

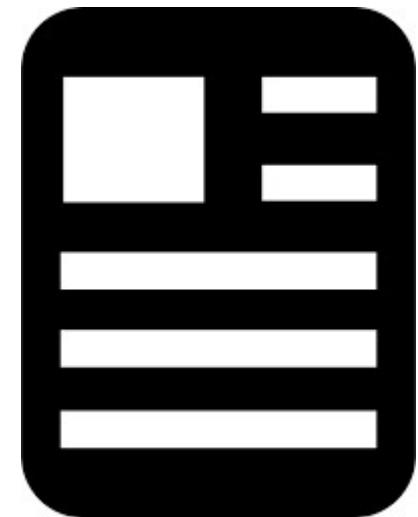
# Puppet





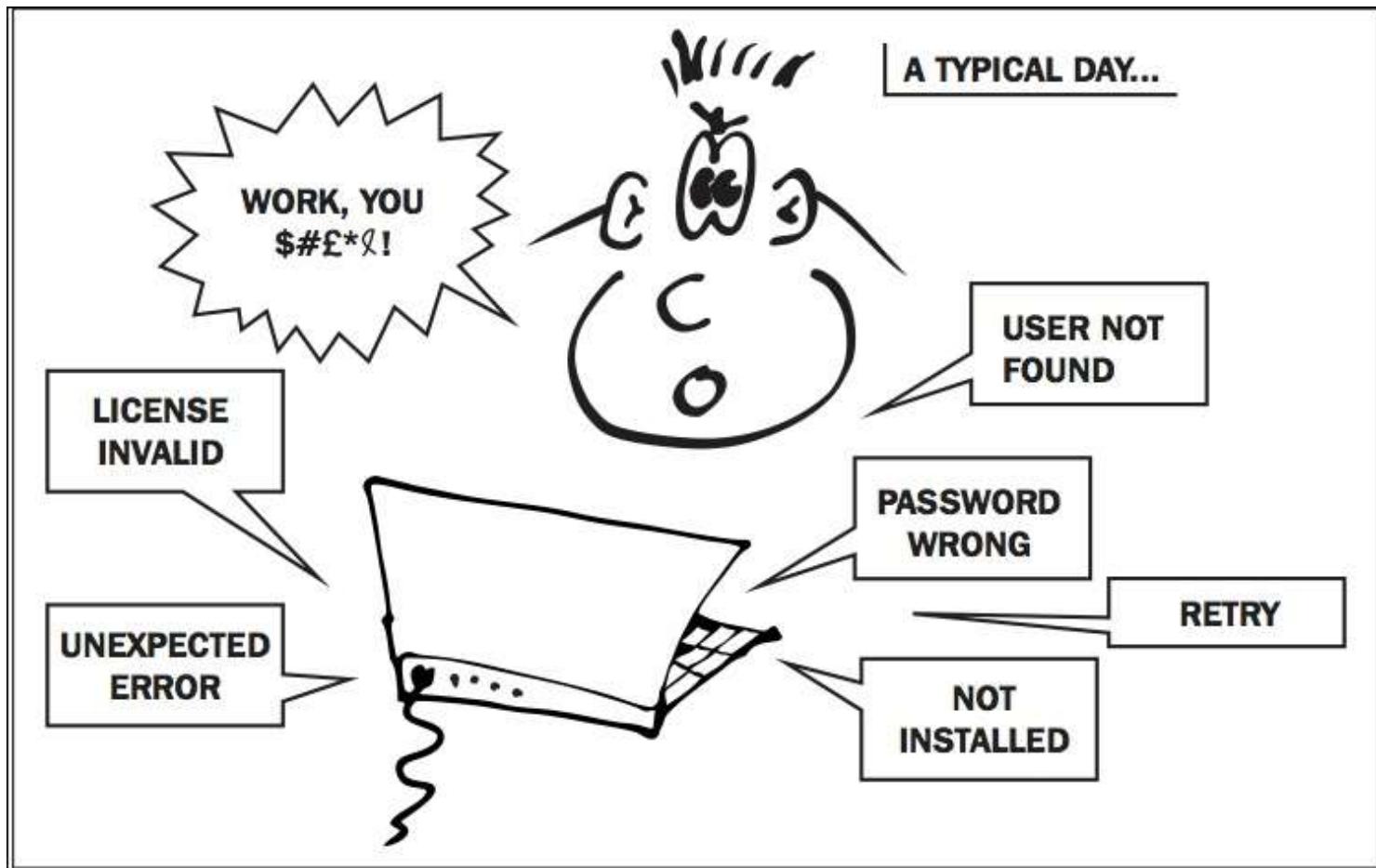
# Table of Contents

1. Getting started with Puppet: 3
2. Creating your first manifests: 26
3. Managing your Puppet code with Git: 52
4. Understanding Puppet resources: 83
5. Variables, expressions, and facts: 136
6. Managing data with Hiera: 178
7. Mastering modules: 247
8. Classes, roles, and profiles: 308
9. Managing files with templates: 342



# 1. Getting started with Puppet





# Why do we need Puppet anyway?

- Add user accounts and passwords
- Configure security settings and privileges
- Install all the packages needed to run the app
- Customize the configuration files for each of these packages
- Create databases and database user accounts; load some initial data



# Keeping the configuration synchronized

- Setting up servers manually is tedious.
- Even if you're the kind of person who enjoys tedium, though, there's another problem to consider. What happens the next time you set up a server, a few weeks or months later?
- Your careful notes will no longer be up to date with reality.
- While you were on vacation, the developers installed a couple of new libraries that the app now depends on

# Repeating changes across many servers

- Humans just aren't good at accurately repeating complex tasks over and over; that's why we invented robots.
- It's easy to make mistakes, miss things out, or be interrupted and lose track of what you've done.
- Changes happen all the time, and it becomes increasingly difficult to keep things up to date and in sync as your infrastructure grows.

# Self-updating documentation

- In real life, we're too busy to stop every five minutes and document what we just did.
- As we've seen, that documentation is of limited use anyway, even if it's kept fanatically up-to-date.
- The only reliable documentation, in fact, is the state of the servers themselves.



# Version control and history

- When you're making manual, ad hoc changes to systems, you can't roll them back to a point in time.
- It's hard to undo a whole series of changes; you don't have a way of keeping track of what you did and how things changed.
- This is bad enough when there's just one of you.



# Why not just write shell scripts?

- Fragile and non-portable
- Hard to maintain
- Not easy to read as documentation
- Very site-specific
- Not a good programming language
- Hard to apply changes to existing servers



# Why not just use containers?

- Containers! Is there any word more thrilling to the human soul? Many people feel as though containers are going to make configuration management problems just go away.
- This feeling rarely lasts beyond the first few hours of trying to containerize an app.
- Yes, containers make it easy to deploy and manage software, but where do containers come from?

# Why not just use serverless?



- If containers are powered by magic pixies, serverless architectures are pure fairy dust.
- The promise is that you just push your app to the cloud, and the cloud takes care of deploying, scaling, load balancing, monitoring, and so forth.
- Like most things, the reality doesn't quite live up to the marketing.

# Configuration management tools



- Configuration management (CM) tools are the modern, sensible way to manage infrastructure as code.
- There are many such tools available, all of which operate more or less the same way
- you specify your desired configuration state, using editable text files and a model of the system's resources, and the tool compares the current state of each node

# What is Puppet?

- What does this language look like? It's not exactly a series of instructions, like a shell script or a Ruby program.
- It's more like a set of declarations about the way things should be.
- Have a look at the following example:

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



# What is Puppet?

- Here's another example of Puppet code:

```
user { 'bridget':  
    ensure => present,  
}
```



# Resources and attributes

- Puppet lets you describe configuration in terms of resources (types of things that can exist, such as users, files, or packages) and their attributes (appropriate properties for the type of resource, such as the home directory for a user, or the owner and permissions for a file).
- You don't have to get into the details of how resources are created and configured on different platforms.

# Puppet architectures



- It's worth noting that there are two different ways to use Puppet.
- The first way, known as agent/master architecture, uses a special node dedicated to running Puppet, which all other nodes contact to get their configuration.
- The other way, known as stand-alone Puppet or masterless, does not need a special Puppet master node.

# Getting ready for Puppet



- Although Puppet is inherently cross-platform and works with many different operating systems, for the purposes of this course
- I'm going to focus on just one operating system, namely the Ubuntu 16.04 LTS distribution of Linux, and the most recent version of Puppet, Puppet 5.
- However, all the examples in the course should work on any recent operating system or Puppet version with only minor changes.

# Installing Git and downloading the repo

- Browse to <https://git-scm.com/downloads>
- Download and install the right version of Git for your operating system.
- Run the following command:  
`git clone https://github.com/fenago/puppet-beginners-guide-3.git`



# Installing VirtualBox and Vagrant

- Browse to <https://www.virtualbox.org/>
- Download and install the right version of VirtualBox for your operating system
- Browse to <https://www.vagrantup.com/downloads.html>
- Select the right version of Vagrant for your operating system: OS X, Windows, and so on
- Follow the instructions to install the software



# Running your Vagrant VM

Once you have installed Vagrant, you can start the Puppet Beginner's Guide virtual machine:

- Run the following commands:

```
cd puppet-beginners-guide-3  
scripts/start_vagrant.sh
```



# Running your Vagrant VM

- Connect to the VM with the following command:

`vagrant ssh`

- You now have a command-line shell on the VM.
- Check that Puppet is installed and working by running the following command (you may get a different version number, which is fine):

`puppet --version`  
5.2.0





# Troubleshooting Vagrant

- If you have any problems running the VM, look for help on the VirtualBox or Vagrant websites.
- In particular, if you have an older machine, you may see a message like the following:

VT-x/AMD-V hardware acceleration is not available on your system. Your 64-bit guest will fail to detect a 64-bit CPU and will not be able to boot.

# Troubleshooting Vagrant

- If not, you can try the 32-bit version of the Vagrant box instead.
- Edit the file named Vagrantfile in the Git repository, and comment out the following line with a leading # character:

```
config.vm.box = "ubuntu/xenial64"
```

- Uncomment the following line by removing the leading # character:

```
# config.vm.box = "ubuntu/xenial32"
```



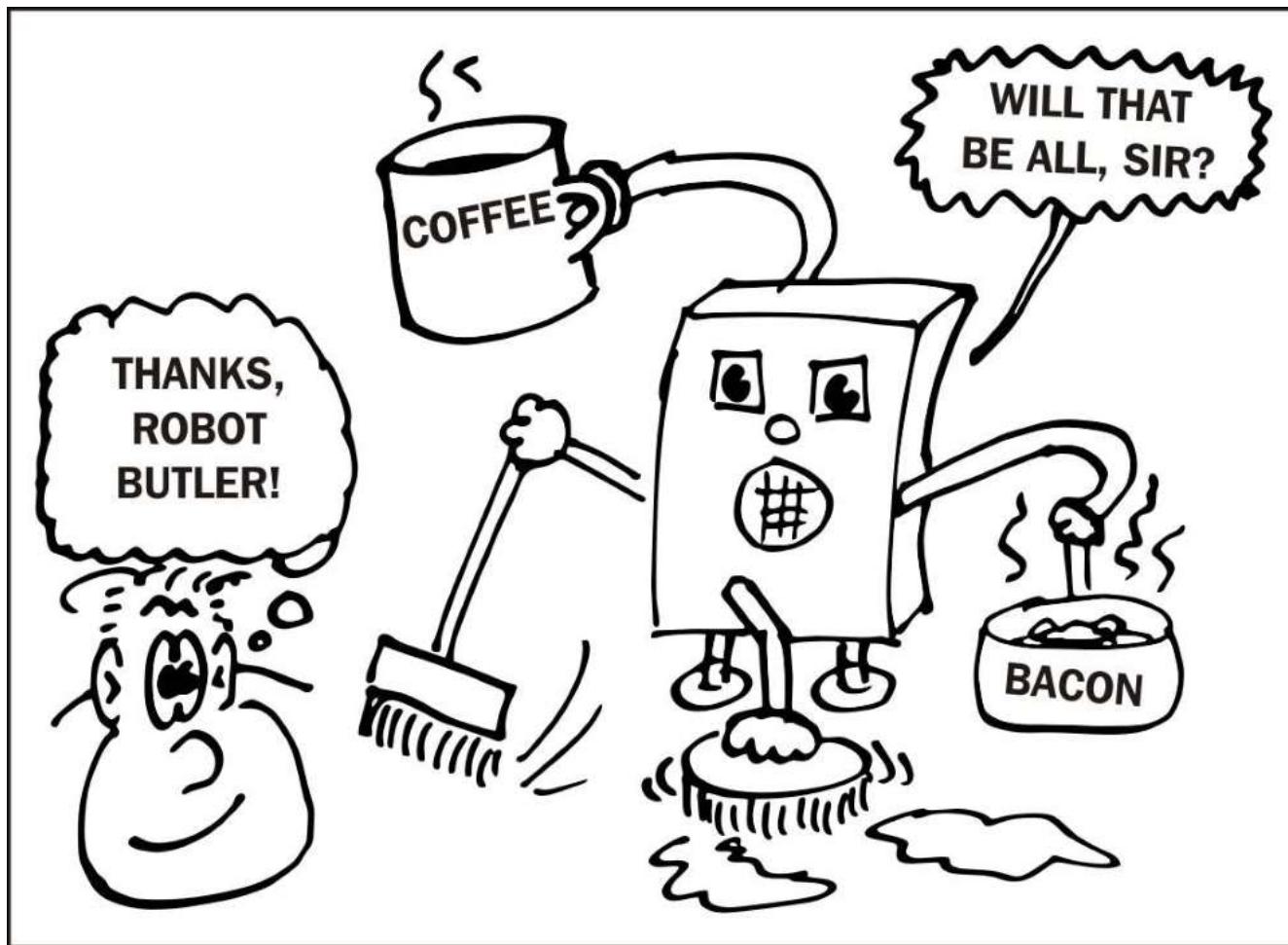
# Summary

- In this lesson, we looked at the various problems that configuration management tools can help solve, and how Puppet in particular models the aspects of system configuration.
- We checked out the Git repository of example code for this course, installed VirtualBox and Vagrant, started the Vagrant VM, and ran Puppet for the first time.



## 2. Creating your first manifests





# Hello, Puppet – your first Puppet manifest

- On your Vagrant box, run the following command:

```
sudo puppet apply /examples/file_hello.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in  
environment production in 0.07 seconds
```

```
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/ensure:  
defined content as
```

```
'{md5}22c3683b094136c3398391ae71b20f04'
```

```
Notice: Applied catalog in 0.01 seconds
```



# Hello, Puppet – your first Puppet manifest

- We can ignore the output from Puppet for the moment, but if all has gone well, we should be able to run the following command:

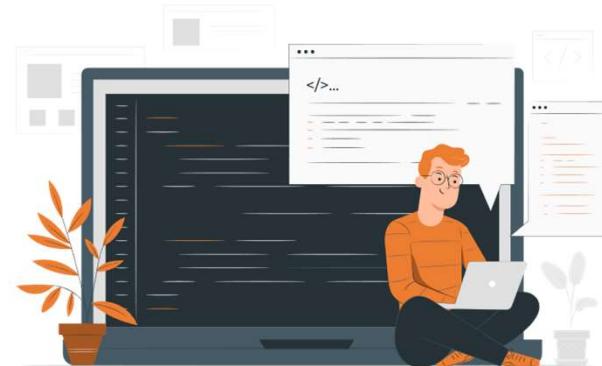
```
cat /tmp/hello.txt  
hello, world
```



# Understanding the code

- Let's look at the example code to see what's going on (run `cat /example/file_hello.pp`, or open the file in a text editor):

```
file { '/tmp/hello.txt':  
  ensure => file,  
  content => "hello, world\n",  
}
```



# Understanding the code

- The code term file begins a resource declaration for a file resource.
- A resource is some bit of configuration that you want Puppet to manage: for example, a file, user account, or package.
- A resource declaration follows this pattern:

```
RESOURCE_TYPE { TITLE:  
  ATTRIBUTE => VALUE,  
  ...  
}
```



# Understanding the code

- Again, the possible values for ensure are specific to the type of resource.
- In this case, we use file to indicate that we want a regular file, as opposed to a directory or symlink:  
`ensure => file,`
- Next, to put some text in the file, we specify the content attribute:  
`content => "hello, world\n",`



# Modifying existing files

- What happens if the file already exists when Puppet runs and it contains something else? Will Puppet change it?

```
sudo sh -c 'echo "goodbye, world" >/tmp/hello.txt'
```

```
cat /tmp/hello.txt
```

```
goodbye, world
```

```
sudo puppet apply /examples/file_hello.pp
```

```
cat /tmp/hello.txt
```

```
hello, world
```



# Modifying existing files

- So it's a good idea to add a comment to files that Puppet is managing: something like the following:

```
# This file is managed by Puppet - any manual edits will  
be lost
```



# Dry-running Puppet

- Adding the --noop flag to puppet apply will show you what Puppet would have done, without actually changing anything:

```
sudo sh -c 'echo "goodbye, world" >/tmp/hello.txt'  
sudo puppet apply --noop /examples/file_hello.pp  
Notice: Compiled catalog for ubuntu-xenial in environment  
production in 0.04 seconds  
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/content:  
current_value {md5}7678..., should be {md5}22c3... (noop)
```

# Dry-running Puppet

- If you want to see what change Puppet would actually make to the file, you can use the `--show_diff` option:

```
sudo puppet apply --noop --show_diff /examples/file_hello.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in environment  
production in 0.04 seconds
```

```
Notice: /Stage[main]/Main/File[/tmp/hello.txt]/content:
```

```
--- /tmp/hello.txt    2017-02-13 02:27:13.186261355 -0800
```

```
+++ /tmp/puppet-file20170213-3671-2yynjt    2017-02-13  
02:30:26.561834755 -0800
```

```
@@@ -1 +1 @@@
```

```
-goodbye, world
```

```
+hello, world
```



# Creating a file of your own

- Create your own manifest file (you can name it anything you like, so long as the file extension is .pp).
- Use a file resource to create a file on the server with any contents you like.
- Apply the manifest with Puppet and check that the file is created and contains the text you specified.

# Managing packages

- Puppet will use the appropriate package manager commands to install it on whatever platform it's running on.
- As you've seen, all resource declarations in Puppet follow this form:

```
RESOURCE_TYPE { TITLE:  
  ATTRIBUTE => VALUE,  
  ...  
}
```



# Managing packages

- package resources are no different.
- The RESOURCE\_TYPE is package, and the only attribute you usually need to specify is ensure, and the only value it usually needs to take is installed:

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



# Managing packages

- Try this example:

```
sudo puppet apply /examples/package.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in  
environment production in 0.52 seconds
```

```
Notice: /Stage[main]/Main/Package[cowsay]/ensure:  
created
```

```
Notice: Applied catalog in 29.53 seconds
```

# Managing packages

- Let's see whether cowsay is installed:

cowsay Puppet rules!

---

< Puppet rules! >

---

\ ^ ^  
 \ (oo)\\_\_\_\_\_  
 (\_\_\_\_)\ ) \|/\  
 ||----w |  
 || ||



# How Puppet applies the manifest

- The title of the package resource is `cowsay`, so Puppet knows that we're talking about a package named `cowsay`.
- The `ensure` attribute governs the installation state of packages: unsurprisingly, `installed` tells Puppet that the package should be installed.



# Exercise

- Create a manifest that uses the package resource to install any software you find useful for managing servers.
- Here are some suggestions: tmux, sysdig, atop, htop, and dstat.



# Querying resources with the puppet resource

- If you want to see what version of a package Puppet thinks you have installed, you can use the `puppet resource` tool:

```
puppet resource package openssl  
package { 'openssl':  
  ensure => '1.0.2g-1ubuntu4.8',  
}
```



# Services

- Puppet models services with the service resource type.
- The service resources look like the following example (you can find this in service.pp in the /examples/ directory).
- From now on, I'll just give the filename of each example, as they are all in the same directory):

```
service { 'sshd':  
    ensure => running,  
    enable => true,  
}
```



# Getting help on resources with puppet describe

- If you're struggling to remember all the different attributes of all the different resources, Puppet has a built-in help feature that will remind you.
- Run the following command, for example:

puppet describe service

# Getting help on resources with puppet describe

- To see a list of all the available resource types, run the following command:

puppet describe --list



# The package-file-service pattern

- It's very common for a given piece of software to require these three Puppet resource types
- the package resource installs the software, the file resource deploys one or more configuration files required for the software, and the service resource runs the software itself.
- Here's an example using the MySQL database server (package\_file\_service.pp):

Refer to the file 2\_1.txt

# Notifying a linked resource

- You might have noticed a new attribute, called `notify`, in the `file` resource in the previous example:

```
file { '/etc/mysql/mysql.cnf':  
    source => '/examples/files/mysql.cnf',  
    notify => Service['mysql'],  
}
```



# Resource ordering with require

- All resources support the require attribute, and its value is the name of another resource declared somewhere in the manifest, specified in the same way as when using notify.
- Here's the package-file-service example again, this time with the resource ordering specified explicitly using require (package\_file\_service\_require.pp):

Refer to the file 2\_2.txt

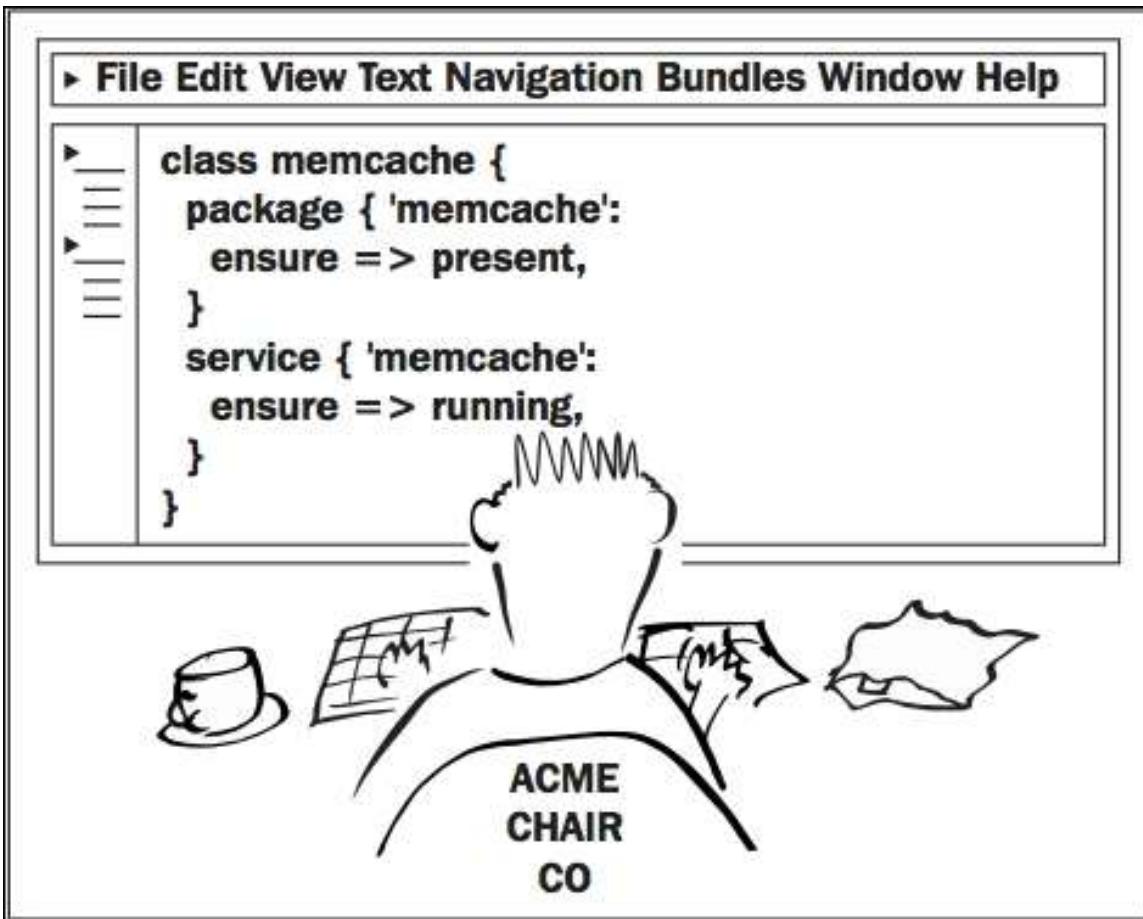
# Summary

- In this lesson, we've seen how a manifest is made up of Puppet resources.
- You've learned how to use Puppet's file resource to create and modify files, how to install packages using the package resource and How to manage services with the service resource.
- We've looked at the common package-file-service pattern and seen how to use the notify attribute on a resource to send a message to another resource indicating that its configuration has been updated.



# 3. Managing your Puppet code with Git





# What is version control?

- If you're already familiar with Git, you can save some reading by skipping ahead to the Creating a Git repo section. If not, here's a gentle introduction.
- Even if you're the only person who works on a piece of source code (for example, Puppet manifests), it's still useful to be able to see what changes you made, and when.

# Tracking changes

- When you're working on code with others, you also need a way to communicate with the rest of the team about your changes.
- A version control tool such as Git not only tracks everyone's changes, but lets you record a commit message, explaining what you did and why.
- The following example illustrates some aspects of a good commit message:

Refer to the file 3\_1.txt



# Sharing code

- A set of files under Git version control is called a repository, which is usually equivalent to a project.
- A Git repository (from now on, just repo) is also a great way to distribute your code to others, whether privately or publicly, so that they can use it, modify it, contribute changes back to you, or develop it in a different direction for their own requirements.

# Creating a Git repo

- Make a directory to hold your versioned files using the following commands:

cd

mkdir puppet



- Now run the following commands to turn the directory into a Git repo:

cd puppet

git init

Initialized empty Git repository in /home/ubuntu/puppet/.git/

# Making your first commit

- Because Git records not only changes to the code, but also who made them, it needs to know who you are.
- Set your identification details for Git (use your own name and email address, unless you particularly prefer mine) using the following commands:

```
git config --global user.name "John Arundel"
```

```
git config --global user.email john@fenagoconsulting.com
```

# Making your first commit

- It's traditional for Git repos to have a README file, which explains what's in the repo and how to use it.
- For the moment, let's just create this file with a placeholder message:

```
echo "Watch this space... coming soon!" > README.md
```

# Making your first commit

- Run the following command:

git status

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

  README.md

nothing added to commit but untracked files present (use "git add" to track)



# Making your first commit

- Because we've added a new file to the repo, changes to it won't be tracked by Git unless we explicitly tell it to.
- We do this by using the git add command, as follows:

git add README.md

```
# Amend the last commit message.  
# Push the changes to remote by force.  
# USAGE: gamend "Your New Commit Msg"  
function gamend() {  
    git commit --amend -m "$@"  
    git push --force-with-lease  
}
```

# Making your first commit

- Git now knows about this file, and changes to it will be included in the next commit.
- We can check this by running git status again:

git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README.md



# Making your first commit

- The file is listed under Changes to be committed, so we can now actually make the commit:

```
git commit -m 'Add README file'  
[master (root-commit) ee21595] Add README file  
 1 file changed, 1 insertion(+)  
 create mode 100644 README.md
```

# Making your first commit

- You can always see the complete history of commits in a repo by using the git log command.
- Try it now to see the commit you just made:

git log

commit ee215951199158ef28dd78197d8fa9ff078b3579

Author: John Arundel <john@fenagoconsulting.com>

Date: Tue Aug 30 05:59:42 2016 -0700

Add README file



# How often should I commit?

- A common practice is to commit when the code is in a consistent, working state, and have the commit include a set of related changes made for some particular purpose.
- So, for example, if you are working to fix bug number 75 in your issue-tracking system, you might make changes to quite a few separate files and then, once you're happy the work is complete, make a single commit with a message such as:

Make nginx restart more reliable (fixes issue #75)

# Branching

- Git has a powerful feature called branching, which lets you create a parallel copy of the code (a branch) and make changes to it independently.
- At any time, you can choose to merge those changes back into the master branch.
- Or, if changes have been made to the master branch in the meantime, you can incorporate those into your working branch and carry on.

# Distributing Puppet manifests

- There are several ways to do this, and as we saw in lesson 1, Getting started with Puppet, one approach is to use the agent/master architecture, where a central Puppet master server compiles your manifests and distributes the catalog (the desired node state) to all nodes.
- Another way to use Puppet is to do without the master server altogether, and use Git to distribute manifests to client nodes, which then runs puppet apply to update their configuration.

# Creating a GitHub account and project

If you already have a GitHub account, or you're using another Git server, you can skip this section.

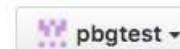


- Browse to <https://github.com/>
- Enter the username you want to use, your email address, and a password.
- Choose the Unlimited public repositories for free plan.
- GitHub will send you an email to verify your email address.

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner



Repository name

pbgtest / puppet ✓

Great repository names are short and memorable. Need inspiration? How about **special-palm-tree**.

Description (optional)

My amazing Puppet repo. I owe it all to @bitfield.



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Add a license: **None** ▾



**Create repository**

# Pushing your repo to GitHub

- In your repo directory, run the following commands.  
After git remote add origin, specify the URL to your GitHub repo:

```
git remote add origin YOUR_REPO_URL  
git push -u origin master
```



# Pushing your repo to GitHub

- GitHub will prompt you for your username and password:

Username for 'https://github.com': pbgtest

Password for 'https://pbgtest@github.com':

Counting objects: 3, done.

Writing objects: 100% (3/3), 262 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/pbgtest/puppet.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.



 pbgtest / puppet

[Watch](#) 0   [Star](#) 0   [Fork](#) 0

[Code](#)   [Issues 0](#)   [Pull requests 0](#)   [Wiki](#)   [Pulse](#)   [Graphs](#)   [Settings](#)

My amazing Puppet repo. I owe it all to [@bitfield](#). — [Edit](#)

 1 commit    1 branch    0 releases    1 contributor

Branch: master ▾   [New pull request](#)   [Create new file](#)   [Upload files](#)   [Find file](#)   [Clone or download](#) ▾

 [bitfield](#) Add README file   Latest commit ee21595 10 days ago

 [README.md](#)   Add README file   10 days ago

 [README.md](#)

Watch this space... coming soon!

# Cloning the repo

- Run the following commands (replace the argument to git clone with the URL of your own GitHub repo, but don't lose the production at the end):

```
cd /etc/puppetlabs/code/environments  
sudo mv production production.sample  
sudo git clone YOUR_REPO_URL production  
Cloning into 'production'...  
remote: Counting objects: 3, done.  
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), done.  
Checking connectivity... done.
```



# Fetching and applying changes automatically

- In a stand-alone Puppet architecture, each node needs to automatically fetch any changes from the Git repo at regular intervals, and apply them with Puppet.
- We can use a simple shell script for this, and there's one in the example repo (/examples/files/run-puppet.sh):

```
#!/bin/bash
cd /etc/puppetlabs/code/environments/production && git
pull
/opt/puppetlabs/bin/puppet apply manifests/
```

# Writing a manifest to set up regular Puppet runs

- Run the following commands to create the required directories in your Puppet repo:

```
cd /home/ubuntu/puppet  
mkdir manifests files
```



# Writing a manifest to set up regular Puppet runs

- Run the following command to copy the run-puppet script from the examples/ directory:

```
cp /examples/files/run-puppet.sh files/
```

- Run the following command to copy the run-puppet manifest from the examples/ directory:

```
cp /ubuntu/examples/run-puppet.pp manifests/
```

# Writing a manifest to set up regular Puppet runs

- Add and commit the files to Git with the following commands:

```
git add manifests files
```

```
git commit -m 'Add run-puppet script and cron job'
```

```
git push origin master
```



# Applying the run-puppet manifest

- Having created and pushed the manifest necessary to set up automatic Puppet runs, we now need to pull and apply it on the target node.
- In the cloned copy of your repo in /etc/puppetlabs/code/environments/production, run the following commands:

Refer to the file 3\_2.txt



# The run-puppet script

The run-puppet script does the following two things in order to automatically update the target node:

- Pull any changes from the Git server (git pull).
- Apply the manifest (puppet apply).



# Testing automatic Puppet runs

- To troubleshoot the problem, try running sudo run-puppet manually.
- If this works, check that the cron job is correctly installed by running sudo crontab -l. It should look something like the following:

Refer to the file 3\_3.txt





# Managing multiple nodes

- Install Puppet (not necessary if you're using the Vagrant box).
- Clone your Git repo (as described in the Cloning the repo section).
- Apply the manifest (as described in the Applying the run-puppet manifest section).

# Summary

- In this lesson, we introduced the concepts of version control, and the essentials of Git in particular.
- We set up a new Git repo, created a GitHub account, pushed our code to it, and cloned it on a node.
- We wrote a shell script to automatically pull and apply changes from the GitHub repo on any node, and a Puppet manifest to install this script and run it regularly from cron.



# 4. Understanding Puppet resources



# Understanding Puppet resources



# Files

- We saw in lesson 2, Creating your first manifests that Puppet can manage files on a node using the file resource, and we looked at an example which sets the contents of a file to a particular string using the content attribute.
- Here it is again (file\_hello.pp):

```
file { '/tmp/hello.txt':  
    content => "hello, world\n",  
}
```



# Files

# The path attribute

- We've seen that every Puppet resource has a title (a quoted string followed by a colon). In the `file_hello` example, the title of the file resource is `'/tmp/hello.txt'`.
- It's easy to guess that Puppet is going to use this value as the path of the created file.

# Managing whole files

- Ideally, we would put a copy of the file in the Puppet repo, and have Puppet simply copy it to the desired place in the filesystem.
- The source attribute does exactly that (`file_source.pp`):

```
file { '/etc/motd':  
    source => '/examples/files/motd.txt',  
}
```

# Managing whole files

- To try this example with your Vagrant box, run the following commands:

```
sudo puppet apply /examples/file_source.pp  
cat /etc/motd
```

The best software in the world only sucks. The worst software is significantly worse than that.

-Luke Kanies



# Managing whole files

- The value of the source attribute can be a path to a file on the node, as here, or an HTTP URL, as in the following example (`file_http.pp`):

```
file { '/tmp/README.md':  
    source =>  
    'https://raw.githubusercontent.com/puppetlabs/puppet/m  
aster/README.md',  
}
```

# Ownership

- On Unix-like systems, files are associated with an owner, a group, and a set of permissions to read, write, or execute the file.
- Since we normally run Puppet with the permissions of the root user (via sudo), the files Puppet manages will be owned by that user:

ls -l /etc/motd

```
-rw-r--r-- 1 root root 109 Aug 31 04:03 /etc/motd
```



# Ownership

- Often, this is just fine, but if we need the file to belong to another user (for example, if that user needs to be able to write to the file), we can express this by setting the owner attribute (file\_owner.pp):

```
file { '/etc/owned_by_ubuntu':  
    ensure => present,  
    owner  => 'ubuntu',  
}  
  
ls -l /etc/owned_by_ubuntu  
-rw-r--r-- 1 ubuntu root 0 Aug 31 04:48  
/etc/owned_by_ubuntu
```



# Ownership

- You can see that Puppet has created the file and its owner has been set to ubuntu.
- You can also set the group ownership of the file using the group attribute (file\_group.pp):

```
file { '/etc/owned_by_ubuntu':  
    ensure => present,  
    owner  => 'ubuntu',  
    group  => 'ubuntu',  
}  
ls -l /etc/owned_by_ubuntu  
-rw-r--r-- 1 ubuntu ubuntu 0 Aug 31 04:48 /etc/owned_by_ubuntu
```

# Permissions

- In the following example, we use the mode attribute to set a mode of 0644 ("read and write for the owner, read-only for the group, and read-only for other users") on a file (file\_mode.pp):

```
file { '/etc/owned_by_ubuntu':  
    ensure => present,  
    owner  => 'ubuntu',  
    mode   => '0644',  
}
```



# Directories

- Creating or managing permissions on a directory is a common task, and Puppet uses the file resource to do this too. If the value of the ensure attribute is directory, the file will be a directory (file\_directory.pp):

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



# Trees of files

- We've already seen that Puppet can copy a single file to the node, but what about a whole directory of files, possibly including subdirectories (known as a file tree)?
- The recurse attribute will take care of this (file\_tree.pp):

```
file { '/etc/config_dir':
```

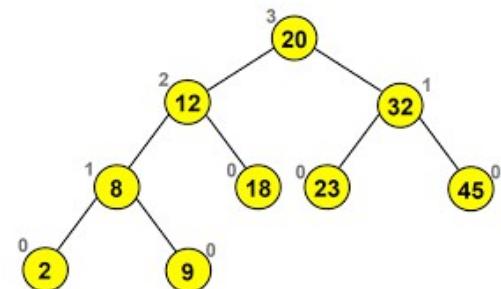
```
  source => '/examples/files/config_dir',
```

```
  recurse => true,
```

```
}
```

```
ls /etc/config_dir/
```

```
1 2 3
```



# Symbolic links

- You can have Puppet do this by setting ensure => link on the file resource and specifying the target attribute (file\_symlink.pp):

```
file { '/etc/this_is_a_link':  
    ensure => link,  
    target => '/etc/motd',  
}
```

```
ls -l /etc/this_is_a_link  
lrwxrwxrwx 1 root root 9 Aug 31 05:05  
/etc/this_is_a_link -> /etc/motd
```



# Uninstalling packages

- The ensure attribute normally takes the value installed in order to install a package, but if you specify absent instead, Puppet will remove the package if it happens to be installed.
- Otherwise, it will take no action.
- The following example will remove the apparmor package if it's installed (package\_remove.pp):

```
package { 'apparmor':  
    ensure => absent,  
}
```



# Installing specific versions

- If there are multiple versions of a package available to the system's package manager, specifying ensure => installed will cause Puppet to install the default version (usually the latest).
- But, if you need a specific version, you can specify that version string as the value of ensure, and Puppet will install that version (package\_version.pp):

```
package { 'openssl':  
    ensure => '1.0.2g-1ubuntu4.8',  
}
```

# Installing the latest version

- On the other hand, if you specify ensure => latest for a package, Puppet will make sure that the latest available version is installed every time the manifest is applied.
- When a new version of the package becomes available, it will be installed automatically on the next Puppet run.

# Installing Ruby gems

- Puppet can install Ruby gems for you using the provider => gem attribute (package\_gem.pp):

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

```
package { 'puppet-lint':  
    ensure  => installed,  
    provider => gem,  
}
```



# Installing Ruby gems

- The puppet-lint tool, by the way, is a good thing to have installed.
- It will check your Puppet manifests for common style errors and make sure they comply with the official Puppet style guide.
- Try it now:

```
puppet-lint /examples/lint_test.pp
```

WARNING: indentation of => is not properly aligned  
(expected in column 11, but found it in column 10) on  
line 2

# Installing Ruby gems

- In this example, puppet-lint is warning you that the => arrows are not lined up vertically, which the style guide says they should be:

```
file { '/tmp/lint.txt':  
    ensure => file,  
    content => "puppet-lint is your friend\n",  
}
```



# Installing gems in Puppet's context

- Puppet itself is written at least partly in Ruby, and makes use of several Ruby gems.
- To avoid any conflicts with the version of Ruby and gems which the node might need for other applications, Puppet packages its own version of Ruby and associated gems under the /opt/puppetlabs/ directory.

# Installing gems in Puppet's context

- The following example demonstrates how to use this provider (package\_puppet\_gem.pp):

```
package { 'r10k':  
    ensure  => installed,  
    provider => puppet_gem,  
}
```



# Using ensure\_packages

- To avoid potential package conflicts between different parts of your Puppet code or between your code and third-party modules
  - The Puppet standard library provides a useful wrapper for the package resource, called `ensure_packages()`.

# Services

- Although services are implemented in a number of varied and complicated ways at the operating system level
- Puppet does a good job of abstracting away most of this with the service resource and exposing just the two attributes of services which you most commonly need to manage
- whether they're running (ensure) and whether they start at boot time (enable).

# The hasstatus attribute

- If you find that Puppet keeps attempting to start the service on every Puppet run, even though the service is running, it may be that Puppet's default service status detection isn't working.
- In this case, you can specify the `hasstatus => false` attribute for the service (`service_hasstatus.pp`):

```
service { 'ntp':  
    ensure  => running,  
    enable   => true,  
    hasstatus => false,  
}
```



# The pattern attribute

- If hasstatus is false and pattern is specified, Puppet will search for the value of pattern in the process table to determine whether or not the service is running.
- To find the pattern you need, you can use the ps command to see the list of running processes:

`ps ax`

# The pattern attribute

- Find the process you're interested in and pick a string which will match only the name of that process.
- For example, if it's ntpd, you might specify the pattern attribute as ntpd (service\_pattern.pp):

```
service { 'ntp':  
    ensure  => running,  
    enable   => true,  
    hasstatus => false,  
    pattern  => 'ntpd',  
}
```



# The hasrestart and restart attributes

- When a service is notified (for example, if a file resource uses the notify attribute to tell the service its config file has changed, a common pattern which we looked at in lesson 2, Creating your first manifests), Puppet's default behavior is to stop the service, then start it again.
- This usually works, but many services implement a restart command in their management scripts.

# The hasrestart and restart attributes

- For example, on Ubuntu). The following example shows the use of hasrestart (service\_hasrestart.pp):

```
service { 'ntp':  
    ensure      => running,  
    enable      => true,  
    hasrestart => true,  
}
```

A screenshot of a terminal window displaying a large block of code. The code appears to be a Puppet manifest named 'service\_hasrestart.pp'. It contains numerous lines of Puppet syntax, including class definitions, resource declarations, and various parameters like 'ensure' and 'hasrestart'. The text is in a monospaced font, typical of terminal output.

# The hasrestart and restart attributes

- You can specify any restart command you like for the service using the restart attribute (service\_custom\_restart.pp):

```
service { 'ntp':  
    ensure => running,  
    enable => true,  
    restart => '/bin/echo Restarting >>/tmp/debug.log &&  
    systemctl restart ntp',  
}
```

# Users

- A user on Unix-like systems does not necessarily correspond to a human person who logs in and types commands, although it sometimes does.
- A user is simply a named entity that can own files and run commands with certain permissions and that may or may not have permission to read or modify other users' files.
- It's very common, for sound security reasons, to run each service on a system with its own user account.

# Creating users

```
group { 'devs':  
  ensure => present,  
  gid   => 3000,  
}
```

```
user { 'hsing-hui':  
  ensure => present,  
  uid   => '3001',  
  home  => '/home/hsing-hui',  
  shell  => '/bin/bash',  
  groups => ['devs'],  
}
```



# The user resource

- The title of the resource is the username (login name) of the user; in this example, hsing-hui.
- The ensure => present attribute says that the user should exist on the system.
- The uid attribute needs a little more explanation.
- On Unix-like systems, each user has an individual numerical id, known as the uid.

# The group resource

- The title of the resource is the name of the group (devs).
- You need not specify a gid attribute but, for the same reasons as the uid attribute, it's a good idea to do so.



# Managing SSH keys

- The following example shows how to use `ssh_authorized_key` to add an SSH key (mine, in this instance) to the `ubuntu` user on our Vagrant VM (`ssh_authorized_key.pp`):

```
ssh_authorized_key { 'john@fenagoconsulting.com':  
  user => 'ubuntu',  
  type => 'ssh-rsa',  
  key =>  
  'AAAAB3NzaC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqT  
  dGnJhNoBer8x6pfWkzNzeM8Zx7/2Tf2pI7kHdbsiTXEUawqzXZQtZ  
  zt/j3Oya+PZjcRpWNRzprSmd2UxEEPTqDw9LqY5S2B8og/NyzWa  
  IYPsKoatcgC7VgYHplcTbzEhGu8BsoEVBGYu3IRy5RkAcZik=',  
}
```



# Managing SSH keys

- Finally, the key attribute sets the key itself. When this manifest is applied, it adds the following to the ubuntu user's authorized\_keys file:

ssh-rsa

```
AAAAB3NzaC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2B  
zeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/2Tf2pl7kHdbsiTxEUa  
wqzXZQtZzt/j3Oya+PZjcRpWRNrzprSmd2UxEEPTqDw9LqY  
5S2B8og/NyzWalYPsKoatcgC7VgYHplcTbzEhGu8BsoEVB  
GYu3IRy5RkAcZik= john@fenagoconsulting.com
```

# Removing users

- we need to retain the user declaration for a while, but set the ensure attribute to absent (`user_remove.pp`):

```
user { 'godot':  
    ensure => absent,  
}
```



# Cron resources

- Puppet provides the cron resource for managing scheduled jobs, and we saw an example of this in the run-puppet manifest we developed in lesson 3, Managing your Puppet code with Git (run-puppet.pp):

```
cron { 'run-puppet':  
    command => '/usr/local/bin/run-puppet',  
    hour   => '*',  
    minute => '*/15',  
}
```



# Attributes of the cron resource

- The cron resource has a few other useful attributes which are shown in the following example (cron.pp):

```
cron { 'cron example':  
    command    => '/bin/date +%F',  
    user       => 'ubuntu',  
    environment => ['MAILTO=admin@example.com', 'PATH=/bin'],  
    hour       => '0',  
    minute     => '0',  
    weekday    => ['Saturday', 'Sunday'],  
}
```

# Randomizing cron jobs

- If you have several such jobs to run, you can also supply a further seed value to the `fqdn_rand()` function, which can be any string and which will ensure that the value is different for each job (`fqdn_rand.pp`):

```
cron { 'run daily backup':  
    command => '/usr/local/bin/backup',  
    minute  => '0',  
    hour    => fqdn_rand(24, 'run daily backup'),  
}  
  
cron { 'run daily backup sync':  
    command => '/usr/local/bin/backup_sync',  
    minute  => '0',  
    hour    => fqdn_rand(24, 'run daily backup sync'),  
}
```



# Removing cron jobs

- Just as with user resources, or any type of resource, removing the resource declaration from your Puppet manifest does not remove the corresponding configuration from the node.
- In order to do that you need to specify ensure => absent on the resource.

# Exec resources

- While the other resource types we've seen so far (file, package, service, user, ssh\_authorized\_key, and cron) have modeled some concrete piece of state on the node, such as a file, the exec resource is a little different.
- An exec allows you to run any arbitrary command on the node.
- This might create or modify state, or it might not; anything you can run from the command line, you can run via an exec resource.

# Automating manual interaction

- The most common use for an exec resource is to simulate manual interaction on the command line.
- Some older software is not packaged for modern operating systems, and needs to be compiled and installed from source, which requires you to run certain commands.

# Attributes of the exec resource

- The following example shows an exec resource for building and installing an imaginary piece of software (exec.pp):

```
exec { 'install-cat-picture-generator':  
    cwd    => '/tmp/cat-picture-generator',  
    command => '/tmp/cat-picture-generator/configure &&  
    /usr/bin/make install',  
    creates => '/usr/local/bin/cat-picture-generator',  
}  
}
```

# The user attribute

- If you need the command to run as a particular user, specify the user attribute, as in the following example (`exec_user.pp`):

```
exec { 'say-hello':  
    command => '/bin/echo Hello, this is `whoami`  
    >/tmp/hello-ubuntu.txt',  
    user   => 'ubuntu',  
    creates => '/tmp/hello-ubuntu.txt',  
}
```



# The user attribute

- Previous code will run the specified command as the ubuntu user.
- The whoami command returns the name of the user running it, so when you apply this manifest, the file /tmp/hello-ubuntu.txt will be created with the following contents:

Hello, this is ubuntu

# The onlyif and unless attributes

- On Unix-like systems, commands generally return an exit status of zero to indicate success and a non-zero value for failure.
- The following example shows how to use onlyif in this way (`exec_onlyif.pp`):

```
exec { 'process-incoming-cat-pictures':  
    command => '/usr/local/bin/cat-picture-generator --  
import /tmp/incoming/*',  
    onlyif => '/bin/ls /tmp/incoming/*',  
}
```

# The onlyif and unless attributes

- You can see what command is being run and what output it generates by running sudo puppet apply with the -d (debug) flag:

```
sudo puppet apply -d exec_onlyif.pp
```

```
Debug: Exec[process-incoming-cat-pictures](provider=posix): Executing check '/bin/ls /tmp/incoming/*'
```

```
Debug: Executing: '/bin/ls /tmp/incoming/*'
```

```
Debug: /Stage[main]/Main/Exec[process-incoming-cat-pictures]/onlyif: /tmp/incoming/foo
```

# The refreshonly attribute

```
file { '/etc/aliases':  
    content => 'root: john@fenagoconsulting.com',  
    notify => Exec['newaliases'],  
}
```

```
exec { 'newaliases':  
    command => '/usr/bin/newaliases',  
    refreshonly => true,  
}
```



# The logoutput attribute

- When Puppet runs shell commands via an exec resource, the output is normally hidden from us.
- However, if the command doesn't seem to be working properly, it can be very useful to see what output it produced, as this usually tells us why it didn't work.
- The logoutput attribute determines whether Puppet will log the output of the exec command along with the usual informative Puppet output.

# The timeout attribute

- Sometimes, commands can take a long time to run, or never terminate at all.
- By default, Puppet allows an exec command to run for 300 seconds, at which point Puppet will terminate it if it has not finished.
- If you need to allow a little longer for the command to complete, you can use the timeout attribute to set this.
- The value is the maximum execution time for the command in seconds.

# How not to misuse exec resources

- The exec resource can do anything to the system that you could do from the command line.
- As you can imagine, such a powerful tool can be misused.
- In theory, Puppet is a declarative language: the manifest specifies the way things should be, and it is up to Puppet to take the necessary actions to make them so.

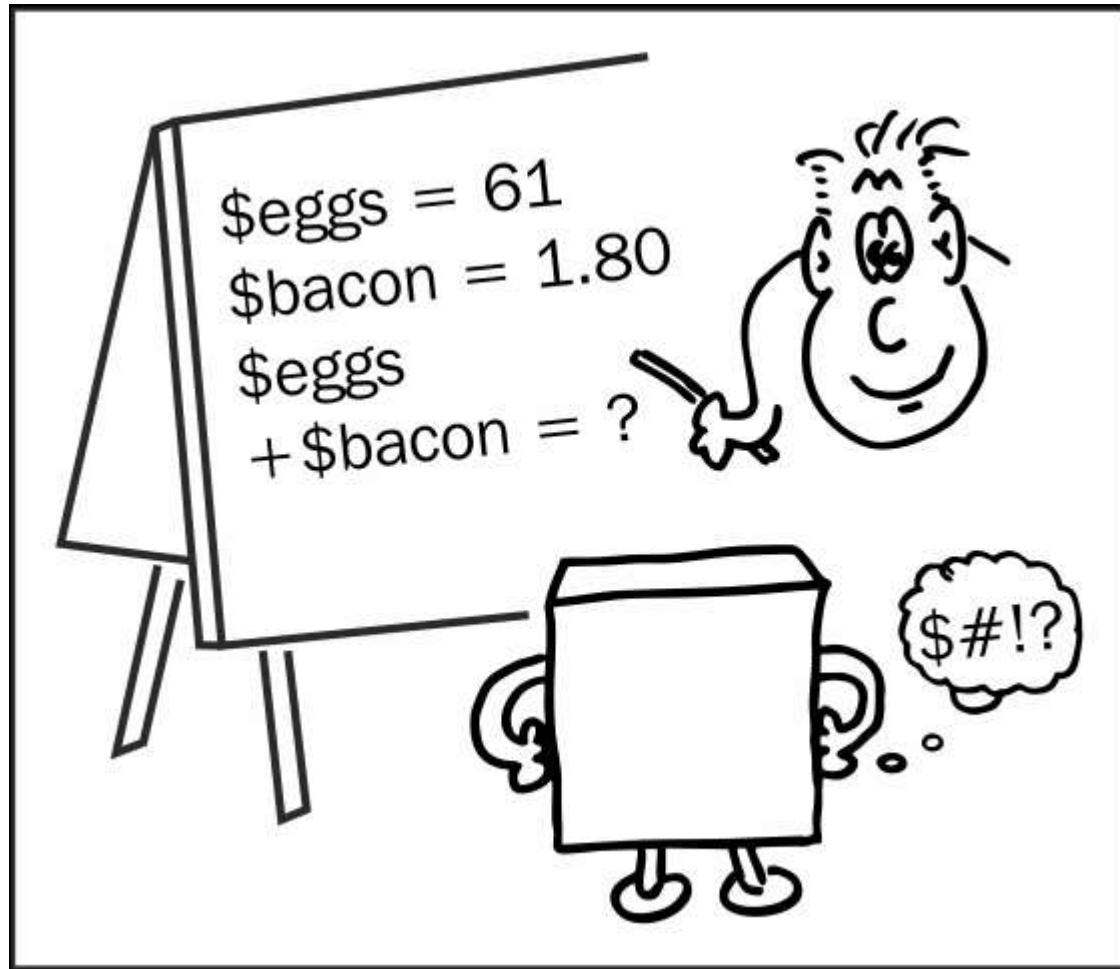
# Summary



- We've explored Puppet's file resource in detail, covering file sources, ownership, permissions, directories, symbolic links, and file trees.
- We've learned how to manage packages by installing specific versions, or the latest version, and how to uninstall packages.
- We've covered Ruby gems, both in the system context and Puppet's internal context.
- Along the way, we met the very useful puppet-lint tool.

# 5. Variables, expressions, and facts



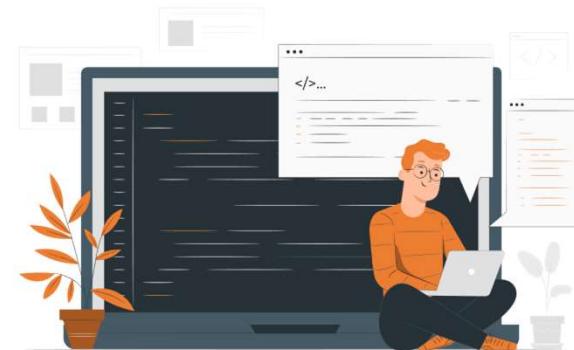


# Introducing variables

- A variable in Puppet is simply a way of giving a name to a particular value, which we could then use wherever we would use the literal value (variable\_string.pp):

```
$php_package = 'php7.0-cli'
```

```
package { $php_package:  
  ensure => installed,  
}
```



# Introducing variables

- A variable can contain different types of data; one such type is a String (like php7.0-cli), but Puppet variables can also contain Number or Boolean values (true or false).
- Here are a few examples (variable\_simple.pp):

```
$my_name = 'Zaphod Beeblebrox'  
$answer = 42  
$scheduled_for_demolition = true
```



# Using Booleans

- Strings and numbers are straightforward, but Puppet also has a special data type to represent true or false values, which we call Boolean values, after the logician George Boole.
- We have already encountered some Boolean values in Puppet resource attributes (service.pp):

```
service { 'sshd':  
    ensure => running,  
    enable => true,  
}
```



# Interpolating variables in strings

- When you do this, Puppet inserts the current value of the variable into the contents of the string, replacing the name of the variable.
- String interpolation looks like this  
(string\_interpolation.pp):

```
$my_name = 'John'  
notice("Hello, ${my_name}! It's great to meet you!")
```

# Interpolating variables in strings

- When you apply this manifest, the following output is printed:

Notice: Scope(Class[main]): Hello, John! It's great to meet you!

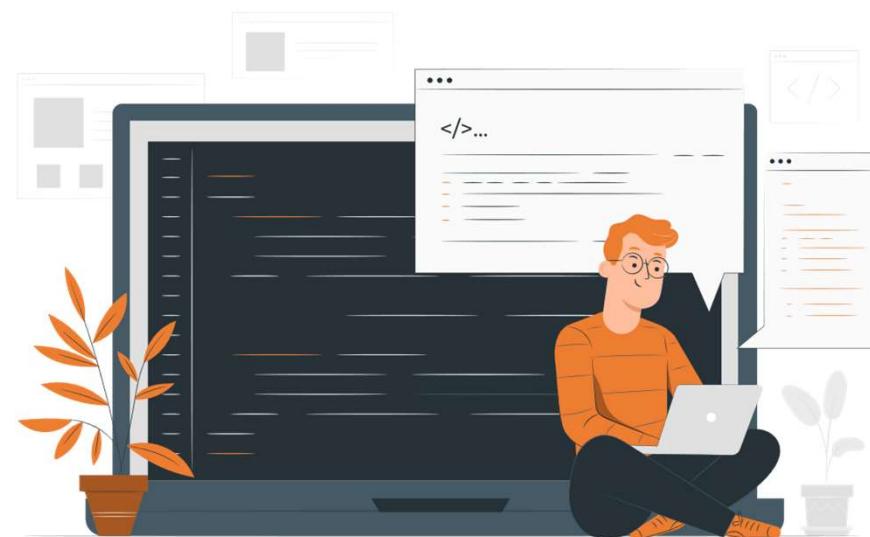
# Creating arrays

- A variable can also hold more than one value.
- An Array is an ordered sequence of values, each of which can be of any type.
- The following example creates an array of Integer values (`variable_array.pp`):

```
$heights = [193, 120, 181, 164, 172]  
$first_height = $heights[0]
```

# Declaring arrays of resources

```
$dependencies = [  
    'php7.0-cgi',  
    'php7.0-cli',  
    'php7.0-common',  
    'php7.0-gd',  
    'php7.0-json',  
    'php7.0-mcrypt',  
    'php7.0-mysql',  
    'php7.0-soap',  
]  
  
package { $dependencies:  
    ensure => installed,  
}
```



# Declaring arrays of resources

- If our intuition is right, applying the previous manifest should give us a package resource for each package listed in the \$dependencies array, and each one should be installed.
- Here's what happens when the manifest is applied:

```
sudo apt-get update
```

```
sudo puppet apply /examples/resource_array.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.68 seconds
```

```
Notice: /Stage[main]/Main/Package[php7.0-cgi]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-cli]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-common]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-gd]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-json]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-mcrypt]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-mysql]/ensure: created
```

```
Notice: /Stage[main]/Main/Package[php7.0-soap]/ensure: created
```

```
Notice: Applied catalog in 56.98 seconds
```

# Understanding hashes

- A hash, also known as a dictionary in some programming languages, is like an array, but instead of just being a sequence of values, each value has a name (variable\_hash.pp):

```
$heights = {  
    'john'  => 193,  
    'rabiah' => 120,  
    'abigail' => 181,  
    'melina'  => 164,  
    'sumiko'  => 172,  
}  
notice("John's height is ${heights['john']}cm.")
```

# Setting resource attributes from a hash

```
$attributes = {  
  'owner' => 'ubuntu',  
  'group' => 'ubuntu',  
  'mode'  => '0644',  
}  
  
file { '/tmp/test':  
  ensure => present,  
  *      => $attributes,  
}
```



# Setting resource attributes from a hash

- The \* character, cheerfully named the attribute splat operator, tells Puppet to treat the specified hash as a list of attribute-value pairs to apply to the resource.
- This is exactly equivalent to specifying the same attributes directly, as in the following example:

```
file { '/tmp/test':  
    ensure => present,  
    owner  => 'vagrant',  
    group  => 'vagrant',  
    mode   => '0644',  
}
```

# Introducing expressions

- Variables are not the only things in Puppet that have a value.
- Expressions also have a value.
- The simplest expressions are just literal values:

42

true

'Oh no, not again.'

# Introducing expressions

- You can combine numeric values with arithmetic operators, such as +, -, \*, and /, to create arithmetic expressions, which have a numeric value, and you can use these to have Puppet do calculations (`expression_numeric.pp`):

```
$value = (17 * 8) + (12 / 4) - 1  
notice($value)
```



# Introducing expressions

- The most useful expressions, though, are which that evaluate to true or false, known as Boolean expressions.
- The following is a set of examples of Boolean expressions, all of which evaluate to true (`expression_boolean.pp`):

Refer to the file `5_1.txt`

# Introducing regular expressions

- The `=~` operator tries to match a given value against a regular expression.
- A regular expression (regular in the sense of constituting a pattern or a rule) is a special kind of expression which specifies a set of strings.
- For example, the regular expression `/a+/` describes the set of all strings that contain one or more consecutive as: `a`, `aa`, `aaa`, and so on,

# Introducing regular expressions

- The following example shows some regular expressions that match the string foo (regex.pp):

```
$candidate = 'foo'  
notice($candidate =~ /foo/) # literal  
notice($candidate =~ /f/) # substring  
notice($candidate =~ /f.*/) # f followed by zero or more  
characters  
notice($candidate =~ /f.o/) # f, any character, o  
notice($candidate =~ /fo+/) # f followed by one or more 'o's  
notice($candidate =~ /[fgh]oo/) # f, g, or h followed by 'oo'
```

# Using conditional expressions

- Boolean expressions, like those in the previous example, are useful because we can use them to make choices in the Puppet manifest.
- We can apply certain resources only if a given condition is met, or we can assign an attribute one value or another, depending on whether some expression is true.
- An expression used in this way is called a conditional expression.

# Making decisions with if statements

```
$install_perl = true  
if $install_perl {  
    package { 'perl':  
        ensure => installed,  
    }  
} else {  
    package { 'perl':  
        ensure => absent,  
    }  
}
```



# Making decisions with if statements

- You can see that the value of the Boolean variable \$install\_perl governs whether or not the perl package is installed.
- If \$install\_perl is true, Puppet will apply the following resource:

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

# Making decisions with if statements

- If, on the other hand, \$install\_perl is false, the resource applied will be:

```
package { 'perl':  
    ensure => absent,  
}
```



# Choosing options with case statements

- The if statement allows you to take a yes/no decision based on the value of a Boolean expression.
- But if you need to make a choice among more than two options, you can use a case statement instead (case.pp):

Refer to the file 5\_2.txt

# Finding out facts

- It's very common for Puppet manifests to need to know something about the system they're running on, for example, its hostname, IP address, or operating system version.
- Puppet's built-in mechanism for getting system information is called Facter, and each piece of information provided by Facter is known as a fact.

# Using the facts hash

- You can access Facter facts in your manifest using the facts hash.
- This is a Puppet variable called \$facts which is available everywhere in the manifest, and to get a particular fact, you supply the name of the fact you want as the key (facts\_hash.pp):

```
notice($facts['kernel'])
```



# Using the facts hash

- In older versions of Puppet, each fact was a distinct global variable, like this:

```
notice($::kernel)
```



# Running the facter command

- You can also use the facter command to see the value of particular facts, or just see what facts are available.
- For example, running facter os on the command line will show you the hash of available OS-related facts:

Refer to the file 5\_3.txt

# Accessing hashes of facts

- To access a value inside a hash, you add another key name in square brackets after the first, as in the following example (`facts_architecture.pp`):

```
notice($facts['os']['architecture'])
```

- You can keep on appending more keys to get more and more specific information

(`facts_distro_codename.pp`):

```
notice($facts['os']['distro']['codename'])
```

# Referencing facts in expressions

- Just as with ordinary variables or values, you can use facts in expressions, including conditional expressions (fact\_if.pp):

```
if $facts['os']['selinux']['enabled'] {  
    notice('SELinux is enabled')  
} else {  
    notice('SELinux is disabled')  
}
```



# Using memory facts

- you might decide to set this to three-quarters of total memory (for example), using a fact and an arithmetic expression, as in the following snippet (`fact_memory.pp`):

```
$buffer_pool = $facts['memory']['system']['total_bytes'] *  
3/4  
notice("innodb_buffer_pool_size=${buffer_pool}")
```

# Discovering networking facts

- Most applications use the network, so you'll find Facter's network-related facts very useful for anything to do with network configuration.
- The most commonly used facts are the system hostname, fully qualified domain name (FQDN), and IP address (`fact_networking.pp`):

```
notice("My hostname is ${facts['hostname']}")  
notice("My FQDN is ${facts['fqdn']}")  
notice("My IP is ${facts['networking']['ip']}")
```

# Providing external facts

- Puppet looks for external facts in the /opt/puppetlabs/facter/facts.d/ directory.
- Try creating a file in that directory called facts.txt with the following contents (fact\_external.txt):

cloud=aws

- A quick way to do this is to run the following command:

```
sudo cp /examples/fact_external.txt  
/opt/puppetlabs/facter/facts.d
```

# Providing external facts

- The cloud fact is now available in your manifests.
- You can check that the fact is working by running the following command:

```
sudo facter cloud  
aws
```



# Providing external facts

- To use the fact in your manifest, query the \$facts hash just as you would for a built-in fact (`fact_cloud.pp`):

```
case $facts['cloud'] {  
    'aws': {  
        notice('This is an AWS cloud node ')  
    }  
    'gcp': {  
        notice('This is a Google cloud node')  
    }  
    default: {  
        notice("I'm not sure which cloud I'm in!")  
    }  
}
```



# Creating executable facts

- Run the following command to copy the executable fact example into the external fact directory:

```
sudo cp /examples/date.sh  
/opt/puppetlabs/facter/facts.d
```

- Set the execute bit on the file with the following command:

```
sudo chmod a+x /opt/puppetlabs/facter/facts.d/date.sh
```

# Creating executable facts

- Now test the fact:

```
sudo facter date
```

```
2017-04-12
```

- Here is the script which generates this output  
(date.sh):

```
#!/bin/bash
```

```
echo "date=`date +%F`"
```

# Iterating over arrays

- Iteration (doing something repeatedly) is a useful technique in your Puppet manifests to avoid lots of duplicated code.
- For example, consider the following manifest, which creates several files with identical properties (`iteration_simple.pp`):

Refer to the file `5_4.txt`

# Using the each function

- Puppet provides the each function to help with just this kind of situation.
- The each function takes an array and applies a block of Puppet code to each element of the array.

```
$tasks = ['task1', 'task2', 'task3']
$tasks.each | $task | {
  file { "/usr/local/bin/${task}":
    content => "echo I am ${task}\n",
    mode   => '0755',
  }
}
```



# Using the each function

- Now this looks more like a computer program! We have a loop, created by the each function.
- The loop goes round and round, creating a new file resource for each element of the \$tasks array.
- Let's look at a schematic version of an each loop:

```
ARRAY.each | ELEMENT | {  
    BLOCK  
}
```

# Iterating over hashes

- The each function works not only on arrays, but also on hashes.
- When iterating over a hash, the loop takes two ELEMENT parameters: the first is the hash key, and the second is the value.

```
$nics = $facts['networking']['interfaces']
$nics.each | String $interface, Hash $attributes | {
    notice("Interface ${interface} has IP ${attributes['ip']}")
}
```

# Iterating over hashes

- Applying the manifest in the previous example gives this result (on my Vagrant box):

```
sudo puppet apply /examples/iteration_hash.pp
```

```
Notice: Scope(Class[main]): Interface enp0s3 has IP  
10.0.2.15
```

```
Notice: Scope(Class[main]): Interface lo has IP  
127.0.0.1
```



# Summary

- In this lesson, we've gained an understanding of how Puppet's variable and data type system works, including the basic data types: Strings, Numbers, Booleans, Arrays, and Hashes.
- We've seen how to interpolate variables in strings and how to quickly create sets of similar resources using an array of resource names.



# 6. Managing data with Hiera





# Why Hiera?

- What do we mean by configuration data? There will be lots of pieces of information in your manifests which we can regard as configuration data: for example, the values of all your resource attributes.
- Look at the following example:

```
package { 'puppet-agent':  
    ensure => '5.2.0-1xenial',  
}
```



# Data needs to be maintained

- Multiply this by all the packages managed throughout your manifest, and there is already a problem.
- But this is just one piece of data that needs to be maintained, and there are many more: the times of cron jobs, the email addresses for reports to be sent to, the URLs of files to fetch from the web
- The parameters for monitoring checks, the amount of memory to configure for the database server, and so on.

# Settings depend on nodes

- Mixing data with code makes it harder to find and edit that data.
- But there's another problem, What if you have two nodes to manage with Puppet, and there's a config value which needs to be different on each of them?
- For example, they might both have a cron job to run the backup, but the job needs to run at a different time on each node.

# Operating systems differ

- What if you have some nodes running Ubuntu 16, and some on Ubuntu 18?
- As you'll know if you've ever had to upgrade the operating system on a node, things change from one version to the next.
- For example, the name of the database server package might have changed from mysql-server to mariadb-server.



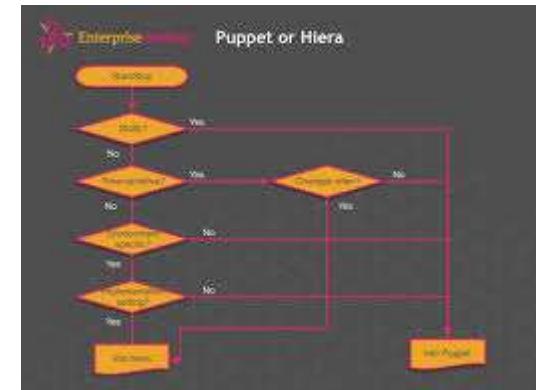
# The Hiera way

- What we want is a kind of central database in Puppet where we can look up configuration settings.
- The data should be stored separately from Puppet code, and make it easy to find and edit values.
- It should be possible to look up values with a simple function call in Puppet code or templates.

# The Hiera way

- Fortunately, Hiera does exactly this. Hiera lets you store your config data in simple text files (actually, YAML, JSON, or HOCON files, which use popular structured text formats), and it looks like the following example:

```
---  
test: 'This is a test'  
consul_node: true  
apache_worker_factor: 100  
apparmor_enabled: true  
...
```

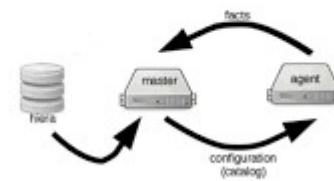


# The Hiera way

- In your manifest, you query the database using the `lookup()` function, as in the following example (`lookup.pp`):

```
file { lookup('backup_path', String):  
    ensure => directory,  
}
```

Puppet Architecture w/  
hiera



# Setting up Hiera

- Hiera needs to know one or two things before you can start using it, which are specified in the Hiera configuration file, named `hiera.yaml` (not to be confused this with Hiera data files, which are also YAML files, and we'll find about those later in this lesson.)
- Each Puppet environment has its own local Hiera config file, located at the root of the environment directory

# Setting up Hiera

- The following example shows a minimal `hiera.yaml` file (`hiera_minimal.config.yaml`):

# version: 5

## defaults:

**datadir:** data

## data\_hash: yaml\_data

# hierarchy:

- name: "Common defaults"  
path: "common.yaml"

# Adding Hiera data to your Puppet repo

- Your Vagrant VM is already set up with a suitable Hiera config and the sample data file, in the /etc/puppetlabs/code/environments/pbg directory.
- Try it now:
- Run the following commands:  
`sudo puppet lookup --environment pbg test`  
--- This is a test



# Troubleshooting Hiera

- If you see the warning Config file not found, using Hiera defaults, check that your Vagrant box has an /etc/puppetlabs/code/environments/pbg directory.
- If not, destroy and re-provision your Vagrant box with:

```
vagrant destroy  
scripts/start_vagrant.sh
```



# Troubleshooting Hiera

- If you see an error like the following, it generally indicates a problem with the Hiera data file syntax:

Error: Evaluation Error: Error while evaluating a Function Call,  
(/etc/puppetlabs/code/environments/pbg/hiera.yaml): did not find expected key while parsing a block mapping at line 11 column 5 at line 1:8 on node ubuntu-xenial

# Querying Hiera

- In general, you can use a call to `lookup()` anywhere in your Puppet manifests you might otherwise use a literal value. The following code shows some examples of this (`lookup2.pp`):

```
notice("Apache is set to use  
${lookup('apache_worker_factor', Integer)} workers")  
unless lookup('apparmor_enabled', Boolean) {  
  exec { 'apt-get -y remove apparmor': }  
}  
notice('dns_allow_query enabled:',  
  lookup('dns_allow_query', Boolean))
```



# Querying Hiera

- To apply this manifest in the example environment, run the following command:

```
sudo puppet apply --environment pbg  
/examples/lookup2.pp
```

Notice: Scope(Class[main]): Apache is set to use 100 workers

Notice: Scope(Class[main]): dns\_allow\_query enabled: true

# Typed lookups

- If you accidentally look up the wrong key, or mistype the value in the data file, you'll get an error like this:

```
Error: Evaluation Error: Error while evaluating a
Function Call, Found value has wrong type, expects a
Boolean value, got String at
/examples/lookup_type.pp:1:8 on node ubuntu-xenial
```

# Types of Hiera data

syslog\_server: '10.170.81.32'

monitor\_ips:

- '10.179.203.46'
- '212.100.235.160'
- '10.181.120.77'
- '94.236.56.148'

cobbler\_config:

manage\_dhcp: true

pxe\_just\_once: true



# Single values

- Most Hiera data consists of a key associated with a single value, as in the previous example:

`syslog_server: '10.170.81.32'`

- The value can be any legal Puppet value, such as a String, as in this case, or it can be an Integer:

`apache_worker_factor: 100`

# Boolean values

- Hiera is fairly liberal in what it interprets as Boolean values: any of true, on, or yes (with or without quotes) are interpreted as a true value, and false, off, or no are interpreted as a false value.
- For clarity, though, stick to the following format:

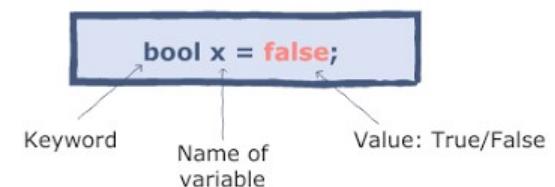
consul\_node: true

# Boolean values

- When you use `lookup()` to return a Boolean value in your Puppet code, you can use it as the conditional expression in, for example, an `if` statement:

```
if lookup('is_production', Boolean) {
```

```
  ...
```

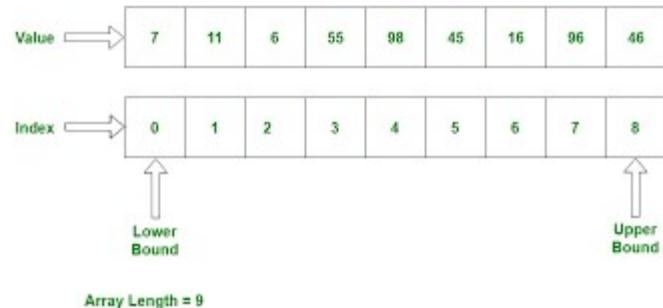


# Arrays

- Usefully, Hiera can also store an array of values associated with a single key:

## monitor\_ips:

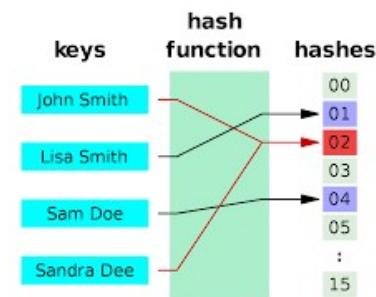
- '10.179.203.46'
  - '212.100.235.160'
  - '10.181.120.77'
  - '94.236.56.148'



# Hashes

- As we saw in lesson 5, Variables, expressions, and facts, a hash (also called a dictionary in some programming languages) is like an array where each value has an identifying name (called the key), as in the following example:

```
cobbler_config:  
  manage_dhcp: true  
  pxe_just_once: true
```



# Hashes

```
$cobbler_config = lookup('cobbler_config', Hash)
$manage_dhcp = $cobbler_config['manage_dhcp']
$pxe_just_once = $cobbler_config['pxe_just_once']
if $pxe_just_once {
    notice('pxe_just_once is enabled')
} else {
    notice('pxe_just_once is disabled')
}
```

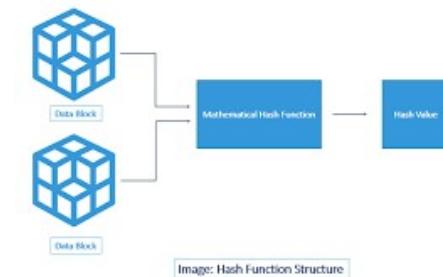


Image: Hash Function Structure

# Hashes

- Since it's very common for Hiera data to be a hash of hashes, you can retrieve values from several levels down in a hash by using the following "dot notation" (lookup\_hash\_dot.pp):

```
$web_root = lookup('cms_parameters.static.web_root',  
String)  
notice("web_root is ${web_root}")
```

# Interpolation in Hiera data

- Hiera data is not restricted to literal values; it can also include the value of Facter facts or Puppet variables, as in the following example:

```
backup_path: "/backup/%{facts.hostname}"
```



# Using lookup()

```
ips:  
    home: '130.190.0.1'  
    office1: '74.12.203.14'  
    office2: '95.170.0.75'  
  
firewall_allow_list:  
    - "%{lookup('ips.home')}"  
    - "%{lookup('ips.office1')}"  
    - "%{lookup('ips.office2')}"
```



# Using alias()

- lets you re-use any Hiera data structure within Hiera, just by referencing its name:

firewall\_allow\_list:

- "%{lookup('ips.home')}"
- "%{lookup('ips.office1')}"
- "%{lookup('ips.office2')}"

vpn\_allow\_list: "%{alias('firewall\_allow\_list')}"



# Using literal()

- Because the percent character (%) tells Hiera to interpolate a value, you might be wondering how to specify a literal percent sign in data.
- For example, to write the value %{HTTP\_HOST} as Hiera data, we would need to write:

```
%{literal('%')}{HTTP_HOST}
```

# Using literal()

- You can see a more complicated example in the sample Hiera data file:

```
force_www_rewrite:  
  comment: "Force WWW"  
  rewrite_cond: "%{literal('%')}{HTTP_HOST} !^www\\.[  
[NC]"  
  rewrite_rule: "^(.*)$  
https://www.%{literal('%')}{HTTP_HOST}%{literal('%')}{R  
EQUEST_URI} [R=301,L]"
```

# The hierarchy

hierarchy:

- ...
- name: "Host-specific data"  
path: "nodes/%{facts.hostname}.yaml"
- name: "OS release-specific data"  
path: "os/%{facts.os.release.major}.yaml"
- name: "OS distro-specific data"  
path: "os/%{facts.os.distro.codename}.yaml"
- name: "Common defaults"  
path: "common.yaml"



# Dealing with multiple values

- You may be wondering what happens if the same key is listed in more than one Hiera data source.
- For example, imagine the first source contains the following:

`consul_node: false`

- Also, assume that `common.yaml` contains:

`consul_node: true`



# Merge behaviors

- Hiera allows you to specify which of these strategies it should use when multiple values match your lookup.
- This is called a merge behavior, and you can specify which merge behavior you want as the third argument to `lookup()`, after the key and data type (`lookup_merge.pp`):

```
notice(lookup('firewall_allow_list', Array, 'unique'))
```

# Data sources based on facts

- The hierarchy mechanism lets you set common default values for all situations (usually in common.yaml), but override them in specific circumstances.
- For example, you can set a data source in the hierarchy based on the value of a Puppet fact, such as the hostname:
  - name: "Host-specific data"  
path: "nodes/%{facts.hostname}.yaml"

# Data sources based on facts

- If something has changed from one version to the next that affects your Puppet manifest, you can use the `os.distro.codename` fact to select the appropriate Hiera data, as in the following example:
  - name: "OS-specific data"  
path: "os/%{facts.os.distro.codename}.yaml"
- Alternatively, you can use the `os.release.major` fact:
  - name: "OS-specific data"  
path: "os/%{facts.os.release.major}.yaml"

# What belongs in Hiera?

- What data should you put in Hiera, and what should be in your Puppet manifests?
- A good rule of thumb about when to separate data and code is to ask yourself what might change in the future.
- For example, the exact version of a package is a good candidate for Hiera data, because it's quite likely you'll need to update it in the future.

# Creating resources with Hiera data

- When we started working with Puppet, we created resources directly in the manifest using literal attribute values.
- In this lesson, we've seen how to use Hiera data to fill in the title and attributes of resources in the manifest. We can now take this idea one step further and create resources directly from Hiera queries.

# Building resources from Hiera arrays

- Run the following command:

```
sudo puppet apply --environment pbg  
/examples/hiera_users.pp
```

```
Notice: /Stage[main]/Main/User[katy]/ensure: created  
Notice: /Stage[main]/Main/User[lark]/ensure: created  
Notice: /Stage[main]/Main/User[bridget]/ensure: created  
Notice: /Stage[main]/Main/User[hsing-hui]/ensure:  
created  
Notice: /Stage[main]/Main/User[charles]/ensure: created
```

# Building resources from Hiera arrays

- Here's the data we're using (from the /etc/puppetlabs/code/environments/pbg/data/common.yaml file):

users:

- 'katy'
- 'lark'
- 'bridget'
- 'hsing-hui'
- 'charles'



# Building resources from Hiera arrays

- And here's the code which reads it and creates the corresponding user instances (`hiera_users.pp`):

```
lookup('users', Array[String]).each | String $username | {
    user { $username:
        ensure => present,
    }
}
```

# Building resources from Hiera hashes

- Run the following command:

```
sudo puppet apply --environment pbg /examples/hiera_users2.pp
Notice: Compiled catalog for ubuntu-xenial in environment pbg in 0.05 seconds
Notice: /Stage[main]/Main/User[katy]/uid: uid changed 1001 to 1900
Notice: /Stage[main]/Main/User[katy]/shell: shell changed " to '/bin/bash'
Notice: /Stage[main]/Main/User[lark]/uid: uid changed 1002 to 1901
Notice: /Stage[main]/Main/User[lark]/shell: shell changed " to '/bin/sh'
Notice: /Stage[main]/Main/User[brIDGET]/uid: uid changed 1003 to 1902
Notice: /Stage[main]/Main/User[brIDGET]/shell: shell changed " to '/bin/bash'
Notice: /Stage[main]/Main/User[hsing-hui]/uid: uid changed 1004 to 1903
Notice: /Stage[main]/Main/User[hsing-hui]/shell: shell changed " to '/bin/sh'
Notice: /Stage[main]/Main/User[charles]/uid: uid changed 1005 to 1904
Notice: /Stage[main]/Main/User[charles]/shell: shell changed " to '/bin/bash'
Notice: Applied catalog in 0.17 seconds
```

# Building resources from Hiera arrays

- The first difference from the previous example is that instead of the data being a simple array, it's a hash of hashes:

Refer to the file [6\\_1.txt](#)



# Building resources from Hiera arrays

- Here's the code which processes that data (hiera\_users2.pp):

```
lookup('users2', Hash, 'hash').each | String $username,  
Hash $attrs | {  
    user { $username:  
        * => $attrs,  
    }  
}
```



# Building resources from Hiera arrays

- When we call each on this hash, we specify two parameters to the loop instead of one:

| String \$username, Hash \$attrs |



# Building resources from Hiera arrays

- Inside the loop, we create a user resource for each element of the hash:

```
user { $username:  
  * => $attrs,  
}
```



# Building resources from Hiera arrays

- You may recall from the previous lesson that the \* operator (the attribute splat operator) tells Puppet to treat \$attrs as a hash of attribute-value pairs.
- So the first time round the loop, with user katy, Puppet will create a user resource equivalent to the following manifest:

```
user { 'katy':  
    ensure => present,  
    uid   => 1900,  
    shell => '/bin/bash',  
}
```



## The advantages of managing resources with Hiera data

- The previous example makes it easy to manage users across your network without having to edit Puppet code
- If you want to remove a user, for example, you would simply change her ensure attribute in the Hiera data to absent.
- Although each of the users happens to have the same set of attributes specified, this isn't essential

# Managing secret data

- Puppet often needs to know your secrets; for example, passwords, private keys, and other credentials need to be configured on the node, and Puppet must have access to this information.
- The problem is how to make sure that no-one else does.



# Setting up GnuPG

- Run the following command:  
`sudo apt-get install gnupg rng-tools`
- Once GnuPG is installed, run the following command to generate a new key pair:  
`gpg --gen-key`

# Setting up GnuPG

- When prompted, select the RSA and RSA key type:

Please select what kind of key you want:

- (1) RSA and RSA (default)
- (2) DSA and Elgamal
- (3) DSA (sign only)
- (4) RSA (sign only)

Your selection? 1



# Setting up GnuPG

- Select a 2,048 bit key size:

RSA keys may be between 1024 and 4096 bits long.

What keysize do you want? (2048) 2048

- Enter 0 for the key expiry time:

Key is valid for? (0) 0

Key does not expire at all

Is this correct? (y/N) y



# Setting up GnuPG

- When prompted for a real name, email address, and comment for the key, enter whatever is appropriate for your site:

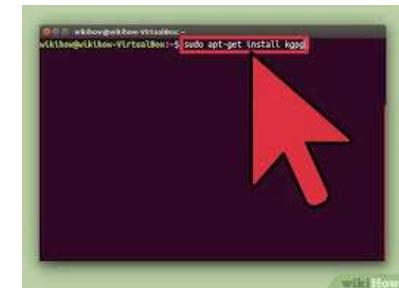
Real name: Puppet

Email address: puppet@cat-pictures.com

Comment:

You selected this USER-ID:

"Puppet <puppet@cat-pictures.com>"



Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o

# Setting up GnuPG

- It may take a few moments to generate the key, but once this is complete, GnuPG will print out the key fingerprint and details (yours will look different):

```
pub 2048R/40486112 2016-09-30
    Key fingerprint = 6758 6CEE D221 7AA0 8369
    FF3A FEC1 0055 4048 6112
uid          Puppet <puppet@cat-pictures.com>
sub 2048R/472954EB 2016-09-30
```

# Since it's Adding an encrypted Hiera source

- A Hiera source using GPG-encrypted data needs a couple of extra parameters.
- Here's the relevant section from the example `hiera.yaml` file:
  - `name: "Secret data (encrypted)"`  
`lookup_key: eyaml_lookup_key`  
`path: "secret.eyaml"`  
`options:`  
`gpg_gnupghome: '/home/ubuntu/.gnupg'`



# Creating an encrypted secret

- Create a new empty Hiera data file with the following commands:

```
cd /etc/puppetlabs/code/environments/pbg  
sudo touch data/secret.eyaml
```



# Creating an encrypted secret

- Run the following command to edit the data file using the eyaml editor (which automatically encrypts the data for you when you save it).
- Instead of `puppet@cat-pictures.com`, use the email address that you entered when you created your GPG key.

```
sudo /opt/puppetlabs/puppet/bin/eyaml edit --gpg-always-trust --gpg-recipients=puppet@cat-pictures.com data/secret.eyaml
```

# Creating an encrypted secret

- Your selected editor will be started with the following text already inserted in the file:

```
#| This is eyaml edit mode. This text (lines starting with #| at the top of the
#| file) will be removed when you save and exit.
#| - To edit encrypted values, change the content of the DEC(<num>)::PKCS7[]!
#| block (or DEC(<num>)::GPG[]!).
#| WARNING: DO NOT change the number in the parentheses.
#| - To add a new encrypted value copy and paste a new block from the
#| appropriate example below. Note that:
#|   * the text to encrypt goes in the square brackets
#|   * ensure you include the exclamation mark when you copy and paste
#|   * you must not include a number when adding a new block
#| e.g. DEC::PKCS7[]! -or- DEC::GPG[]!
```

# Creating an encrypted secret

- Enter the following text below the commented message, exactly as shown, including the beginning three hyphens:

---

test\_secret: DEC::GPG[This is a test secret]!



# Creating an encrypted secret

- Run the following command to test that Puppet can read and decrypt your secret:

```
sudo puppet lookup --environment pbg test_secret  
--- This is a test secret
```

# How Hiera decrypts secrets

- To prove to yourself that the secret data is actually encrypted, run the following command to see what it looks like in the data file on disk:

```
cat data/secret.eyaml
```

```
---
```

```
test_secret:  
ENC[GPG,hQEMA4+8DyxHKVTrAQf/QQPL4zD2kkU7T+FhaEdptu68RAw2m2KAX  
GujjnQPXoONrbh1QjtZiJBlhqOP+7JwvzejED0NXNMkmWTGfCrOBvQIZS0U9Vrgs  
yq5mACPHyeLqFbdeOjNEIR7gLP99aykAmbO2mRqfXvns+cZgaTUEPXOPyipY5Q  
6w6/KeBEvekTlZ6ME9Oketj+1/zxDz4qWH+0nLwdD9L279d7hnokpts2tp+gpCUc0/q  
KsTXpdTRPE2R0kg9BI84OP3fFITSTgcT+pS8Dfa1/ZzALfHmULcC3hckG9ZSR+0cd  
6MyJzucwiJCrelfR/cDfqpsENNM6PNkTAHEHrAqPrSDXilg1KtJSAfZ9rS8KtRyoSsk  
+XyrxIRH/S1Qg1dgFb8VqJzWjFl6GBJZemy7z+xjoWHyznbABVwp0KXNGgn/0idxfh  
z1mTo2/49POFiVF4MBo/6/EEU4cw==]
```

# How Hiera decrypts secrets

- When you reference a Hiera key such as `test_secret` in your manifest, what happens next? Hiera consults its list of data sources configured in `hiera.yaml`.
- The first source in the hierarchy is `secret.eyaml`, which contains the key we're interested in (`test_secret`). Here's the value:

ENC[GPG,hQEMA4 ... EEU4cw==]



# How Hiera decrypts secrets

- Assuming it finds the key, GPG decrypts the data and passes the result back to Hiera, which returns it to Puppet, and the result is the plaintext:

This is a test secret



# Editing or adding encrypted secrets



- If the secret data is stored in encrypted form, you might be wondering how to edit it when you want to change the secret value.
- Fortunately, there's a way to do this. Recall that when you first entered the secret data, you used the following command:

```
sudo /opt/puppetlabs/puppet/bin/eyaml edit --gpg-
always-trust --gpg-recipients=puppet@cat-pictures.com
data/secret.eyaml
```

# How Hiera decrypts secrets

- If you run the same command again, you'll find that you're looking at your original plaintext (along with some explanatory comments):

---

test\_secret: DEC(1)::GPG[This is a test secret]!

# How Hiera decrypts secrets



- There's an example on your Vagrant box, at `/examples/eyaml_edit.sh`, which you can copy to `/usr/local/bin` and edit (as before, substitute the gpg-recipients email address with the one associated with your GPG key):

```
#!/bin/bash
/opt/puppetlabs/puppet/bin/eyaml edit --gpg-always-trust --
gpg-recipients=puppet@cat-pictures.com
/etc/puppetlabs/code/environments/pbg/data/secret.eyaml
```

# How Hiera decrypts secrets

- Now, whenever you need to edit your secret data, you can simply run the following command:  
`sudo eyaml_edit`
- To add a new secret, add a line like this:  
`new_secret: DEC::GPG[Somebody wake up Hicks]!`

# Distributing the decryption key

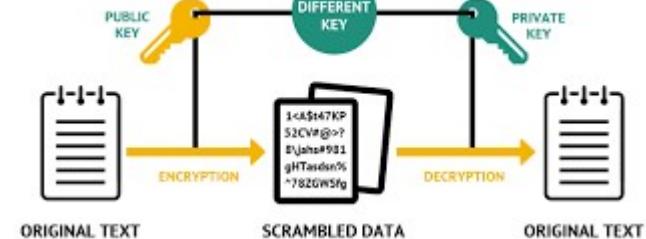
- Now that your Puppet manifests use encrypted Hiera data, you'll need to make sure that each node running Puppet has a copy of the decryption key.
- Export the key to a text file using the following command (use your key's email address, of course):

```
sudo sh -c 'gpg --export-secret-key -a puppet@cat-pictures.com >key.txt'
```

# Distributing the decryption key

- Copy the key.txt file to any nodes which need the key, and run the following command to import it:

```
sudo gpg --import key.txt  
sudo rm key.txt
```



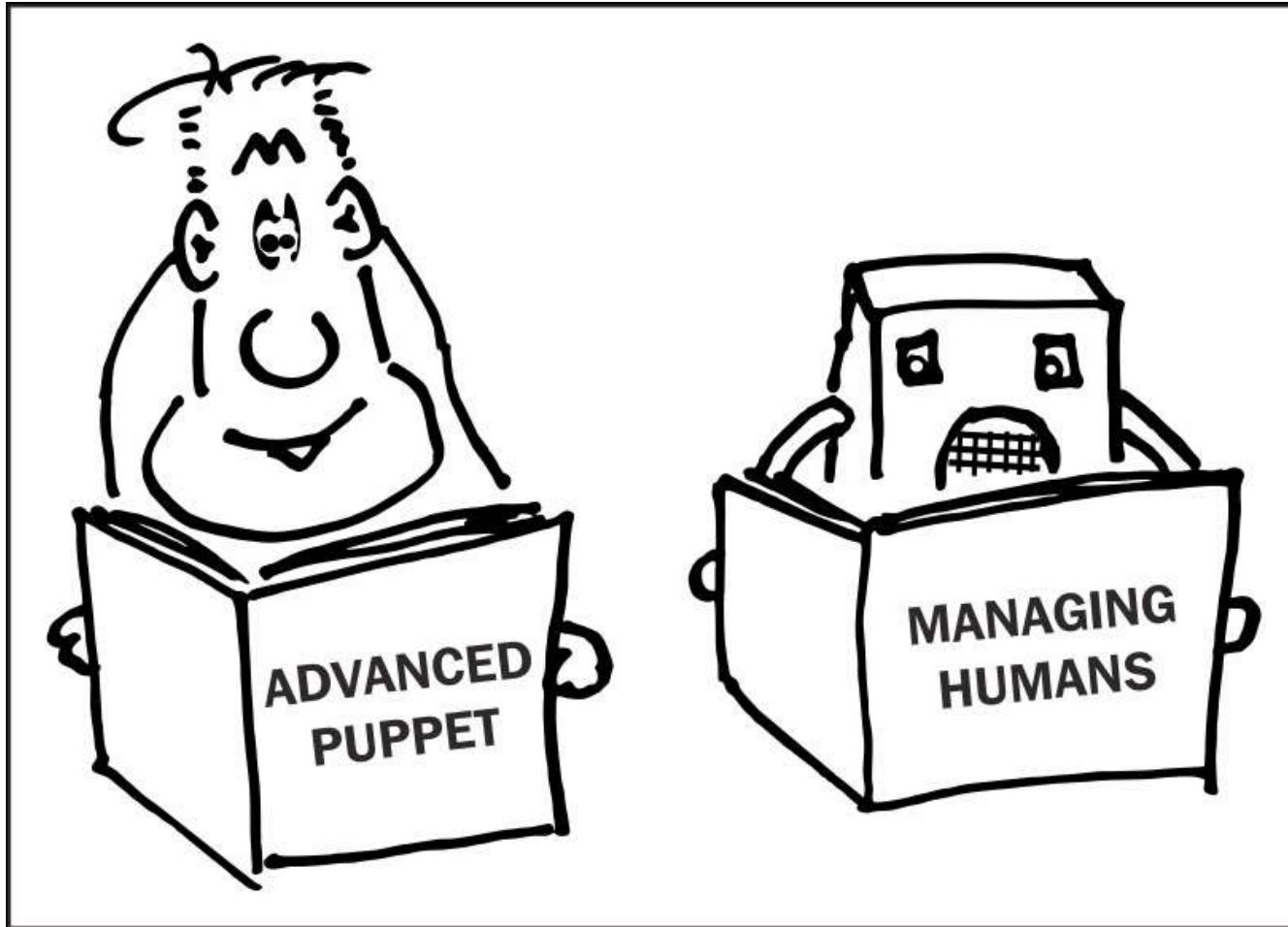
# Summary



- In this lesson we've outlined some of the problems with maintaining configuration data in Puppet manifests, and introduced Hiera as a powerful solution.
- We've seen how to configure Puppet to use the Hiera data store, and how to query Hiera keys in Puppet manifests using `lookup()`.

# 7. Mastering modules





# Using Puppet Forge modules

- A module in Puppet is a self-contained unit of shareable, reusable code, usually designed to manage one particular service or piece of software, such as the Apache web server.



# What is the Puppet Forge?

- The Puppet Forge is a public repository of Puppet modules, many of them officially supported and maintained by Puppet and all of which you can download and use.
  - You can browse the Forge at the following URL:  
<https://forge.puppet.com/>



# Finding the module you need

- The Puppet Forge home page has a search bar at the top.
- Type what you're looking for into this box, and the website will show you all the modules which match your search keywords.
- Often, there will be more than one result, so how do you decide which module to use?

# Using r10k

- In the past, many people used to download Puppet Forge modules directly and check a copy of them into their codebase, effectively forking the module repo (and some still do this).
- There are many drawbacks to this approach.
- One is that your codebase becomes cluttered with code that is not yours, and this can make it difficult to search for the code you want.



# Using r10k

- In this example, we'll use r10k to install the puppetlabs/stdlib module.
- The Puppetfile in the example repo contains a list of all the modules we'll use in this course.
- Here it is (we'll look more closely at the syntax in a moment):

Refer to the file [7\\_1.txt](#)

# Using r10k

- Run the following commands to clear out your modules/ directory, if there's anything in it (make sure you have backed up anything here you want to keep):

```
cd /etc/puppetlabs/code/environments/pbg  
sudo rm -rf modules/
```



# Using r10k

- Run the following command to have r10k process the example Puppetfile here and install your requested modules:

```
sudo r10k puppetfile install --verbose
```



# Using r10k

- To test that the stdlib module is correctly installed, run the following command:

```
sudo puppet apply --environment pbg -e  
"notice(upcase('hello'))"  
Notice: Scope(Class[main]): HELLO
```



# Understanding the Puppetfile

- The example Puppetfile begins with the following:

forge 'http://forge.puppetlabs.com'



# Understanding the Puppetfile

- There follows a group of lines beginning with mod:

```
mod 'garethr/docker', '5.3.0'  
mod 'puppet/archive', '1.3.0'  
mod 'puppet/staging', '2.2.0'
```

...



# Managing dependencies with generate-puppetfile

- Run the following command to install the generate-puppetfile gem:

```
sudo gem install generate-puppetfile
```



# Managing dependencies with generate-puppetfile

- Run the following command to generate the Puppetfile for a list of specified modules (list all the modules you need on the command line, separated by spaces):

Refer to the file 7\_2.txt



# Managing dependencies with generate-puppetfile

- Run the following command to generate a list of updated versions and dependencies for an existing Puppetfile:

```
generate-puppetfile -p  
/etc/puppetlabs/code/environments/pbg/Puppetfile
```

# Using modules in your manifests

- If you've previously followed the steps in the Using r10k section, the required module will already be installed. If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg  
sudo r10k puppetfile install
```

# Using modules in your manifests

- Run the following command to apply the manifest:

Refer to the file 7\_3.txt



# Using modules in your manifests

- Let's take a look at the example manifest (module\_mysql.pp).
- The first part installs the MySQL server itself, by including the class mysql::server:

```
# Install MySQL and set up an example database
include mysql::server
```

# Using modules in your manifests

- In this example, our parameters are set in the example Hiera data file (/etc/puppetlabs/code/environments/pbg/data/common.yaml):

```
mysql::server::root_password: 'hairline-quotient-inside-tableful'
```

```
mysql::server::remove_default_accounts: true
```

# Using modules in your manifests

- Next comes a resource declaration:

```
mysql::db { 'cat_pictures':  
  user    => 'greebo',  
  password => 'tabby',  
  host    => 'localhost',  
  grant   => ['SELECT', 'UPDATE'],  
}
```



# Using puppetlabs/apache

- Here's an example manifest which uses the apache module to create a simple virtual host serving an image file (module\_apache.pp):

Refer to the file 7\_4.txt



# Using puppetlabs/apache

- If you've previously followed the steps in the Using r10k section, the required module will already be installed.
- If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg  
sudo r10k puppetfile install
```



# Using puppetlabs/apache

- Run the following command to apply the manifest:

```
sudo puppet apply --environment=pbg  
/examples/module_apache.pp
```



# Using puppetlabs/apache



# Using puppetlabs/apache

- It starts with the include declaration which actually installs Apache on the server (module\_apache.pp):

include apache



# Using puppetlabs/apache

- There are many parameters you could set for the apache class, but in this example, we only need to set one, and as with the other examples, we set it using Hiera data in the example Hiera file:

apache::default\_vhost: false



# Using puppetlabs/apache

- Next comes a resource declaration for an apache::vhost resource, which creates an Apache virtual host or website.

```
apache::vhost { 'cat-pictures.com':  
    port      => '80',  
    docroot   => '/var/www/cat-pictures',  
    docroot_owner => 'www-data',  
    docroot_group => 'www-data',  
}
```

# Using puppetlabs/apache



- Finally, we create an index.html file to add some content to the website, in this case, an image of a happy cat.

```
file { '/var/www/cat-pictures/index.html':  
  content => "<img  
    src='http://fenagoconsulting.com/files/happycat.jpg'>",  
  owner  => 'www-data',  
  group  => 'www-data',  
}
```

# Using puppet/archive

- If you've previously followed the steps in the Using r10k section, the required module will already be installed.
- If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg  
sudo r10k puppetfile install
```



# Using puppet/archive

- Run the following command to apply the manifest:

```
sudo puppet apply --environment=pbg  
/examples/module_archive.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in environment  
production in 2.50 seconds
```

```
Notice: /Stage[main]/Main/Archive[/tmp/wordpress.tar.gz]/ensure:  
download archive from https://wordpress.org/latest.tar.gz to  
/tmp/wordpress.tar.gz and extracted in /var/www with cleanup
```

# Using puppet/archive

- Let's look at the example in detail to see how it works  
(module\_archive.pp):

```
archive { '/tmp/wordpress.tar.gz':  
    ensure      => present,  
    extract     => true,  
    extract_path => '/var/www',  
    source      => 'https://wordpress.org/latest.tar.gz',  
    creates     => '/var/www/wordpress',  
    cleanup     => true,  
}
```

# Exploring the standard library

- If you've previously followed the steps in the Using r10k section, the required module will already be installed.
- If not, run the following commands to install it:

```
cd /etc/puppetlabs/code/environments/pbg  
sudo r10k puppetfile install
```

# Safely installing packages with ensure\_packages

- As you know, you can install a package using the package resource, like this (package.pp):

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



# Safely installing packages with ensure\_packages

- But what happens if you also install the same package in another class in a different part of your manifest? Puppet will refuse to run, with an error like this:

```
Error: Evaluation Error: Error while evaluating a  
Resource Statement, Duplicate declaration:  
Package[cowsay] is already declared in file  
/examples/package.pp:1; cannot redeclare at  
/examples/package.pp:4 at /examples/package.pp:4:1  
on node ubuntu-xenial
```

# Safely installing packages with ensure\_packages

- The standard library provides this facility in the `ensure_packages()` function.
- Call `ensure_packages()` with an array of package names, and they will be installed if they are not already declared elsewhere (`package_ensure.pp`):  
`ensure_packages(['cowsay'])`
- To apply this example, run the following command:  
`sudo puppet apply --environment=pbg  
/examples/package_ensure.pp`

# Safely installing packages with ensure\_packages

- If you need to pass additional attributes to the package resource, you can supply them in a hash as the second argument to ensure\_packages(), like this (package\_ensure\_params.pp):

```
ensure_packages(['cowsay'],
{
  'ensure' => 'latest',
}
)
```



# Modifying files in place with file\_line

- Here's an example of using the `file_line` resource to add a single line to a system config file (`file_line.pp`):

```
file_line { 'set ulimits':  
    path => '/etc/security/limits.conf',  
    line => 'www-data        -      nofile      32768',  
}  
}
```

# Modifying files in place with file\_line

- You can also use `file_line` to find and modify an existing line, using the `match` attribute (`file_line_match.pp`):

```
file_line { 'adjust ulimits':  
    path => '/etc/security/limits.conf',  
    line => 'www-data      -      nofile      9999',  
    match => '^www-data .* nofile',  
}
```

# Modifying files in place with file\_line

- You can also use `file_line` to delete a line in a file if it is present (`file_line_absent.pp`):

```
file_line { 'remove dash from valid shells':  
    ensure      => absent,  
    path        => '/etc/shells',  
    match       => '^/bin/dash',  
    match_for_absence => true,  
}
```



# Introducing some other useful functions

- The grep() function will search an array for a regular expression and return all matching elements (grep.pp):

```
$values = ['foo', 'bar', 'baz']
notice(grep($values, 'ba.*'))
```

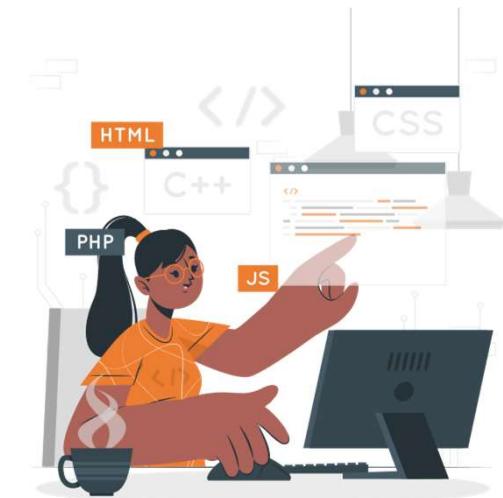
```
# Result: ['bar', 'baz']
```



# Introducing some other useful functions

- The member() and has\_key() functions return true if a given value is in the specified array or hash, respectively (member\_has\_key.pp):

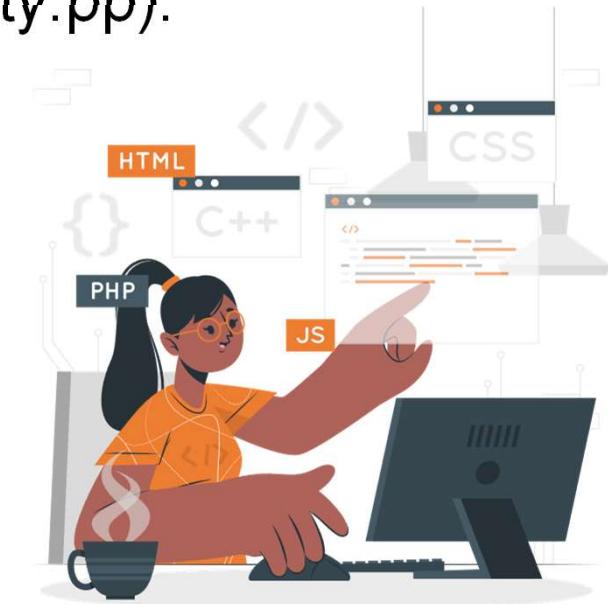
Refer to the file 7\_5.txt



# Introducing some other useful functions

- The `empty()` function returns true if its argument is an empty string, array, or hash (`empty.pp`):

```
notice(empty(""))
# Result: true
notice(empty([]))
# Result: true
notice(empty("{}"))
# Result: true
```



# Introducing some other useful functions

- The `join()` function joins together the elements of a supplied array into a string, using a given separator character or string (`join.pp`):

```
$values = ['1', '2', '3']
notice(join($values, '... '))
```

```
# Result: '1... 2... 3'
```



# Introducing some other useful functions

- The `pick()` function is a neat way to provide a default value when a variable happens to be empty.
- It takes any number of arguments and returns the first argument which is not undefined or empty (`pick.pp`):

```
$remote_host = "  
notice(pick($remote_host, 'localhost'))
```

```
# Result: 'localhost'
```



# Introducing some other useful functions

- If that data is in YAML format, you can use the `loadyaml()` function to read and parse it into a native Puppet data structure (`loadyaml.pp`):

```
$db_config = loadyaml('/examples/files/database.yml')
notice($db_config['development']['database'])
```

```
# Result: 'dev_db'
```

# Introducing some other useful functions

- The `dirname()` function is very useful if you have a string path to a file or directory and you want to reference its parent directory, for example to declare it as a Puppet resource (`dirname.pp`):

```
$file = '/var/www/vhosts/mysite'  
notice(dirname($file))
```

```
# Result: '/var/www/vhosts'
```



# The pry debugger

- Printing out the values of variables and data structures with notice() can help as can running puppet apply -d to see detailed debug output
- But if all else fails, you can use the standard library's pry() method to enter an interactive debugger session (pry.pp):

pry()



# The pry debugger

- When you apply the manifest, Puppet will start an interactive Pry shell at the point where the `pry()` function is called.
- You can then run the `catalog` command to inspect Puppet's catalog, which contains all the resources currently declared in your manifest:

**Refer to the file 7\_6.txt**

# Writing your own modules

- In this section, we'll develop a module of our own to manage the NTP service, familiar to most system administrators as the easiest way to keep server clocks synchronized with the Internet time standard.
- (Of course, it's not necessary to write your own module for this because a perfectly good one exists on Puppet Forge.)
- But we'll do so anyway, for learning purposes.)

# Writing the module code

- Run the following commands to create the manifests and files subdirectories:

```
cd pbg_ntp  
mkdir manifests  
mkdir files
```



# Writing the module code

- Create the file manifests/init.pp with the following contents:

[\*\*Refer to the file 7\\_7.txt\*\*](#)

- Create the file files/ntp.conf with the following contents:

[\*\*Refer to the file 7\\_8.txt\*\*](#)



# Writing the module code

```
git add manifests/ files/  
git commit -m 'Add module manifest and config file'  
[master f45dc50] Add module manifest and config file  
 2 files changed, 29 insertions(+)  
 create mode 100644 files/ntp.conf  
 create mode 100644 manifests/init.pp  
git push origin master
```



# Writing the module code

- Notice that the source attribute for the ntp.conf file looks like the following:

puppet:///modules/pbg\_ntp/ntp.conf



# Creating and validating the module metadata

- Every Puppet module should have a file in its top-level directory named `metadata.json`, which contains helpful information about the module that can be used by module management tools, including Puppet Forge.
- Create the file `metadata.json` with the following contents (use your own name and GitHub URLs):  
**Refer to the file 7\_9.txt**

# Creating and validating the module metadata

- Run the following commands to install metadata-json-lint and check your metadata:

```
sudo gem install metadata-json-lint  
metadata-json-lint metadata.json
```



# Creating and validating the module metadata

- Run the following commands to add, commit, and push your metadata file to GitHub:

```
git add metadata.json  
git commit -m 'Add metadata.json'  
git push origin master
```



# Tagging your module

- For the first release of your module, which according to the metadata is version 0.1.1, run the following commands to create and push the release tag:

```
git tag -a 0.1.1 -m 'Release 0.1.1'  
git push origin 0.1.1
```



# Installing your module

- Add the following mod statement to your Puppetfile (using your GitHub URL instead of mine):

```
mod 'pbg_ntp',  
  :git => 'https://github.com/fenago/pbg_ntp.git',  
  :tag => '0.1.1'
```



# Installing your module

- Because the module also requires puppetlabs/stdlib, add this mod statement too:  
`mod 'puppetlabs/stdlib', '4.17.0'`
- Now install the module in the normal way with r10k:  
`sudo r10k puppetfile install --verbose`

# Applying your module

- Now that you've created, uploaded, and installed your module, we can use it in a manifest:

```
sudo puppet apply --environment=pbg -e 'include  
pbg_ntp'
```



# Summary

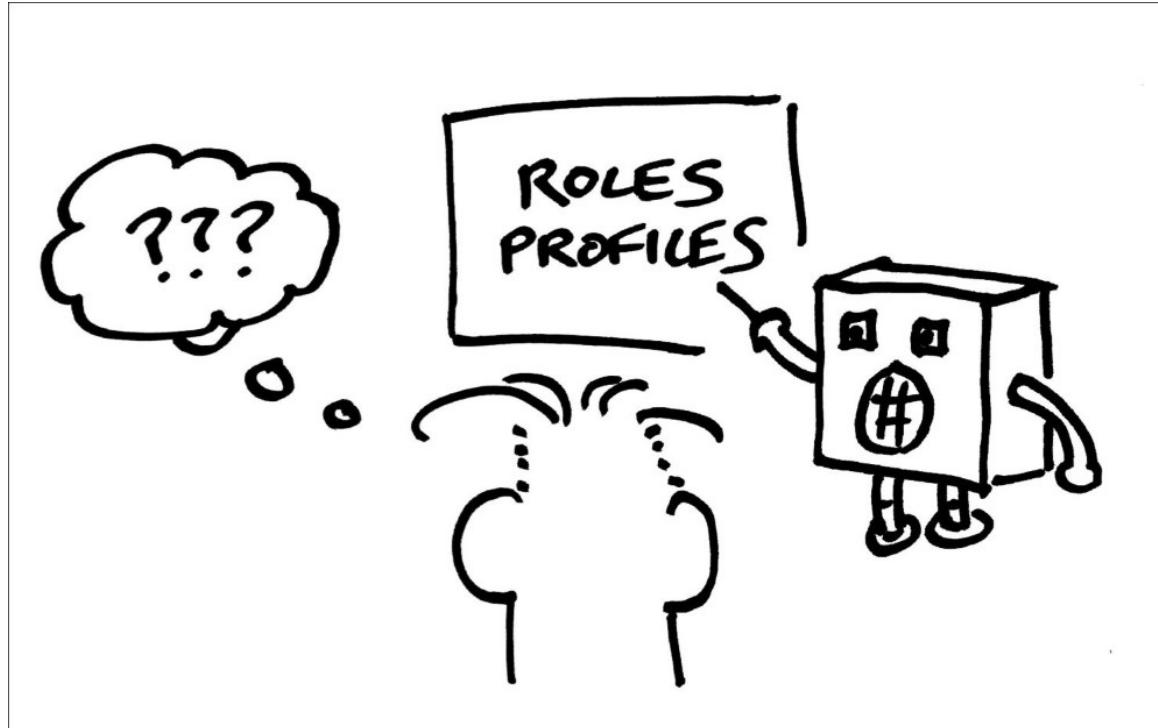


- In this lesson, we've gained an understanding of Puppet modules, including an introduction to the Puppet Forge module repository.
- We've seen how to search for the modules we need and how to evaluate the results, including Puppet Approved and Puppet Supported modules, operating system support, and download count.

# 8. Classes, roles, and profiles



# Classes, roles, and profiles



# Classes

## The class keyword

- You may have noticed that in the code for our example NTP module in lesson 7, Mastering modules (in the Writing the module code section), we used the class keyword:

```
class pbg_ntp {  
    ...  
}
```

# The class keyword

- You can then use this name elsewhere to tell Puppet to apply all the resources in the class together.
- We declared our example module by using the include keyword:

`include ntp`

- The following example shows a class definition, which makes the class available to Puppet, but does not (yet) apply any of its contained resources:

`class CLASS_NAME {`

`...`

`}`

# The class keyword

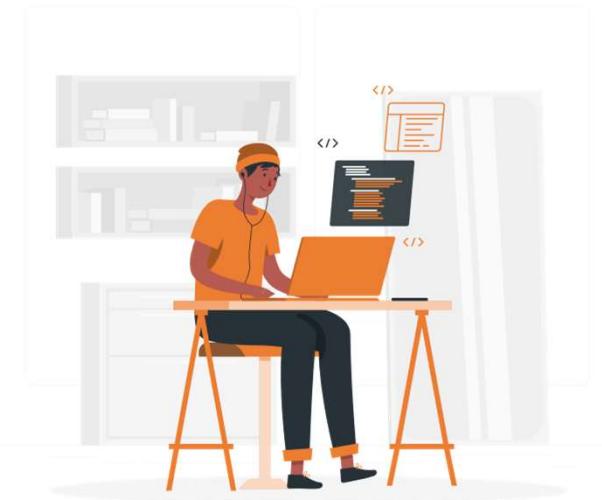
- The following example shows a declaration of the CLASS\_NAME class.
- A declaration tells Puppet to apply all the resources in that class (and the class must have already been defined):

```
include CLASS_NAME
```

# Declaring parameters to classes

- The following example shows how to define a class that takes parameters.
- It's a simplified version of the pbg\_ntp class we developed for our NTP module (class\_params.pp):

```
# Manage NTP
class pbg_ntp_params (
    String $version = 'installed',
) {
    ensure_packages(['ntp'],
    {
        'ensure' => $version,
    }
)
}
```



# Declaring parameters to classes

- The important part to look at is in parentheses after the start of the class definition.
- This specifies the parameters that the class accepts:

String \$version = 'installed',

# Declaring parameters to classes

- When you declare this class, you do it in exactly the same way that we did previously with the Puppet Forge modules, using the include keyword and the name of the class:

```
include pbg_ntp_params
```

# Declaring parameters to classes

- There are no mandatory parameters for this class, so you need not supply any, but if you do, add a value like the following to your Hiera data, and Puppet will look it up automatically when the class is included:

```
pbg_ntp_params::version: 'latest'
```

# Declaring parameters to classes

- Classes can take more than one parameter, of course, and the following (contrived) example shows how to declare multiple parameters of various types (`class_params2.pp`):

Refer to the file `8_1.txt`



# Declaring parameters to classes

- To pass parameters to this class, add Hiera data like the following:

```
pbg_ntp_params2::start_at_boot: true  
pbg_ntp_params2::version: 'latest'  
pbg_ntp_params2::service_state: 'running'
```



# Declaring parameters to classes

- Let's look closely at the parameter list:

```
Boolean $start_at_boot,  
String[1] $version = 'installed',  
Enum['running', 'stopped'] $service_state = 'running',
```



# Declaring parameters to classes

- In our example, if you try to declare the `pbg_ntp_params2` class and pass the value `bogus` to the `$service_state` parameter, you'll get this error:

Error: Evaluation Error: Error while evaluating a Resource Statement, Class[Pbg\_ntp\_params2]: parameter 'service\_state' expects a match for Enum['running', 'stopped'], got String at /examples/class\_params2.pp:22:1 on node ubuntu-xenial

# Automatic parameter lookup from Hiera data

- If we include a class named ntp, which accepts a parameter version, and a key exists in Hiera named ntp::version, its value will be passed to the ntp class as the value of version.
- For example, if the Hiera data looks like the following:

```
ntp::version: 'latest'
```

# Parameter data types

- You should always specify types for your class parameters, as it makes it easier to catch errors where the wrong parameters or values are being supplied to the class.
- If you're using a String parameter, for example, if possible, make it an Enum parameter with an exact list of the values your class accepts.

# Defined resource types

- A defined resource type definition looks a lot like a class (`defined_resource_type.pp`):

Refer to the file `8_2.txt`



# Defined resource types

- The type is called `user_with_key`, and once it's defined, we can declare as many instances of it as we want, just like any other Puppet resource:

```
user_with_key { 'john':  
  key_type => 'ssh-rsa',  
  key      => 'AAAA...AcZik=',  
}
```



# Type aliases

```
type ServiceState = Enum['running', 'stopped']
define myservice(ServiceState $state) {
    service { $name:
        ensure => $state,
    }
}
myservice { 'ntp':
    state => 'running',
}
```



# Type aliases

- Creating a type alias can be very useful when you want to ensure, for example, that parameter values match a complex pattern, which would be tiresome to duplicate.
- You can define the pattern in one place and declare multiple parameters of that type (type\_alias\_pattern.pp):

Refer to the file 8\_3.txt



# Managing classes with Hiera

- In lesson 3, Managing your Puppet code with Git, we saw how to set up your Puppet repo on multiple nodes and auto-apply the manifest using a cron job and the run-puppet script.
- The run-puppet script runs the following commands:

```
cd /etc/puppetlabs/code/environments/production && git  
pull/opt/puppetlabs/bin/puppet apply manifests/
```

# Using include with lookup()

- Previously, when including classes in our manifest, we've used the `include` keyword with a literal class name, as in the following example:

```
include postgresql  
include apache
```

- However, `include` can also be used as a function, which takes an array of class names to include:

```
include(['postgresql', 'apache'])
```

# Using include with lookup()

- We already know that we can use Hiera to return different values for a query based on the node name (or anything else defined in the hierarchy), so let's define a suitable array in Hiera data, as in the following example:

classes:

- postgresql
- apache



# Using include with lookup()

- Now we can simply use `lookup()` to get this Hiera value, and pass the result to the `include()` function:

```
include(lookup('classes'), Array[String], 'unique')
```

# Common and per-node classes

- Some classes will only be needed on particular nodes. Add these to the per-node Hiera data file.
- For example, our pbg environment on the Vagrant box contains the following in `hiera.yaml`:
  - name: "Host-specific data"  
path: "nodes/%{facts.hostname}.yaml"



# Common and per-node classes

Let's see a complete example.

- Suppose your common.yaml file contains the following:

classes:

- postgresql
- apache



# Common and per-node classes

- And suppose your per-node file (nodes/node1.yaml) also contains:

classes:

- tomcat
- my\_app

- Now, what happens when you apply the following manifest in manifests/site.pp on node1?

```
include(lookup('classes'), Array[String], 'unique')
```

# Common and per-node classes

- Which classes will be applied? You may recall from lesson 6, Managing data with Hiera that the unique merge strategy finds all values for the given key throughout the hierarchy, merges them together, and returns them as a flattened array, with duplicates removed.
- So the result of this `lookup()` call will be the following array:

[apache, postgresql, tomcat, my\_app]

# Roles and profiles

- For example, consider the following list of included classes for a certain node:

classes:

- postgresql
- apache
- java
- tomcat
- my\_app



# Roles

```
# Be an app server
class role::app_server {
    include postgresql
    include apache
    include java
    include tomcat
    include my_app
}
```



# Roles

- The value of classes in Hiera is now reduced to just the following:

classes:

- role::app\_server



# Profiles

- Let's look at a role class such as role::app\_server. It contains lots of lines including modules, like the following:

include tomcat



# Profiles

- Let's rewrite the `app_server` role to include profiles, instead of modules (`role_app_server_profiles.pp`):

```
# Be an app server
class role::app_server {
  include profile::postgresql
  include profile::apache
  include profile::java
  include profile::tomcat
  include profile::my_app
}
```



# Profiles

- The profile::tomcat class might look something like the following example, adapted from a real production manifest (profile\_tomcat.pp):

Refer to the file 8\_4.txt



# Summary

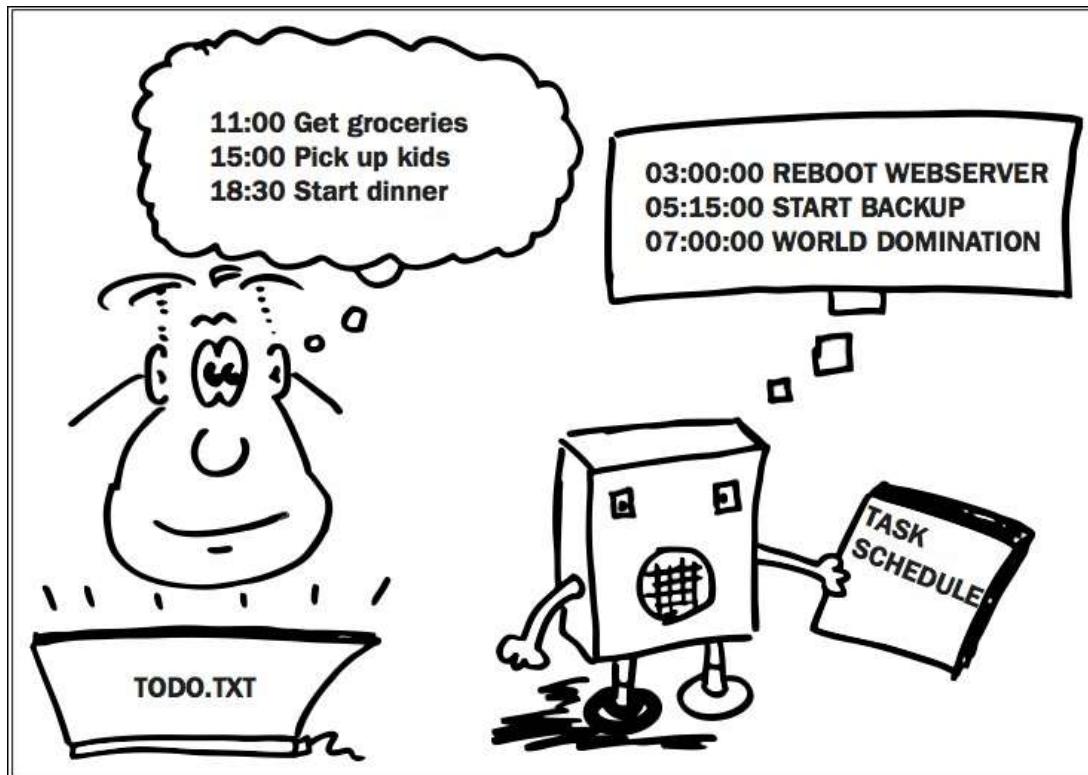
- In this lesson, we've looked at a range of different ways of organizing your Puppet code.
- We've covered classes in detail, explaining how to define them using the class keyword to define a new class, using the include keyword to declare the class
- Using Hiera's automatic parameter lookup mechanism to supply parameters for included classes.



# 9. Managing files with templates



# Managing files with templates



# The dynamic data problem

- To see why this is a problem, consider a common Puppet file management task such as a backup script.
- There are a number of site- and node-specific things the backup script needs to know: the local directories to back up, the destination to copy them to, and any credentials needed to access the backup storage.
- While we could insert these into the script as literal values, this is rather inflexible.

# Puppet template syntax

- Puppet's template mechanism is one way to achieve this.
- A template is simply an ordinary text file, containing special placeholder markers which Puppet will replace with the relevant data values.
- The following example shows what these markers look like (`aws_credentials.epp`):

```
aws_access_key_id = <%= $aws_access_key %>
```



# Puppet template syntax

- For example, if the variable \$aws\_access\_key has the value AKIAIAF7V6N2PTOIZVA2, then when the template is processed by Puppet the resulting output text will look like the following:

aws\_access\_key\_id = AKIAIAF7V6N2PTOIZVA2

# Referencing template files

- Recall from lesson 2, Creating your first manifests, that you can use the content attribute to set a file's contents to a literal string:

```
file { '/tmp/hello.txt':  
  content => "hello, world\n",  
}
```



# Referencing template files

- And, of course, you can interpolate the value of Puppet expressions into that string:

```
file { "/usr/local/bin/${task}":  
    content => "echo I am ${task}\n",  
    mode   => '0755',  
}
```



# Referencing template files

- So far, so familiar, but we can take one further step and replace the literal string with a call to the `epp()` function (`file_epp.pp`):

```
file { '/usr/local/bin/backup':  
    content => epp('/examples/backup.sh.epp',  
    {  
        'data_dir' => '/examples',  
    }  
,  
    mode   => '0755',  
}
```



# Referencing template files

- The template file might look something like the following (backup.sh.epp):

```
<%- | String $data_dir | %->
#!/bin/bash
mkdir -p /backup
tar cvzf /backup/backup.tar.gz <%= $data_dir %>
```

# Referencing template files

- To reference a template file from within a module (for example, in our NTP module from lesson 7, Mastering modules), put the file in the modules/pbg\_ntp/templates/ directory, and prefix the filename with pbg\_ntp/, as in the following example:

```
file { '/etc/ntp.conf':  
    content => epp('pbg_ntp/ntp.conf.epp'),  
}
```

# Inline templates

```
$web_root = '/var/www'  
$backup_dir = '/backup/www'  
  
file { '/usr/local/bin/backup':  
    content => inline_epp('rsync -a <%= $web_root %>/  
    <%= $backup_dir %>/'),  
    mode   => '0755',  
}  
}
```

# Template tags

- The tag we've been using in the examples so far in this lesson is known as an expression-printing tag:

```
<%= $aws_access_key %>
```



# Template tags

- A non-printing tag is very similar, but will not generate any output. It has no = sign in the opening delimiter:  
`<% notice("This has no effect on the template output") %>`
- You can also use a comment tag to add text which will be removed when Puppet compiles the template:  
`<%# This is a comment, and it will not appear in the output of the template %>`

# Computations in templates

- Naturally, whatever we can do in Puppet code, we can also do in a template, so here's the same computation in template form (`template_compute.epp`):

```
innodb_buffer_pool_size=<%=$facts['memory']['system']['total_bytes'] * 3/4 %>
```



# Computations in templates

- The generated output (on my Vagrant box) is as follows:

```
sudo puppet exec render --environment pbg  
/examples/template_compute.epp  
innodb_buffer_pool_size=780257280
```



# Conditional statements in templates

- We've already met conditional statements in manifests in lesson 5, Variables, expressions, and facts, where we used them to conditionally include sets of Puppet resources (if.pp):

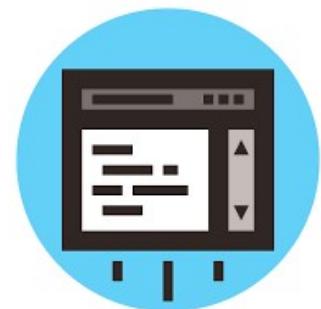
```
if $install_perl {  
    ...  
} else {  
    ...  
}
```



# Conditional statements in templates

- Here's a similar example to the previous one, but this time controlling inclusion of a block of configuration in a template (template\_if.epp):

```
<% if $ssl_enabled { -%>
## SSL directives
SSLEngine on
SSLCertificateFile    "<%= $ssl_cert %>"
SSLCertificateKeyFile  "<%= $ssl_key %>"
...
<% } -%>
```



# Conditional statements in templates

- If \$ssl\_enabled is true, the file generated by the template will contain the following:

```
## SSL directives
SSLEngine on
SSLCertificateFile    "<%= $ssl_cert %>"
SSLCertificateKeyFile "= $ssl_key %&gt;"
...</pre
```



# Iteration in templates

- If we can generate parts of a file from Puppet expressions, and also include or exclude parts of the file depending on conditions, could we generate parts of the file with a Puppet loop?
- That is to say, could we iterate over an array or hash, generating template content for each element?  
Indeed we can.

# Iterating over Facter data

- You may recall from lesson 5, Variables, expressions, and facts that in order to iterate over a hash, we use a syntax like the following:

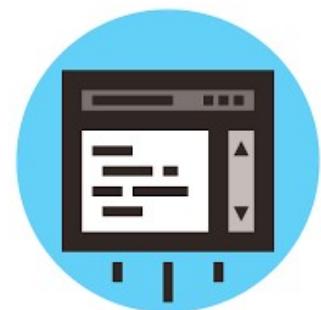
```
HASH.each | KEY, VALUE | {  
    BLOCK  
}
```



# Iterating over Facter data

- So let's apply this pattern to the Facter data and see what the output looks like (`template_iterate.epp`):

```
<% $facts['networking']['interfaces'].each |String  
$interface, Hash $attrs| { -%>  
interface <%= $interface %>;  
<% } -%>
```



# Iterating over Facter data

- Each time round the loop, the value of \$interface is set to the name of the next interface in the list, and a new output line like the following is generated:

interface em1;



# Iterating over Facter data

- Here's the final output, on a node with lots of network interfaces:

```
interface em1;  
interface em2;  
interface em3;  
interface em4;  
interface em5;  
interface lo;
```



# Iterating over structured facts

- The following example shows how this works  
(template\_iterate2.epp):

```
<% $facts['networking']['interfaces'].each |String  
$interface, Hash $attrs| { -%>  
local_address <%= $attrs['bindings'][0]['address'] %>;  
<% } -%>
```

# Iterating over structured facts

- Here's the final output:

```
local_address 10.170.81.11;  
local_address 75.76.222.21;  
local_address 204.152.248.213;  
local_address 66.32.100.81;  
local_address 189.183.255.6;  
local_address 127.0.0.1;
```



# Iterating over Hiera data

- Just as we did when generating Puppet user resources, we will make a call to `lookup()` to get the array of users, and iterate over this using `each`.
- The following example shows what this looks like in the template (`template_hiera.epp`):

```
AllowUsers<% lookup('users').each | $user | { -%>
  <%= $user -%>
<% } %>
```

# Iterating over Hiera data

- Here's the result:

AllowUsers katy lark bridget hsing-hui charles



# Passing parameters to templates

- To declare parameters for a template, list them between pipe characters (|) inside a non-printing tag, as shown in the following example (template\_params.epp):

```
<% | String[1] $aws_access_key,  
     String[1] $aws_secret_key,  
| -%>  
aws_access_key_id = <%= $aws_access_key %>  
aws_secret_access_key = <%= $aws_secret_key %>
```



# Passing parameters to templates

- The following example shows how to do this (epp\_params.pp):

```
file { '/root/aws_credentials':  
  content => epp('/examples/template_params.epp',  
  {  
    'aws_access_key' => 'AKIAIAF7V6N2PTOIZVA2',  
    'aws_secret_key' =>  
      '7IBpXjoYRVbJ/rCTVLaAMyud+i4co11IVt1Df1vt',  
  }  
),  
}
```

# Passing parameters to templates

- First, we need to declare a \$users parameter in the template (template\_hiera\_params.epp):

```
<% | Array[String] $users | -%>
AllowUsers<% $users.each | $user | { -%>
  <%= $user -%>
<% } %>
```



# Passing parameters to templates

- we pass in the Hiera data by calling `lookup()` in the parameters hash (`epp_hiera.pp`):

```
file { '/tmp/sshd_config_example':  
    content =>  
    epp('/examples/template_hiera_params.epp',  
        {  
            'users' => lookup('users'),  
        }  
    ),  
}
```



# Validating template syntax



- Fortunately, Puppet includes a tool to check and validate your templates on the command line: `puppet epp validate`.
- To use it, run the following command against your template file:

```
puppet epp validate /examples/template_params.epp
```

# Validating template syntax

- If there is no output, the template is valid. If the template contains an error, you will see an error message, something like the following:

```
Error: Syntax error at '%' at  
/examples/template_params.epp:3:4  
Error: Errors while validating epp  
Error: Try 'puppet help epp validate' for usage
```



# Rendering templates on the command line

- To use it, run the following command:

```
puppet exec render --values "{ 'aws_access_key' =>  
'foo', 'aws_secret_key' => 'bar' }"  
/examples/template_params.epp  
aws_access_key_id = foo  
aws_secret_access_key = bar
```



# Rendering templates on the command line

- Alternatively, you can use the `--values_file` argument to reference a Puppet manifest file containing the hash of parameters:

```
echo "{ 'aws_access_key' => 'foo', 'aws_secret_key' =>  
'bar' }" >params.pp
```

```
puppet epp render --values_file params.pp  
/examples/template_params.epp
```

```
aws_access_key_id = foo  
aws_secret_access_key = bar
```



# Rendering templates on the command line

- Parameters given on the command line will take priority over those from the file:

```
puppet exec render --values_file params.pp --values "{  
'aws_access_key' => 'override' }"  
/examples/template_params.epp  
aws_access_key_id = override  
aws_secret_access_key = bar
```



# Rendering templates on the command line

- You can also use puppet `epp render` to test inline template code, using the `-e` switch to pass in a literal template string:

```
puppet epp render --values "{ 'name' => 'Dave' }" -e  
'Hello, <%= $name %>'
```

```
Hello, Dave
```



# Rendering templates on the command line

- Just as when testing your manifests, you can also use `puppet apply` to test your templates directly, using a command similar to the following:

```
sudo puppet apply -e "file { '/tmp/result': content =>
epp('/examples/template_iterate.epp')}"
```



# Legacy ERB templates

- ERB template syntax looks quite similar to EPP.
- The following example is a snippet from an ERB template:

```
AllowUsers <%= @users.join(' ') %><%=  
scope['::ubuntu'] == 'yes' ? ',ubuntu' : " %>
```



# Legacy ERB templates

- This required some complicated plumbing to manage the interface between Puppet and Ruby; for example, accessing variables in non-local scope in ERB templates requires the use of the scope hash, as in the previous example.
- Similarly, in order to access Puppet functions such as `strftime()`, you have to call:

```
scope.call_function('strftime', ...)
```

# Summary



- In this lesson we've looked at one of the most powerful tools in Puppet's toolbox, the template file.
- We've examined the EPP tag syntax and seen the different kinds of tags available, including printing and non-printing tags.
- We've learned that not only can you simply insert values from variables into templates, but that you can also include or exclude whole blocks of text.