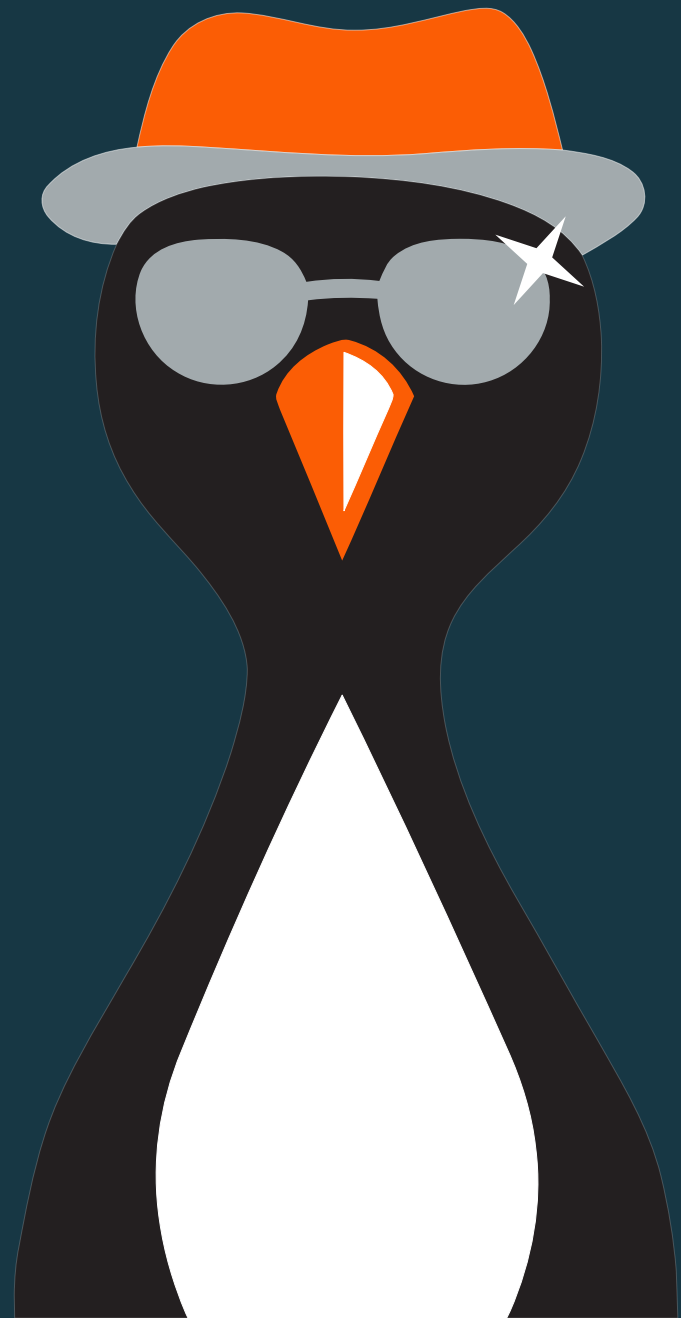
THE URBAN PENGUIN

Puppet 6 Essentials

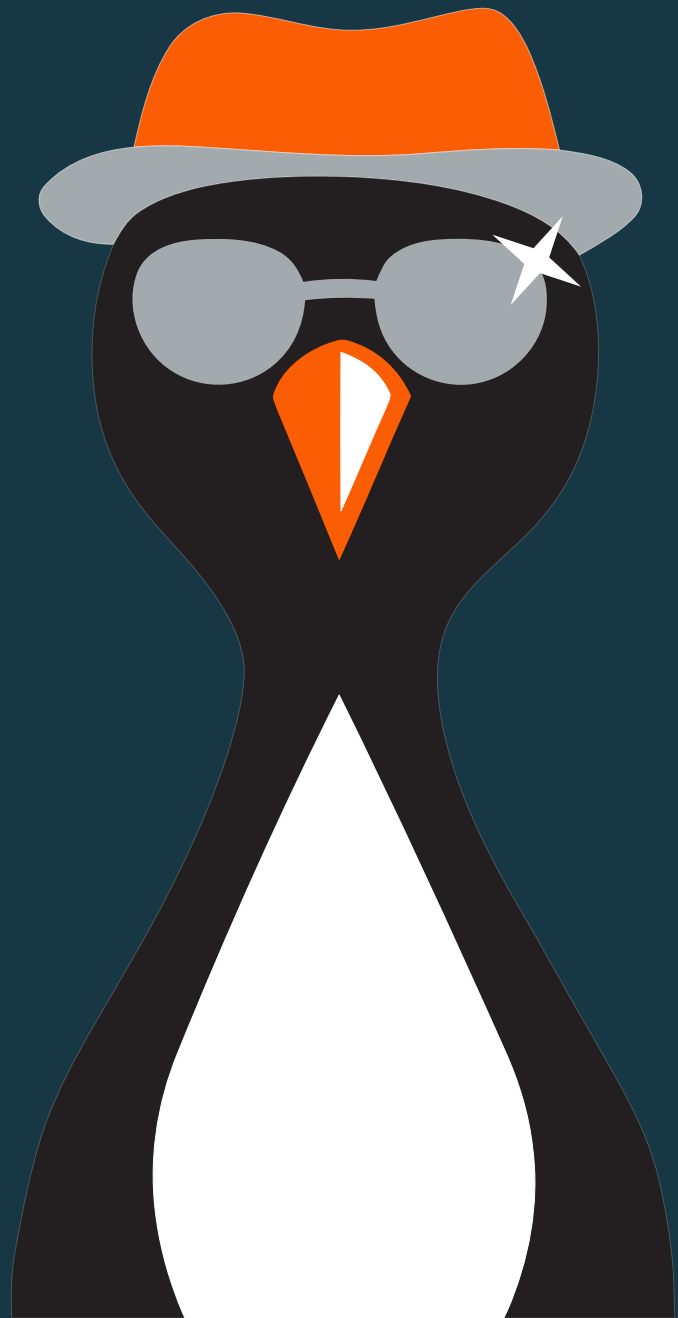
Understanding and working with the essentials elements of the Puppet 6 configuration management system



THE URBAN PENGUIN

1: Introduction

In this module we will identify the Puppet 6 Essentials course contents and what is needed for the lab systems



Contents

Puppetlabs

Puppet created in 2005 by Luke Kanies

Support for Linux, Unix, Windows

First commercial release in 2011

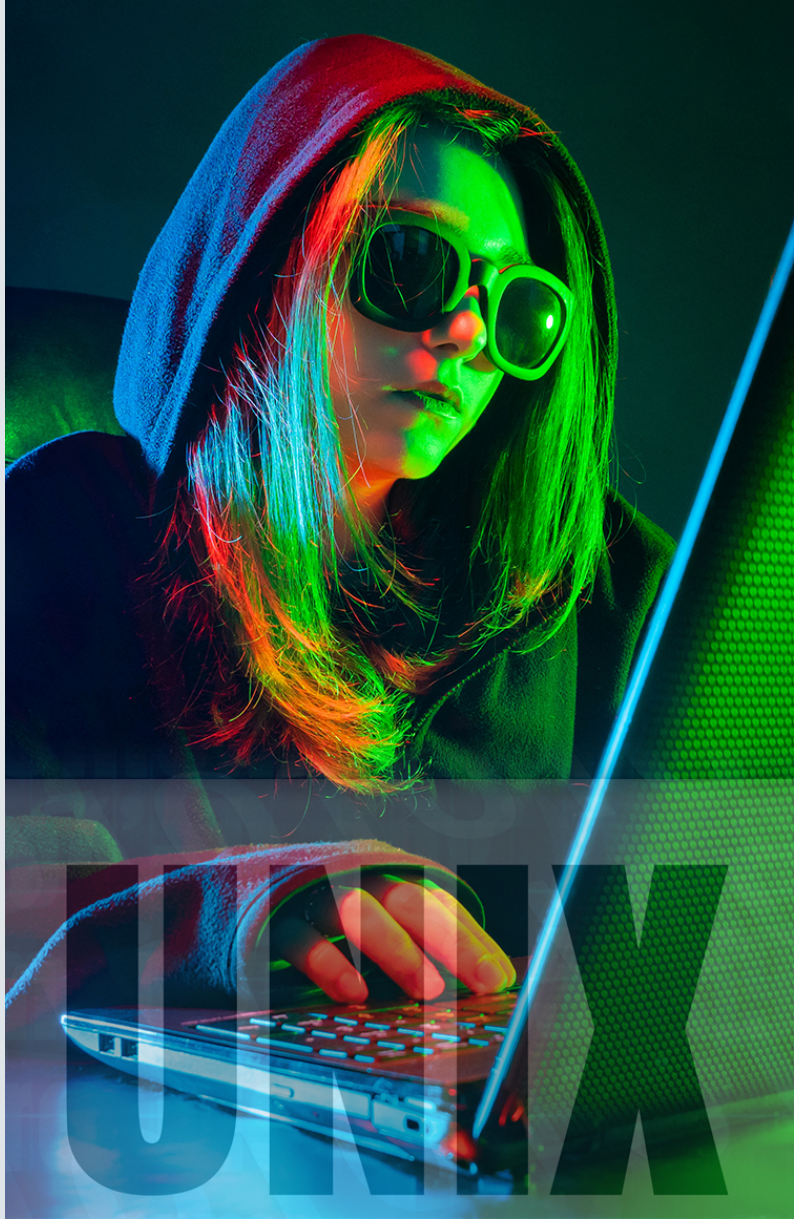
Puppet Open Source & Puppet Enterprise

Multi-Platform Puppet

Puppet does not just work on *NIX based systems you can run Puppet on windows too

For the Labs we use two AWS CentOS 8 systems, the server being t2.small and the standalone agent used towards the end is a t2.micro

You could use your own local VMs or physical devices. The Puppet Server that will run both the server and the agent should have 2GB RAM and the system that runs just the agent 1GB RAM



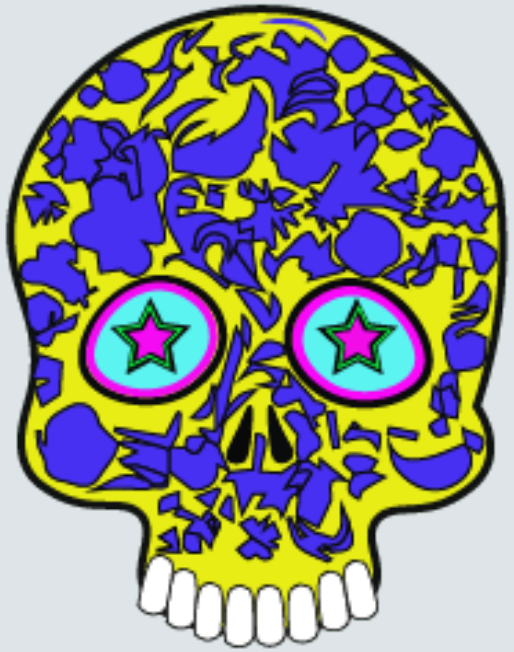
Configuration Management Systems

Common systems include:

- Puppet
- Chef
- Salt
- Ansible

They all help automate your estate. Without management one admin can look after 100 servers with management that goes to 1000 servers

Architecture



Usually Puppet uses the client server model

Can be master-less with just agent and local manifests, but not so scalable in this setup

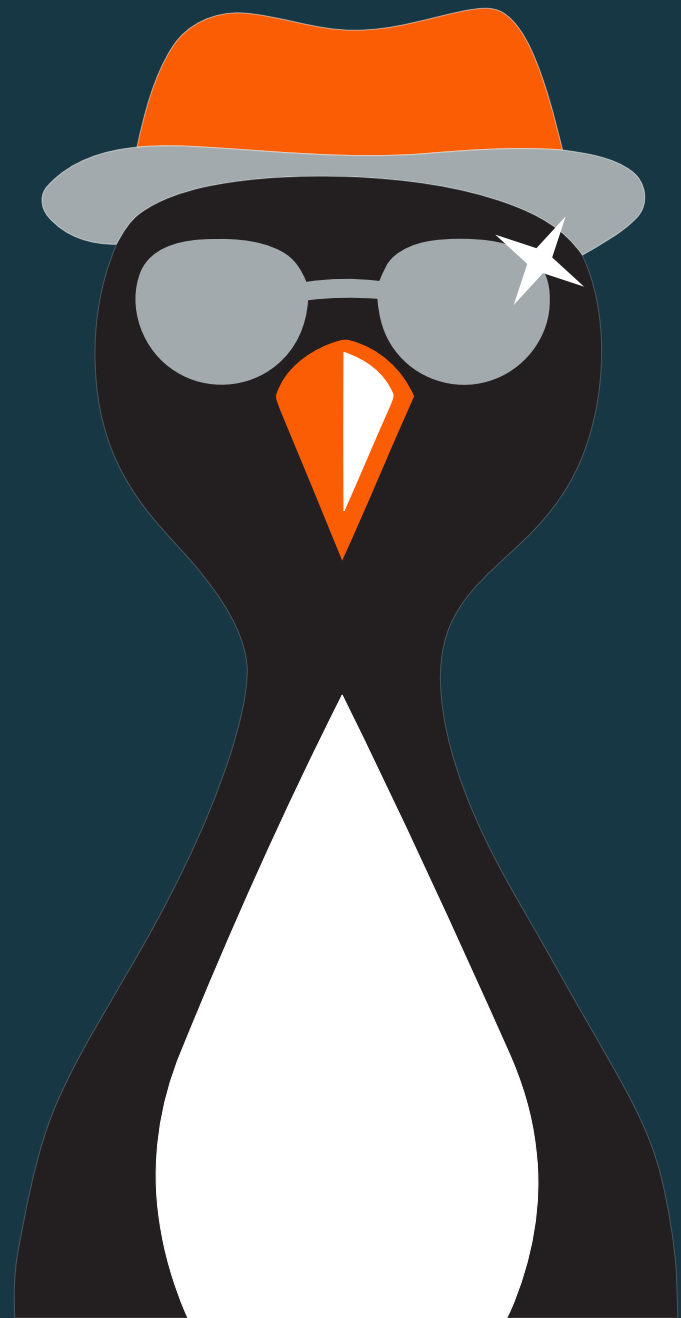
Puppet Server runs in a JVM and listens on TCP Port 8140 by default, so don't forget firewall considerations

Agents connect to the host 'puppet' by default so considerer a DNS alias, we use host entries as we only have two systems

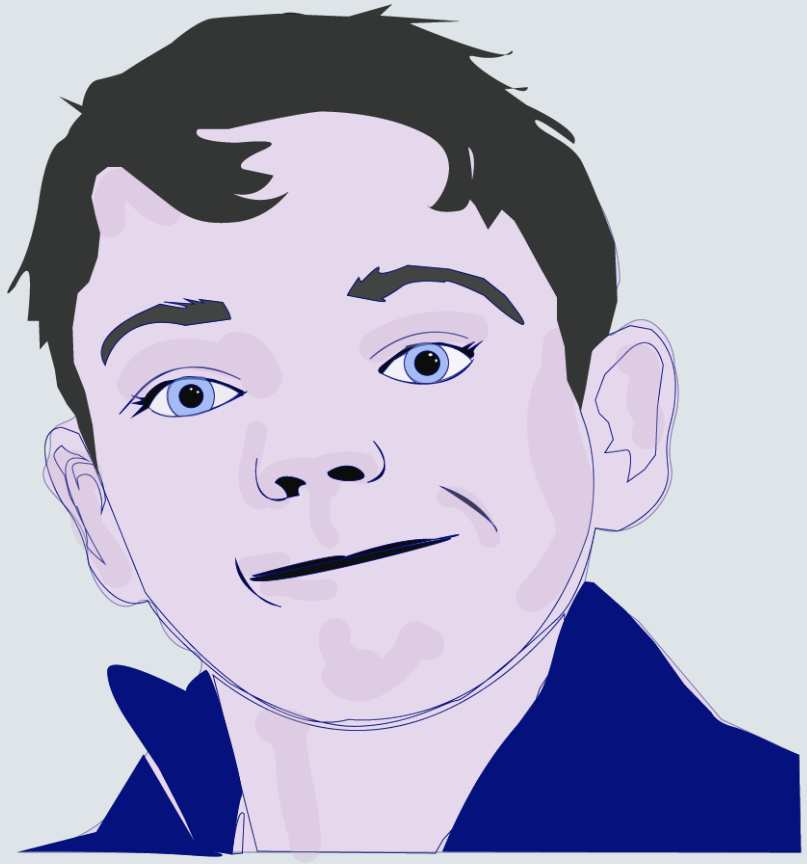
THE URBAN PENGUIN

2: Installing Puppet 6 Server and Agent

We being on the 2GB System that will act as the Puppet Server. Installing the Puppet Server will also add the Puppet Agent.



Puppet Repos



CentOS does not ship with puppet but the agent is available from the EPEL repo. We need to add the Puppet repo to get the very latest and agent and server

Ubuntu 18.04 ships with version 5 not version 6, so we are always best adding the Puppet repos from Puppetlabs

```
# yum install https://yum.puppet.com/puppet-release-el-8.noarch.rpm
# yum list --disablerepo=* --enablerepo=puppet available
```

Add and List Repo

We can easily add and list the Puppetlabs repo on CentOS 8

```
# echo "127.0.0.2 puppet" >> /etc/hosts
# yum install puppetserver
# exit ; sudo -i puppet --version
```

Installing the Server and Agent

Installing the Server will install the agent. The agent will look for a server called puppet by default. The PATH will need to be updated so we exit and sudo again

```
# puppet apply -e 'package { "chrony": ensure => installed }'
# puppet apply -e 'service { "chronyd": ensure => running , enable => true }'
```

TimeSync

Ideally, we have accurate time on the server and agents. We can test Puppet from the command line to ensure Chronyd is installed and running

```
# puppet apply -e 'service { "chronyd": ensure => running , enable => true }'
# systemctl stop chronyd
# puppet apply -e 'service { "chronyd": ensure => running , enable => true }'
```

Idempotent

Puppet, being Idempotent, allows the command to run many times and only actions if we do not meet the configuration requirement

```
# sed -i 's/2g/1g/g' /etc/sysconfig/puppetserver
# systemctl enable --now puppetserver puppet
# ss -ntl
# puppet agent -t
```

Java Memory and Firewall

By default the Puppet Server is configured to use 2GB RAM in the JVM. For our systems we can reduce it to 1GB. Normally TCP port 8140 will need to be opened in the firewall

```
# puppet module install puppetlabs/apache
# puppet apply -e "include apache"
# ss -ntl
# curl localhost
```

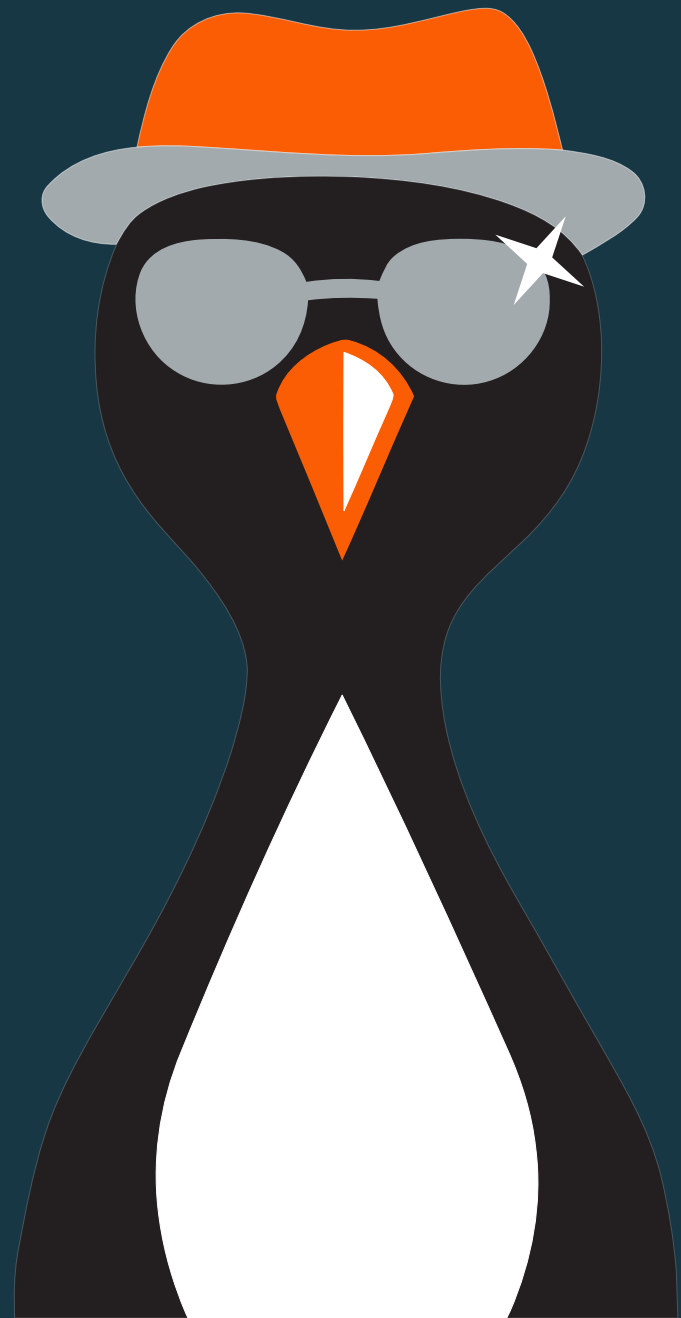
20 Second Overview of Puppet's Powers

Even without writing code we can easily make use of the Power of Puppet by installing and using a module. We look at modules more later but they have pre-written code

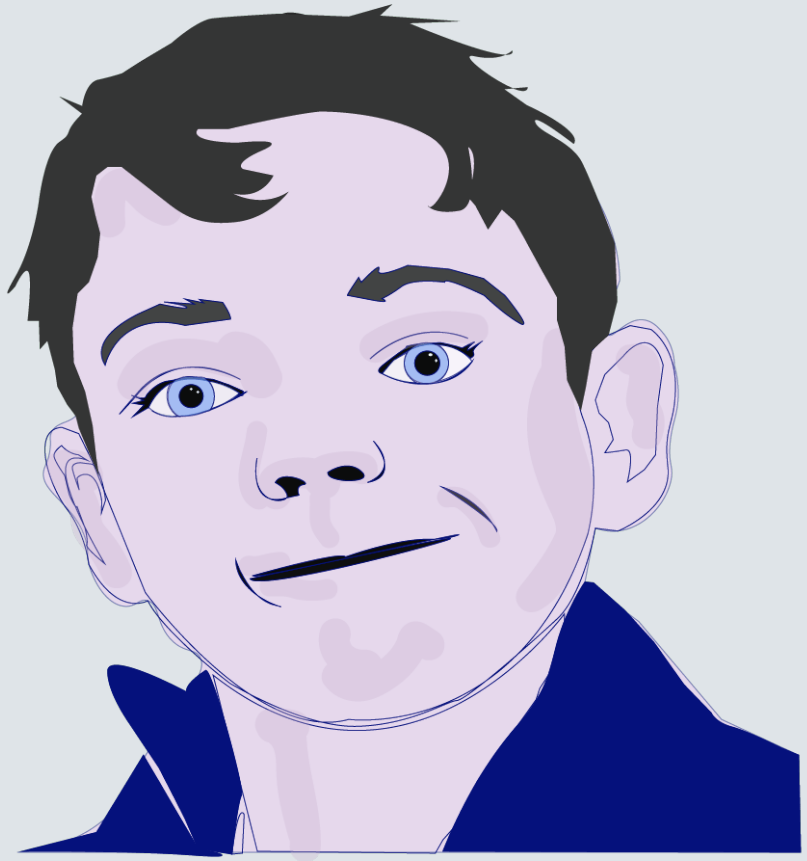
THE URBAN PENGUIN

3: Writing and Working with Puppet Manifests

We normally do not want to write everything from the command line each time we need an action to happen. Manifest become a script that can be played and replayed when we want



Manifests



Manifests allow us to save code into scripts with an extension of .pp

Manifests can be applied locally with puppet apply

For full automation manifests need to be stored on the server in the correct environment. The default environment is production

```
# puppet config print
# puppet config print config
# puppet config print manifest --section master --environment production
```

Configuration

Using the puppet config command we can view and set configuration parameters including the path to the manifest. If the manifest is a directory then each manifest is processed in alphanumeric order

```
# vim /etc/puppetlabs/code/environments/production/manifests/site.pp

notify {'Hello World':
  message => "Hello World!",
}

# puppet apply
/etc/puppetlabs/code/environments/production/manifests/site.pp

# puppet agent -t
```

Manifest Files

The puppet agent can apply local manifests by specifying the path. If the path is a directory all manifests are processed. If the agent connects to the server the agent the manifests are located by the settings of the server

```
# puppet config print runinterval

# expr 1800 / 60

# puppet agent -t
Notice: Hello World!
```

Agent Run

Be default the agent runs every 30 minutes. We can force a run from the CLI.

```
# mkdir -p /etc/puppetlabs/code/environments/dev/manifests
# puppet config set environment dev --section=agent
```

Create Environments

By default all agents work in the production environment. We can create more environments on the Puppet server and configure agents to use them

```
# alias cdpp="cd $(puppet config print manifest)"
# alias
# vim ~/.bashrc #add the line to bottom of file
alias cdpp="cd /etc/puppetlabs/code/environments/production/manifests"
# unalias cdpp
# source ~/.bashrc
# alias ; cdpp; pwd
```

BASH Aliases

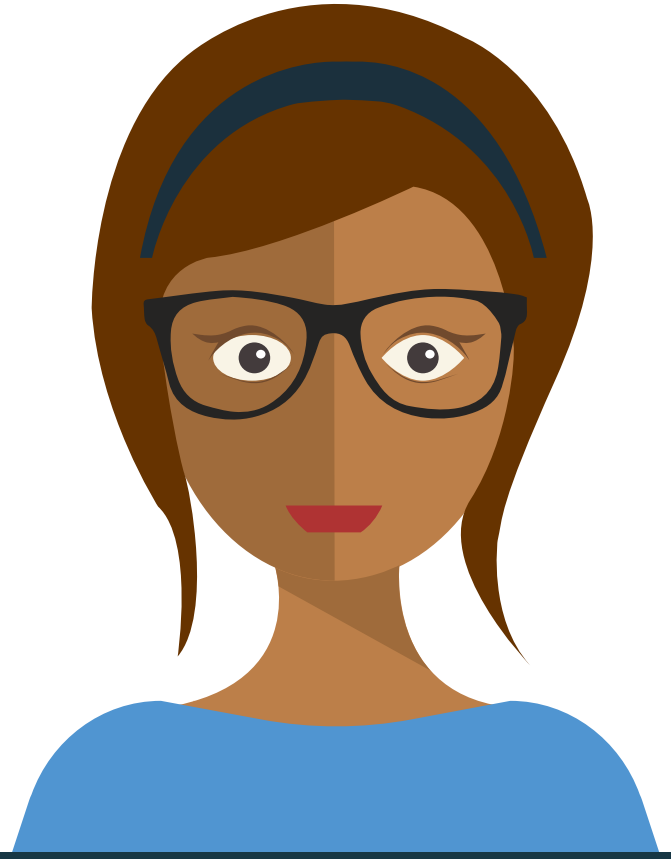
We may often move to the manifest directory. We can create an alias to make this easier

Resources

Resources are the main building block of Puppet:

- notify resource with Hello World
- package resource to ensure chrony was installed
- service resource to ensure chronyd was running

Puppet is declarative, meaning it states what it want done not how to do it. On CentOS the packaging will be done with dnf/yum, on Ubuntu with apt and Windows with chocolatly.



```
# puppet resource --type

# puppet describe service

# puppet resource service chronyd
#service should show as running and enabled

# puppet resource service atd
# at is not installed on my system so should show as stopped and disabled
```

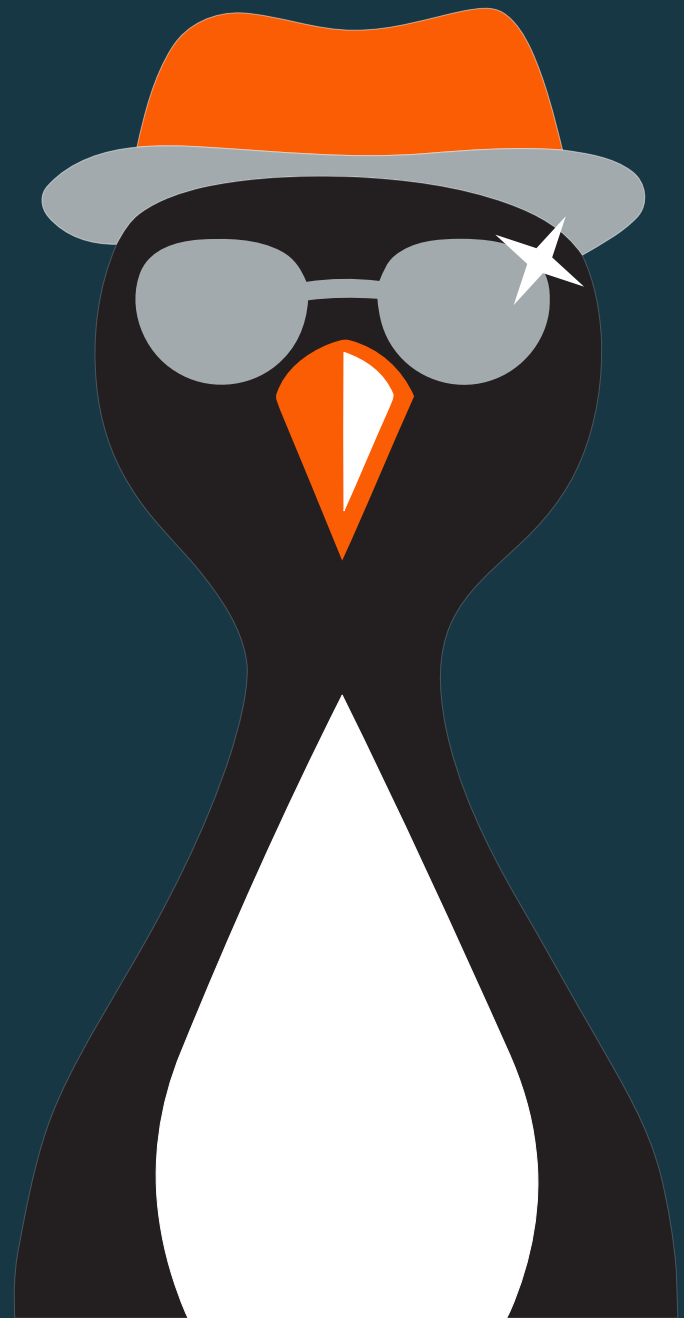
List Resources and Help

The first command list all resource types. We can gain help on a type with describe and we can print the example configuration from a resource.

THE URBAN PENGUIN

4: Extending Manifests Into Modules

We started off with a quick look at installing the Apache Web Server with a downloaded module. We now look more at modules and creating our own.



```
# puppet module list

# puppet module install puppetlabs/stdlib

# puppet module uninstall puppetlabs/stdlib

# puppet module install -i /etc/puppetlabs/code/modules puppetlabs/stdlib

# #Installing in a shared module directory makes it available across
environments

# cat /etc/puppetlabs/code/modules/stdlib/examples/file_line.pp
```

Modules

Modules are great ways to encapsulate code in to reusable lumps. Puppet forge is a great resource for modules that have been shared. The stdlib from puppetlabs is always useful

```
# cp /etc/puppetlabs/code/modules/stdlib/examples/file_line.pp ~/
# puppet apply file_line.pp
# cat /tmp/dansfile
```

Example Manifest From Module

A puppet manifest will have the extension .pp and contains code to be run on agents. We can use puppet apply to execute local manifests

```
# vim ssh.pp
service { 'sshd':
  ensure  => 'running',
  enable  => true,
  require => Package['openssh'],
}
file_line { 'root_login_ssh':
  path   => '/etc/ssh/sshd_config',
  ensure => 'present',
  line   => 'PermitRootLogin no',
  match  => '^PermitRootLogin',
  notify => Service['sshd'],
}
```

File Edits Using file_line Resource

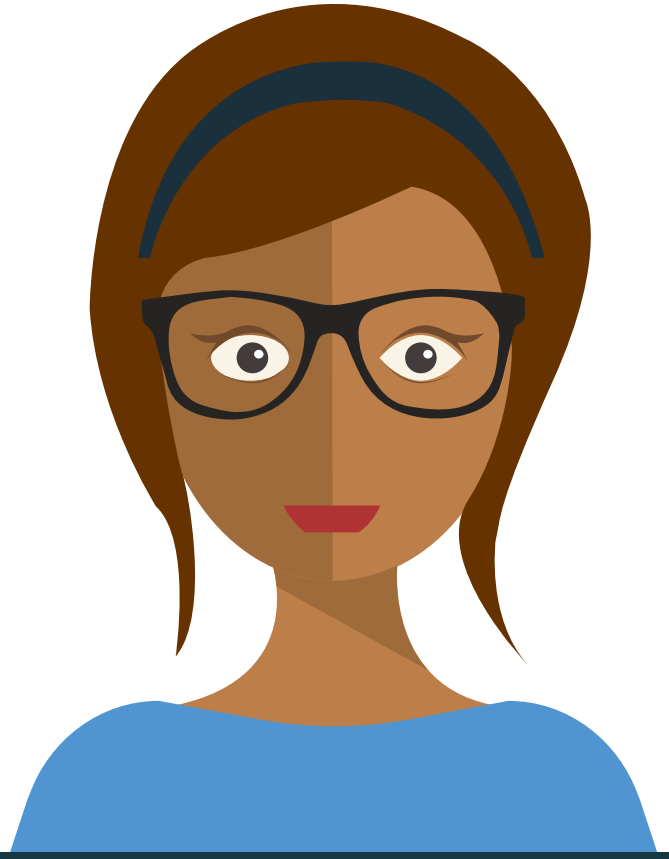
Rather than deliver a complete file we can edit the file with file_line. The file_line resource that ships with the puppetlabs/stdlib module and shares the same top level namespace. We can easily replace or add lines

The BIG Three

The 3 big resources are:

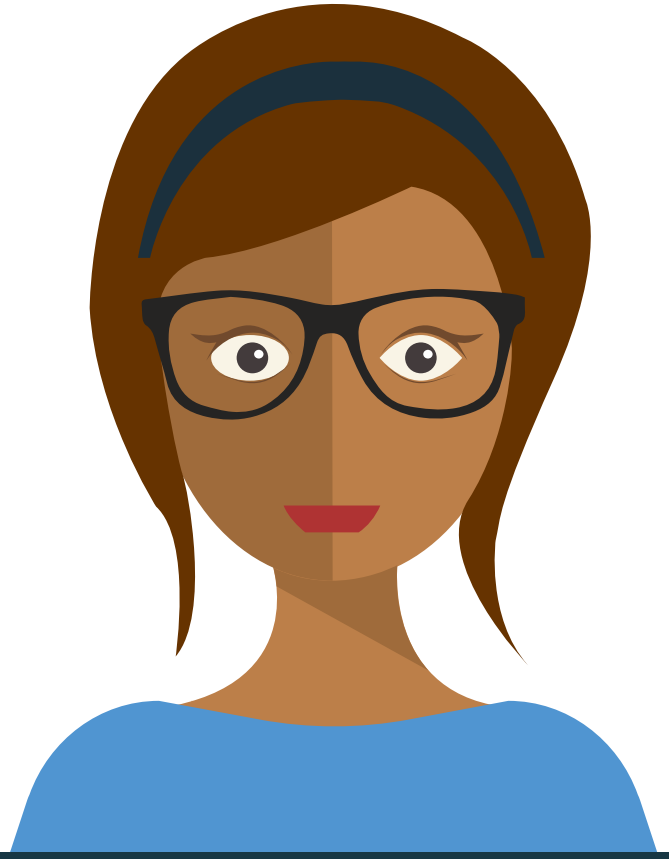
- Package
- Service
- File

Many tasks can be managed using these resources and we will now start by building our own module to manage the time sync service Chrony.



Organize Code On Server

Creating or own modules will help us reuse and orgainize code on the server. Modules can be shared or created within an environment. We only have the production environment so we will use that



```
# cd /etc/puppetlabs/code/environments/production/modules
# mkdir -p chrony/{manifests,files,examples}
# tree chrony
```

Creating The Module Structure

A simple module can be created at the command line in the correct environment

```
# vim chrony/manifests/init.pp
class chrony {
  package { 'chrony':
    ensure => 'installed',
  }
}
```

The Module Manifest

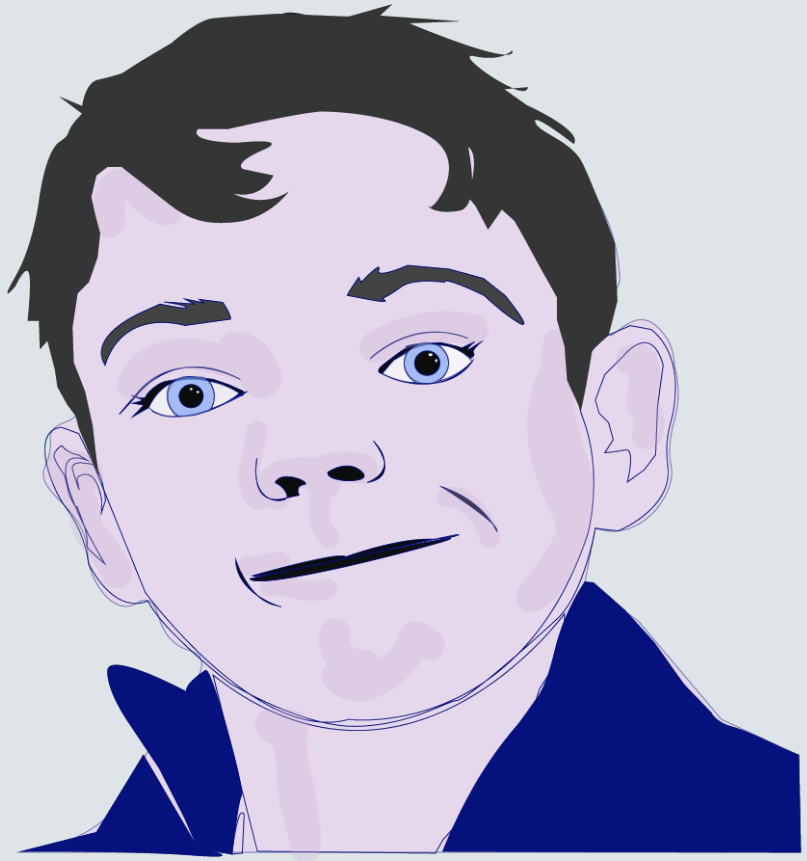
The main manifest in a module should be init.pp. It is called by the module name only. The code within module manifest make up a class definition Here we make sure the package is installed

```
# vim chrony/manifests/init.pp
class chrony {
  package { 'chrony':
    ensure => 'installed',
  }
  service { 'chronyd':
    ensure  => 'running',
    enable  => true,
    require => Package['chrony'],
  }
}
```

Service

Using the service resource we can ensure the service is running and auto-starts. We can also enforce the correct order the resources are applied.

Restart Service when File Changes



We can use notify event to ensure a service is restarted if a file changes

We can use grep to clean the empty lines and commented lines from the chrony.conf to ensure we have a consistent configuration.

Adding the new file to the chrony/files/ directory ensures that the Puppet file server can deliver the file

```
# cd /etc/puppetlabs/code/environment/production/modules

# vim chrony/manifests/init.pp
class chrony {
  package { 'chrony':
    ensure => 'installed',
  }
  service { 'chronyd':
    ensure  => 'running',
    enable  => true,
    require => Package['chrony'],
  }
  file { '/etc/chrony.conf':
    ensure => file,
    content => file('chrony/chrony.conf'),
    notify => Service['chronyd'],
  }
}
```

init.pp

The default manifest of a module should be called init.pp. Notice the delivery of the file and the path used.

```
# cdpp
# rm 02.pp
# vim 01.pp
include chrony
include apache
# puppet agent -t
# rm /etc/chrony.conf
# puppet agent -t
```

Including Modules in Manifests

We can return to our manifests and perhaps remove the message and add in the include statement to make sure that both apache and chrony modules are delivered.

```
# cdpp

# vim 01.pp
if $osfamily == 'RedHat' {
 include chrony
 include apache
}

# facter --show-legacy | grep osfamily
```

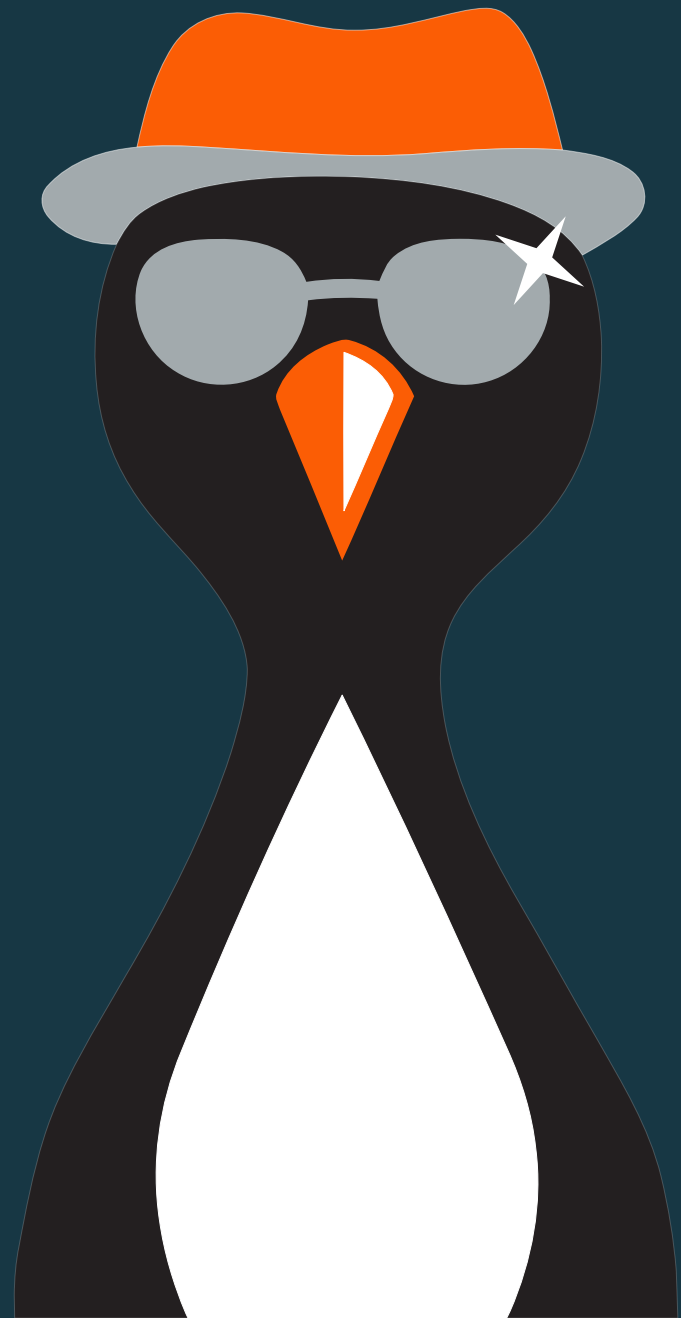
Using Conditionals

Conditional statement can help tune the manifest. These can be based on facts. Facts can be printed and researched with facter

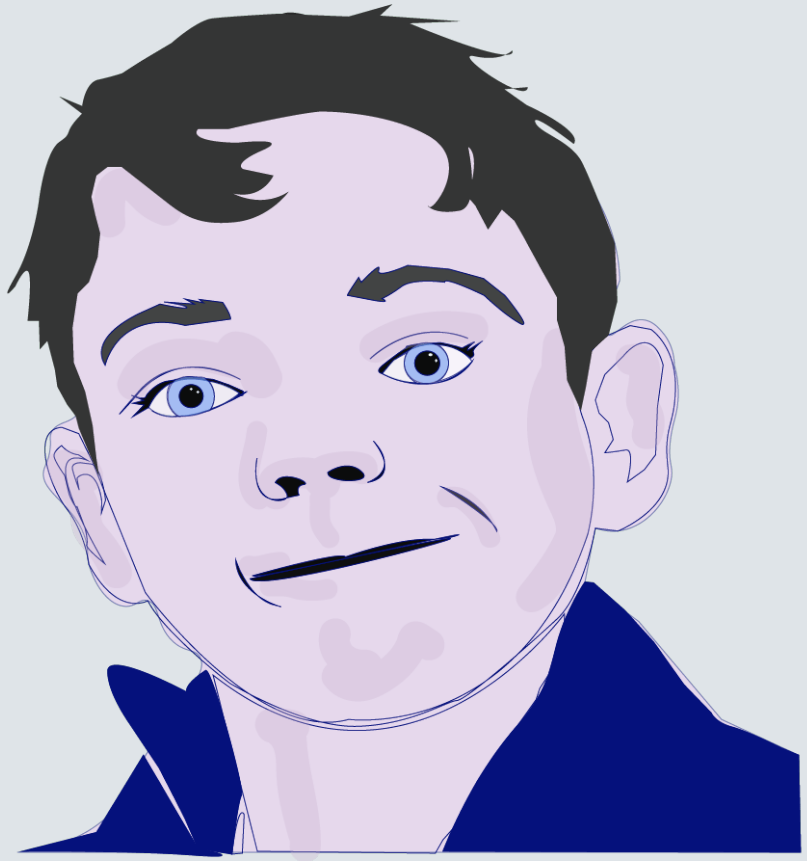
THE URBAN PENGUIN

5: Using Templates

We can use templates to customize configuration files. We can use a template to make sure a local time zone is used to the agents timezone.



Templating Languages



With Puppet 6 we have the choice of two languages that we can use for templates.

- Embedded Puppet (EPP) has been available since Puppet 4 and use Puppet expressions enclosed in tags
- Embedded Ruby (ERB) which is available in all Puppet version but used embedded Ruby expressions

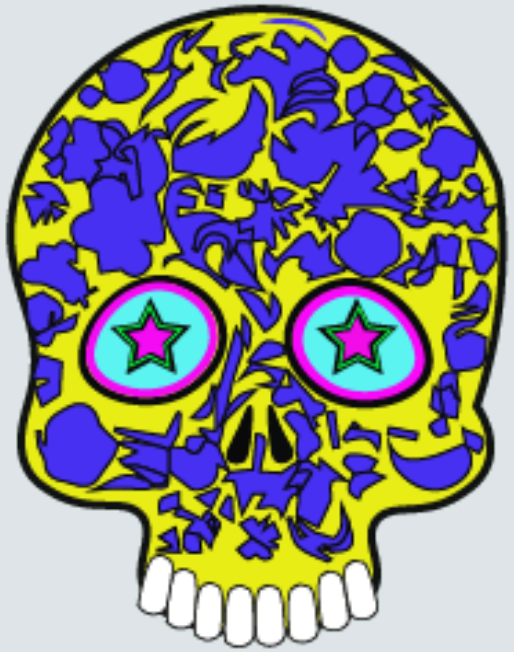
```
# vim test.epp
You have connected to the Server:
<%= $facts['ipaddress'] %> <%= $facts['fqdn'] %>

# puppet epp render test.epp
You have connected to the Server:
192.168.0.119 centos8
```

Simple Template

A simple template like this may work for the /etc/motd file. We can test the result with puppet epp render

Complex Configurations



Templates can allow for easier management of complex configuration where customization is required

Using the chrony module we created we can adjust the time server used based on the timezone of the agent

```
# cat /etc/puppetlabs/code/environments/production/modules/chrony/files/chrony.conf

server uk.pool.ntp.org iburst
driftfile /var/lib/chrony/drift
makestep 1.0 3
rtcsync
keyfile /etc/chrony.keys
leapsectz right/UTC
logdir /var/log/chron
```

Static File

With this file all agents will be directed to one on the pool servers in the UK pool. If we have servers in other time zones we may want to set another pool

```
mkdir /etc/puppetlabs/code/environments/production/modules/chrony/templates

# mv /etc/puppetlabs/code/environments/production/modules/chrony/files/chrony.conf \
 /etc/puppetlabs/code/environments/production/modules/chrony/templates/chrony.epp

<% if $timezone == 'BST'{ -%>
server uk.pool.ntp.org iburst
<% } elsif  $timezone == 'GMT' { -%>
server uk.pool.ntp.org iburst
<% } else { -%>
server us.pool.ntp.org iburst
<% } -%>
driftfile /var/lib/chrony/drift
makestep 1.0 3
rtcsync
keyfile /etc/chrony.keys
leapsectz right/UTC
logdir /var/log/chron
```

EPP Template

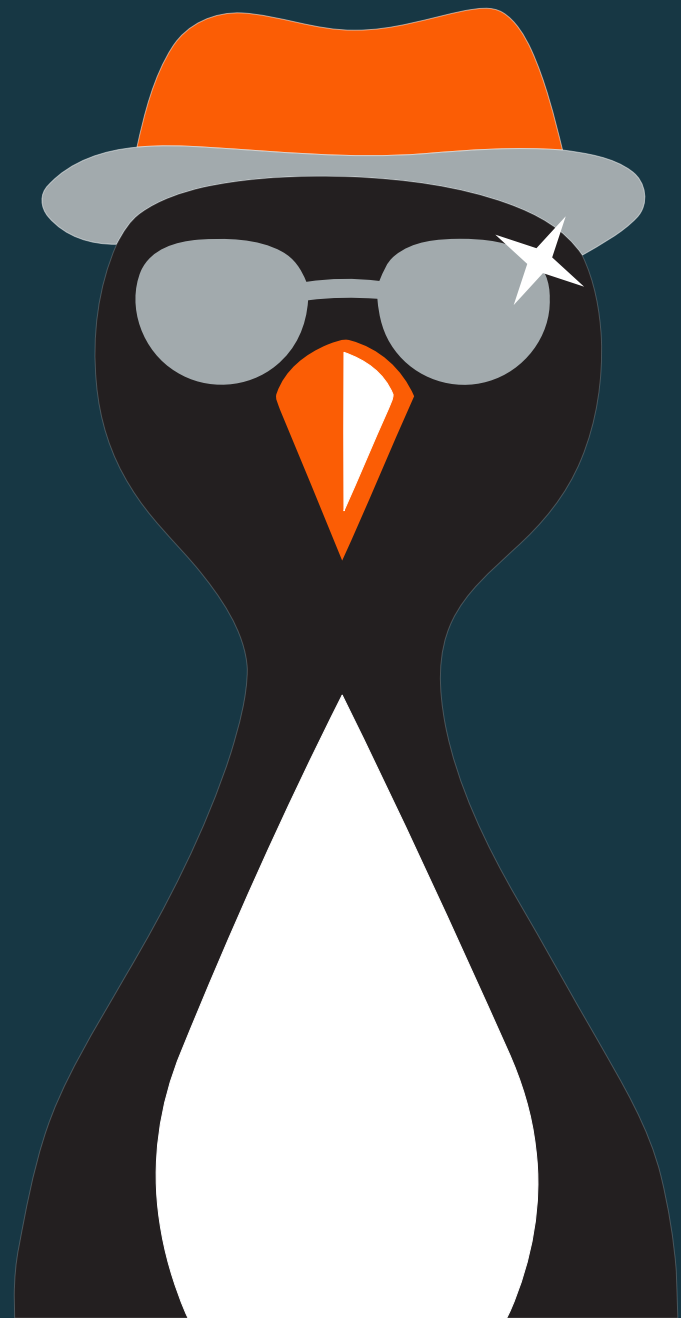
We can embed Puppet logic to the file to read the $timezone fact

@theurbanpenguin

THE URBAN PENGUIN

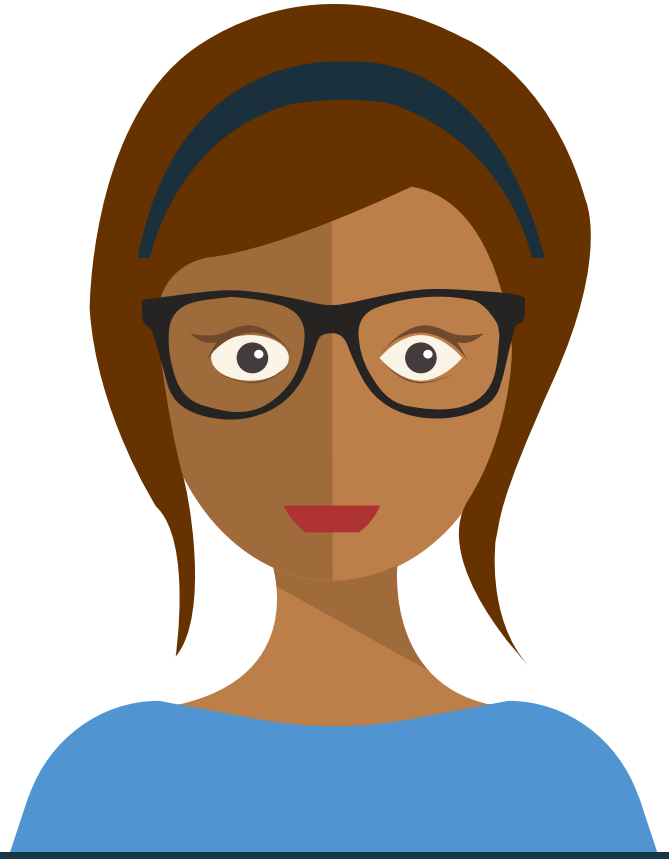
6: Implementing Puppet Bolt

Puppet being a client server model we do need the agent installed for it to be useful. This may be part of our build process but we can also use Puppet Bolt to deploy the agent



Puppet Bolt

Bolt is a stand-alone configuration management command line tool that allows you to run code remotely on agentless systems. Perhaps allowing you to install the puppet agent on the remote system. The default protocol is SSH but winRM can also be used.



```
# yum install puppet-bolt

# bolt command run whoami -t 127.0.0.1 \
  -u root -p Password1 --no-host-key-check

# bolt command run whoami -t 18.133.123.30 \
  -u tux -p [Password1] --no-host-key-check --run-as root

# ssh-keygen ; ssh-copy-id tux@18.133.123.30
```

Using Bolt

Although bolt can be every extensive it is often simply used to install the puppet agent so devices can be managed via puppet. Using ssh keys we don't need to specify the password

```
# bolt command run 'yum install puppet' -t 18.133.123.30 -u tux --no-host-key-check --run-as root

# bolt command run 'echo "18.133.1.1 puppet" >> /etc/hosts ' -t 18.133.123.30 \
 -u tux --no-host-key-check --run-as root

# bolt command run 'systemctl enable --now puppet' -t 18.133.123.30 \
 -u tux --no-host-key-check --run-as root
```

Install Agent

Make sure you use the correct IP Address for your remote agent and add the correct address for your Puppet Master

```
# puppetserver ca list
# puppetserver ca list --all
# puppetserver ca sign --certname cert.example.com
```

Puppet CA

The Puppet CA running on the Puppet Server must authorize new agents by signing their keys

@theurbanpenguin