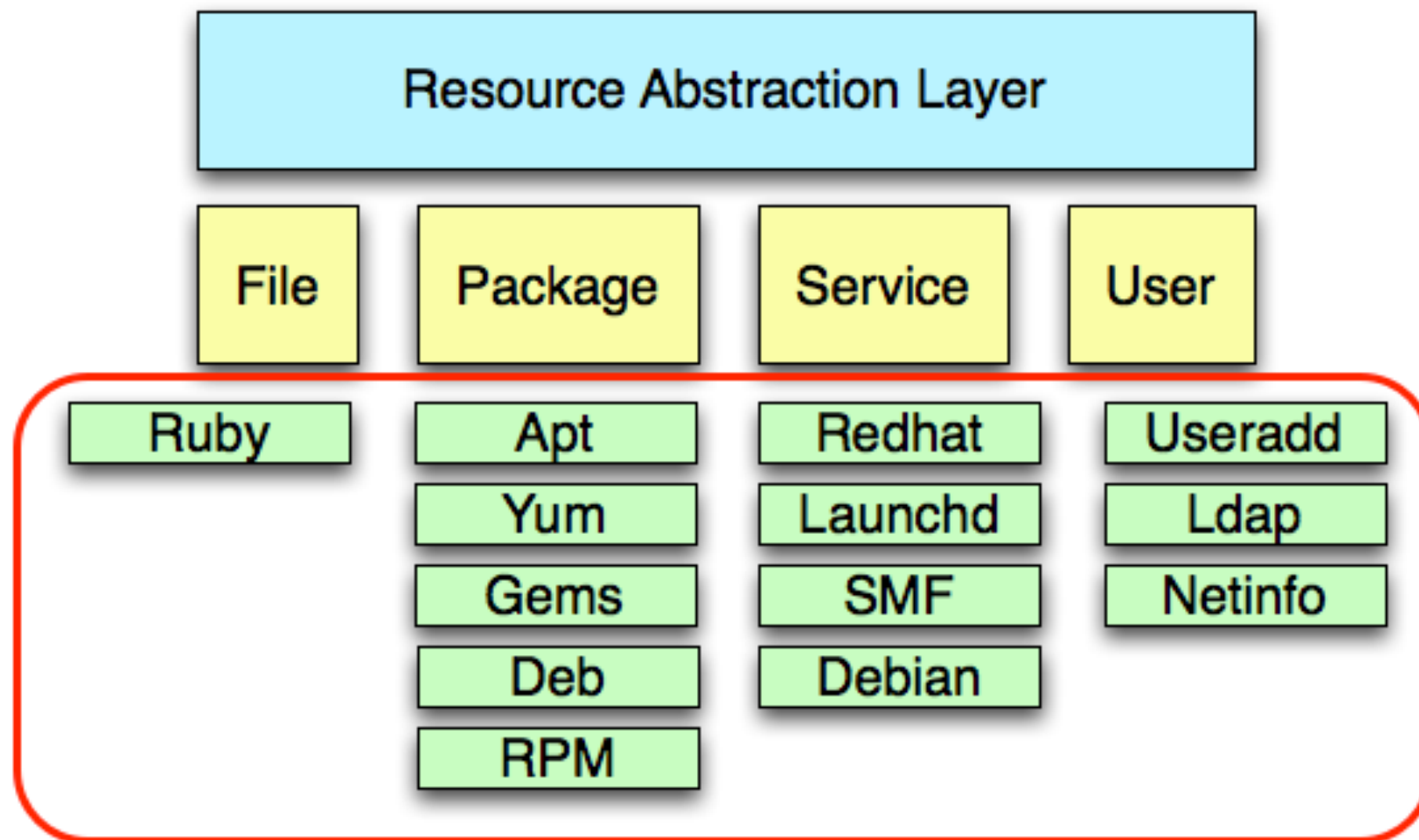
- Currently on puppet.example.com
- Configuration found in /etc/puppet/puppet.conf
- Main server lives in /etc/puppet
- Modules are in /etc/puppet/modules
 - Separated by function
 - “environment” is the largest and most heavily used module
 - dev in /etc/puppet/environments/development
 - testing in /etc/puppet/environments/testing
 - staging in /etc/puppet/environments/staging
- Node lists/configurations in /etc/puppet/manifests
- Starts automatically at boot time via /etc/init.d/puppetmaster

Puppet Client

- Retrieves the client configuration from the puppetmaster and applies it to the local host.
- Currently Two test nodes
 - puppet.example.com
 - puppetclient.example.com
- Configuration found in /etc/puppet/puppet.conf
 - Runs manually in our implementation
 - can be started via init/launchd script
 - Can run out of Cron on a 'less than frequent' basis

Resources and Abstraction

Puppet's Resource abstraction provides a consistent model for resources, across platforms. The resource types (File, Package, etc.) are the interface to the underlying OS "provider" types.



Resource Declarations

Resources are the building blocks Puppet uses to model system configurations. Each resource describes some aspect of a system such as a file, user, a service that must be running, or a package that must be installed.

Resource Declarations:

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

```
service { "ntpd":  
  ensure => running,  
  hasstatus => true,  
  hasrestart => true,  
  enable  => true,  
}
```

```
user { "root":  
  ensure => present,  
  uid    => 0,  
  gid    => 0,  
  home   => "/root"  
  shell  => "/bin/bash",  
  managehome => true,  
}
```

```
package { "iptables":  
  ensure => present,  
}
```

Anatomy of a Resource Declaration

Puppet uses a “declarative language” as its configuration elements. The format and syntax are simple to learn (*and puppetlabs.com has an extensive reference*).

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

 } <= Resource

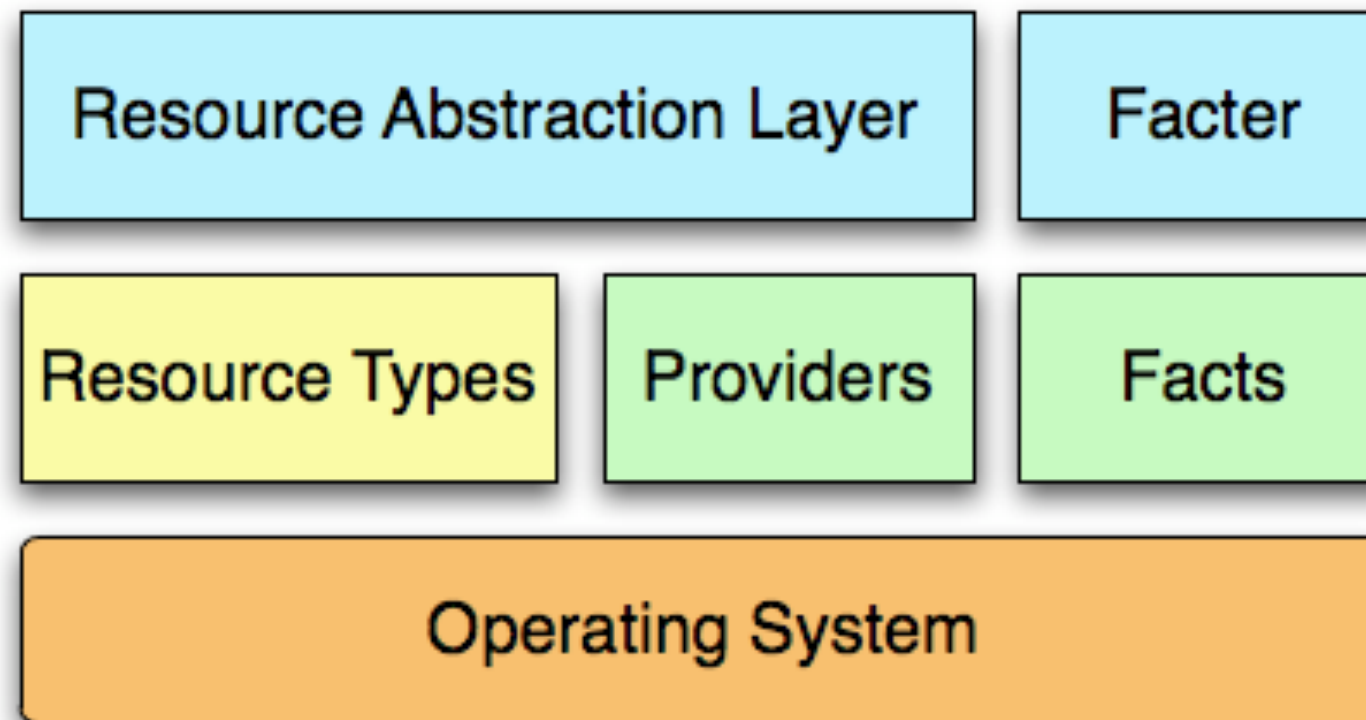
 } <= Attributes

File Attributes

backup
checksum
content
ctime
ensure
force
group
ignore
links
mode
mtime
owner
path
provider
purge
recurse
recurselimit
replace
selinux_ignore_defaults
selrange
selrole
seltype
seluser
source
sourceselect
target
type

Source: <http://docs.puppetlabs.com/references/latest/type.html#file>

Client-Side Abstraction



Classes, Manifests, and Modules

- Modules are a collection of manifests all related to a particular configuration
- Manifests are the configuration elements (consisting of classes) that utilize resources to describe the managed node
- Classes are individual code parts within manifests, related to the configured node and manage the varied resources on that node.

Let's look at each of these...

Classes

Classes are named blocks of Puppet code which are not applied unless they are invoked by name. They can be stored in modules for later use and then declared (added to a node's catalog) with the `include` function or a resource-like syntax.

Classes, simply stated, are a collection of resource declarations organized in some reasonable way to facilitate ease of use, operation, and interface. Here's an example of one of our classes (slightly modified). Classes are usually contained in manifests, which exist in modules.

```
class sudo {
  case $operatingsystem {
    OracleLinux: {
      package { sudo:
        ensure => present,
      }

      file { "/etc/sudoers":
        owner => "root",
        group => "root",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
        require => Package["sudo"],
      }
    }

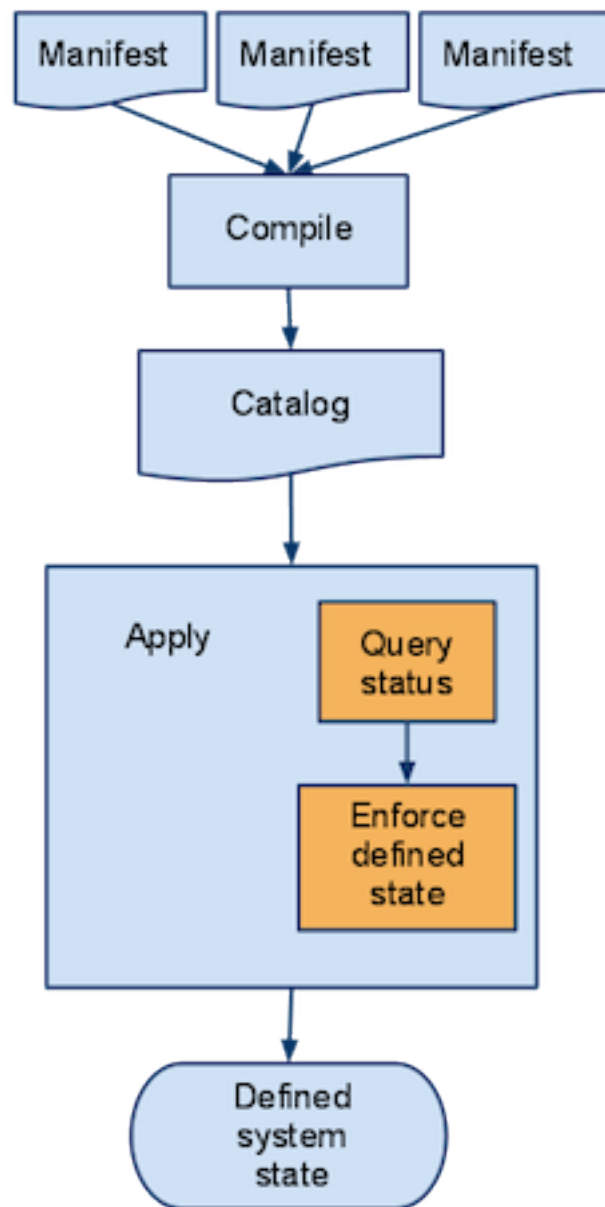
    Darwin: {
      file { "/var/gcs/sudo_provided_by_netboot":
        ensure => present,
      }

      file { "/etc/sudoers":
        owner => "root",
        group => "wheel",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
      }
    }
  }
}
```

As you can see, the more you declare, the more unwieldy the class gets!
(will solve with modules later!)

Manifests

Puppet programs are called “manifests,” and they use the .pp file extension. They can live anywhere inside of Puppet, they just need to be referenced to a node so that Puppet knows about them. The way Puppet “applies” a manifest to a node is like so:



Manifests you write

Are compiled into a catalog inside of Puppet
(i.e., we don't interact with it)

Applied to the Node

And ensures a consistent, cohesive configuration.

Which Node and Where?

```
class base {
  include cron
  include cups
  include dns
  include environment
  include ftp
  include groups
  include iptables
  include java
  include ldap
  include ntp
  include pam
  include postfix
  include puppet
  include snmp
  include sox
  include ssh
  include sudo
  include syslog
  include users
}

node 'puppet1.example.com' {
  include base
  include puppet::master
}

node 'puppet2.example.com' {
  include base
  include mysql
  include posapp
}

node 'puppet3.example.com' {
  include base
  include mysql
  include fooapp
}
```

How do we tell Puppet which node gets what configuration?

`/etc/puppet/manifests/nodes.pp`

- Manifests and modules associated with nodes in the nodes.pp file
- Lives in /etc/puppet/manifests
- This configuration has abstracted the individual manifests into a “base” grouping and a “per host” grouping.
- This is only an example. We can separate by pilot hosts vs “everything else”. Geography vs Time Zone. Your options are completely flexible

Modules

- Are not language constructs
- Are a convention for encapsulating configurations
- Necessary for autoloading of classes
- Necessary for fileserving of templates and files
- Simplify organization and maintenance

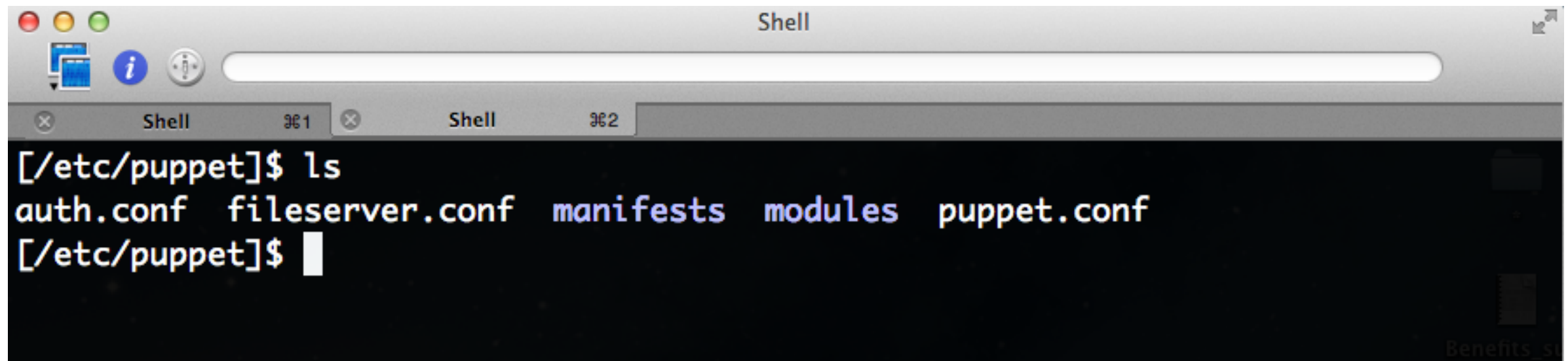
Non-Modularized Puppet Setups

- All configurations for all resources in the same location
- Becomes difficult to organize and differentiate between configuration elements
- Confuses the management and configuration of file serving
- Confuses the management of templates and custom facts
- Best for small environments with one OS and few facts/manifests

Modularized Puppet Configurations

- Organize manifests by functional units
- Compartmentalize custom facts and templates with like elements
- Enables Autoloading of classes
- Allows us to abstract at best level of granularity

“Out of the Box” Puppet Server Installations

A screenshot of a macOS terminal window titled "Shell". The terminal shows the command `ls` being executed in the directory `/etc/puppet`. The output lists five items: `auth.conf`, `fileserver.conf`, `manifests`, `modules`, and `puppet.conf`. The prompt `[/etc/puppet]$` is visible at the end of the line.

```
[/etc/puppet]$ ls
auth.conf  fileserver.conf  manifests  modules  puppet.conf
[/etc/puppet]$
```

- Lives in `/etc/puppet`
- Comes with Default Configurations
- All Manifests go in the “manifests” directory and are manually loaded in the “nodes.pp” file

Modularized Configuration

```
/etc/puppet/modules/dns
├─ files
├─ lib
├─ manifests
│   └─ init.pp
├─ templates
└─ tests
    └─ init.pp
```

- Moves manifests under their own area
- Keeps all module related configurations together
- Places all custom facts and templates with the modules they support
- Supports autoloading of classes via the “init.pp” mechanism

Module Basics

- Live in /etc/puppet/modules
- Named according to function/feature

```
[root@kopf /etc/puppet/modules]# ls
cron  environment  iptables  mysql  posapp  snmp  stdlib  users
cups  ftp          java      ntp    postfix sox   sudo
dns   groups      ldap      pam    puppet ssh   syslog
[root@kopf /etc/puppet/modules]#
```

ignore "stdlib"...more on that later)

- Main Convention for Modules Directories
 - Each module has *files*, *lib*, *manifests*, and *templates* directories
 - **files** - “blessed” file versions that Puppet distributes to hosts
 - **lib** - location for custom facter facts (ruby)
 - **manifests** - primary manifest file location for Puppet
 - **templates** - location for Template files
 - All files owned by puppet:puppet on server
 - Puppet automatically uses the “init.pp” mechanism for class loading

Manifest Basics

In the Manifests directory, the main manifest is “init.pp”. As we mentioned before, all configuration can be declared there. However, it lends clarity to divide a manifest into it’s separate parts. Let’s take our previous “sudo” example:

```
class sudo {
  case $operatingsystem {
    OracleLinux: {
      package { sudo:
        ensure => present,
      }

      file { "/etc/sudoers":
        owner => "root",
        group => "root",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
        require => Package["sudo"],
      }
    }

    Darwin: {
      file { "/var/gcs/sudo_provided_by_netboot":
        ensure => present,
      }

      file { "/etc/sudoers":
        owner => "root",
        group => "wheel",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
      }
    }
  }
}
```

```
class sudo {
  include sudo::config
  include sudo::install
}
```

init.pp

```
class sudo::config {
  case $operatingsystem {
    OracleLinux: {
      file { "/etc/sudoers":
        owner => "root",
        group => "root",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
        require => Package["sudo"],
      }
    }

    Darwin: {
      file { "/etc/sudoers":
        owner => "root",
        group => "wheel",
        mode => 0440,
        source => "puppet://$puppetserver/modules/sudo/etc_sudoers",
      }
    }
  }
}
```

config.pp

Manifest Basics (cont.)

```
class sudo::install {
  case $operatingsystem {
    OracleLinux: {
      package { sudo:
        ensure => present,
      }
    }
    Darwin: {
      file { "/var/gcs/sudo_provided_by_netboot":
        ensure => present,
      }
    }
  }
}
```

install.pp

While not terribly confusing in a single file, it helps to separate the loading from the configuration to the installation. As a manifest, manifests, classes, or sub-parts of modules expand, it's easier to just go to the relevant config. While less important for a small module like sudo, big ones benefit greatly! :

```
/etc/puppet/modules/environment/manifests
[jsheets@kopf manifests]$ ls
at.pp          etc_hosts_equiv.pp  keys.pp        security.pp
auditd.pp     etc_profile.pp      ksh.pp         skel.pp
automisc.pp   files_0400.pp       logindefs.pp   sssd.pp
bashrc.pp     files_0644.pp       motd.pp        sysstat.pp
batch.pp      files_0700.pp       ngs.pp         ttys.pp
control_file.pp gcs.pp              nis.pp         yum_repo.pp
cshlogin.pp  grub.pp             nscd.pp        zeroconf.pp
cshlogout.pp haldaemon.pp        ossec.pp       zsh.pp
cshrc.pp     init.pp             rootbashprofile.pp
dirs_0755.pp  ipv6.pp             root_profile.pp
dirs_1777.pp  issue.pp            rpc.pp
[jsheets@kopf manifests]$
```

Factor

- is an independent, cross-platform Ruby library designed to gather information on all the nodes you will be managing with Puppet. It is available on all platforms that Puppet is available.

- is a lightweight program that gathers basic node information about the hardware and operating system. Factor is especially useful for retrieving things like operating system names, hardware characteristics, IP addresses, MAC addresses, and SSH keys.

```
[jsheets@puppet ~]$ factor -p  
architecture => x86_64  
augeasversion => 0.9.0  
domain => example.com  
facterversion => 1.6.6  
fqdn => puppet.example.com  
hardwareisa => x86_64  
hardwaremodel => x86_64  
hostname => puppet  
id => jsheets  
interfaces => eth0,lo  
ipaddress => 1.2.3.4  
ipaddress_eth0 => 1.2.3.4  
ipaddress_lo => 127.0.0.1  
is_virtual => true  
kernel => Linux
```

```
[jsheets@puppet ~]$ factor -p ipaddress  
1.2.3.4
```

```
[jsheets@puppet ~]$ factor -p operatingsystem  
OracleLinux
```

```
pupeptclient:~ admin$ factor -p sp_cpu_type  
Intel Core 2 Duo
```

Advantages of Facter

- We can now write scripts to facter queries identically across platforms
- This provides a uniform API for system information on a box (no more crazy regex or chained awks)
- If we have a custom piece of information we need, we can extend facter to present that information in the same consistent way:

```
root@puppet:/root# facter -p router1  
1.2.3.1
```

```
root@puppet:/root# facter -p router2  
1.2.3.2
```

```
root@puppet:/root# facter -p custom1  
fooapp
```

```
root@puppet:/root# facter -p custom2  
dev
```

Custom Facts and Templates

Ex. from our “dns” module:

/etc/puppet/modules/dns/lib/facter/router.rb

```
# Custom Fact that figures out the first Router Address from
# another custom fact I wrote to get the network section of the
# IP space.
#
Facter.add("router1") do
  setcode do
    %x{facter -p prefix |usr/bin/awk -F "." '{print $1"."$2"."($3+1)".4}'}.chomp
  end
end
```

and the template in /etc/puppet/modules/dns/templates/resolv-dyn.erb

```
# This file managed by Puppet. DO NOT EDIT.
# auto-generated via custom facts & a Ruby template
#
search example.com foo.example.com bar.example.com
domain example.com
nameserver <%= router1 %>
nameserver <%= router2 %>
nameserver 1.2.1.2
```

As you can see... A little bit of Ruby and templating, and we can accommodate *any* situation in our environment.

Yeah, but what about Perl?

Use Cases

Provide a Perl program via “file” resource:

```
file { “/foo/bar/baz.pl”:  
  ensure => present,  
  owner => “foo”,  
  group => “bar”,  
  mode => 0755,  
  source => “puppet://$puppetserver/modules/foo/baz.pl”,  
}
```

And then run it via “exec” with a requirement of its existence:

```
exec { “runbaz”:  
  command => “/foo/bar/baz.pl”,  
  path => “/foo/bar”,  
  creates => “/foo/bar/baz.report”,  
  require => File[“/foo/bar/baz.pl”],  
}
```

How about conditional Execution?

Within your Perl program:

```
my($foo) = `sudo /usr/bin/facter -p <value>`;
my(@foo) = `sudo /usr/bin/facter -p`;
```

...and then use the value from facter as a conditional, or execute different Perl scripts based on some facter variable:

```
case $operatingsystem {
  CentOS: {
    exec { "foo":....
  }
}
Solaris: {
  exec { "bar":....
}
}
```

You can also grab ALL of facter's output and pump that into a hash, a json or XML element, and iterate over those values as you like.

How about a CPAN-ish way?

Sys::Facter can do the trick:

```
use Sys::Facter;  
use Data::Dumper;
```

```
my $facter = new Sys::Facter(modules => [ /foo/bar"]);
```

Grab some facts and print them:

```
$facter->load("kernel", "operatingsystem", "ipaddress");  
print Dumper $facter->facts;
```

```
# Or just print some facts directly  
print $facter->kernel;  
print $facter->operatingsystem;
```


How do I learn this thing?

- **Version 3 Reference**

- <http://docs.puppetlabs.com/puppet/3/reference/>
- <http://docs.puppetlabs.com/learning/>

- **Puppet Function Reference**

- <http://docs.puppetlabs.com/references/latest/function.html>

- **Puppet Type Reference**

- <http://docs.puppetlabs.com/references/latest/type.html>

Help & Community

- Books!

- “Pro Puppet”
- “Puppet Cookbook”
- “Pulling Strings With Puppet”

- Mailing Lists!

- “Puppet Users”
- “Puppet Developers”

- IRC

- <http://webchat.freenode.net/?channels=puppet>
- <http://webchat.freenode.net/?channels=puppet-dev>

Your Own Test Environment

- The Puppet Virtual Machine
 - <http://info.puppetlabs.com/download-learning-puppet-VM.html>
 - VMX and OVF Format
 - Made to be used with the Puppet Training materials:
 - <http://docs.puppetlabs.com/learning/>
 - Great with the slide deck I have from Puppet Training