Puppet

- Introduction & Motivation
- Declarative vs. Imperative
- Resource Types
- Resource Abstraction Layers (RALs)
- Classes

Logistics/Agenda/Intro Why use configuration management? Why use Puppet? What is Puppet? Declarative vs. Imperative Resource Types Writing and applying our first manifest Resource Abstraction Layer (RAL) Classes

What is Puppet?

Declarative Configuration Management:

- Automate the provisioning, configuration and ongoing management of machines
- Make rapid, repeatable changes and automatically enforce the consistency of systems and devices
- Works remotely or locally

Stop using bash to automate your cloud!

Why use Configuration Management?

- Reliability
- Consistency
- Reduce human error / deployment mishaps
- Reduce cost of managing infrastructure
- Increased effectiveness of IT / devops personnel
- Less grey hairs

Why Puppet?

There are other deployment/configuration solutions out there, so why Puppet?

- Delivers us from ordering (declarative)
- Excellent open-source community (IRC + mailing list)
- Excellent free repository of snippets (`puppetforge`)
- Open-source, written in Ruby
- Decentralized catalog application (truely parallel)

What does Puppet do, really?

- Configure machines at a single, group, or total level
- Handle OS specific differences
- Filter based on tags / function / type of server
- Logging for entire deployment / configuration operation
- Parallel execution

Best: You declare the final state (users, services, packages) and Puppet does the rest.

IT Story

- Started with single instance of webserver (LAMP)
- Break out database into separate server
- Traffic increases, add more webservers!
- Growth explodes, IT department demands you add IT-accessible user to each server
- Imagine doing this with 3, 30, or 3,000 servers!

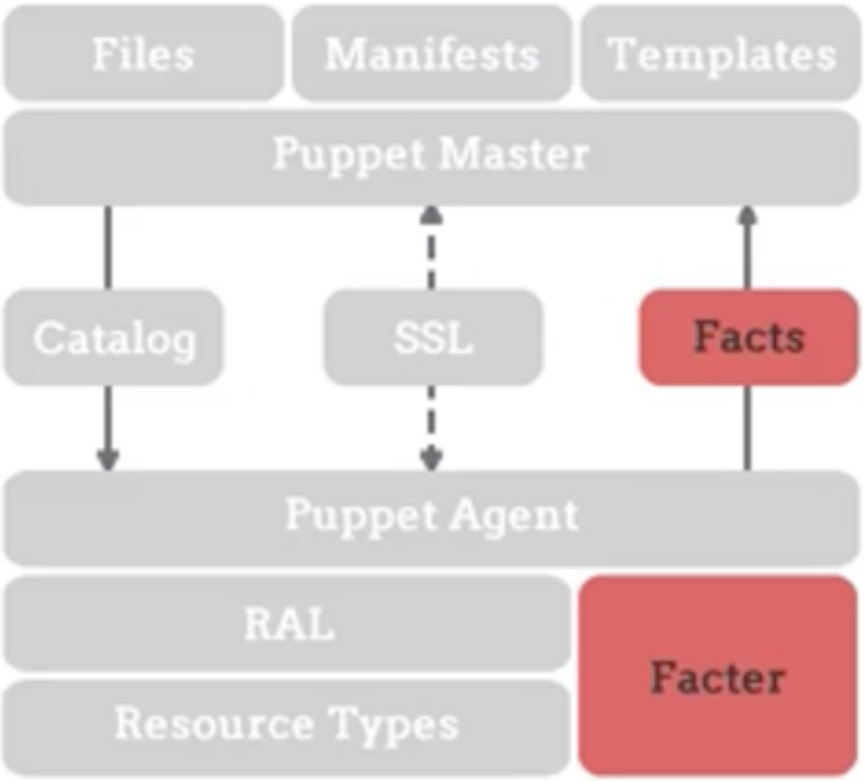
In Summary...

Puppet is:

- Open-source automation tool
- Ruby-based
- Declare
    - which users you want on system
    - which packages you want
    - which services, etc...

and Puppet does the rest!

Puppet Architecture



Where are things in Puppet??

- Executables are in: **/opt/puppetlabs/bin/**
- Private copy of ruby in: **/opt/puppetlabs/puppet**
- Configuration directory: **/etc/puppetlabs/puppet**
- SSL directory: **confdir/ssl/**
- vardir: **/opt/puppetlabs/puppet/cache/** - used only by the puppet agent and puppet apply
- rundir: **/var/run/puppetlabs** - don't mess with this, just PID files
- codedir: Modules, manifests, and hiera are all in: **/etc/puppetlabs/code**
  - environments
  - modules
  - hiera.yaml

If you ever forget!

```
$ puppet config print |grep dir
```

Declaritive vs. Imperative

Typical programming & shell scripts is imperative/procedural - we must define the order.

Declarative is when the lower level work has been done and we can simply state the end result. Puppet is such a higher level wrapper around system management.

Example: Ensuring Creation of a Unix User

```
$ sudo useradd -u 100 -g 200 -c "John User" -m john
$ sudo useradd -u 100 -g 200 -c "John User" -m john
useradd: user 'john' already exists
```

So in bash, an imperative language, we'd need some logic to handle this:

```
#!/bin/bash
getent password john > /dev/null 2> /dev/null
if [ $? -ne 0 ]; then
    sudo useradd -u 100 -g 200 -m john
else
    echo User john already exists
fi
```

This is just a user...imagine doing this for 100 servers all with different needs!

Idempotence

One of the hallmarks of a declarative language or procedure is that the operations are idempotent, i.e. applying them the 1st time has the same effect as the Nth and Nth + 1 time.

Once again, Puppet is a wonderful tool - someone else has done the dirty work for us!

Examples: x^2 (not idempotent), x*1 (idempotent)

Example: User Creation (Puppet Style)

Here's a sneak peek at a Puppet resource declaration:

```
user { 'john':
    ensure     => present,
    uid        => '100',
    gid        => '200',
    comment    => 'John User',
    managehome => true,
}
```

Doesn't that feel better?

Resources

- Fundamental unit of Puppet configurations
- Singular element you want to evaluate/create/remove
- Puppet comes with built-in resources to maniupate familiar components
  - users
  - groups
  - files
  - packages

A block of code that defines a resource is called a **resource declaration**.

Resource Abstraction Layer

We have only one way of describing a resource (user, group, file, package), but there are many implementations.

Example: Anatomy of a Resource Declaration

Format:

```
type { 'title' :
    attribute1 => value1,
    ...
    attributeN => valueN,
}
```

An example:

```
user { 'gary' :
    ensure => present,
    uid    => '100',
    home   => '/home/gary/',
}
```

Resources: Types

```
type {
    ...
}
```

What types of resources are availiable on Puppet?

```
`augeas`, `computer`, `cron`, `exec`, `file`, `filebucket`, `group`, `h
ost`, `index`, `interface`, `k5login`, `macauthorization`, `mailalias`,
`maillist`, `mcx`, `mount`, `notify`, `package`, `resources`, `router`,
`schedule`, `ssh_authorized_key`, `sshkey`, `stage`, `tidy`, `user`
```

And many more. You can even write your own custom types in Ruby!

Resources: Titles

```
type { 'title' :
    ...
}
```

This is the human-friendly name for this resource. We'll use it later, so don't forget about it.

Important: titles must be **unique per resource type**!

Manifests

Manifests are files with Puppet declarations that may include classes (which we'll cover later) and other Puppet code.

Think of them as scripts, with the exception that of course they are declarative.

Your First Manifest

Inside the `manifests` folder, create the file `hello.pp` and add the following text:

```
notify { 'greeting' :
    message => 'Hello, Puppet World!'
}
```

Now we have a manifest file!

Applying a Manifest

We don't "run" manifests, since Puppet is declarative and idempotent, we "apply" manifests.

So let's try it! Apply the manifest with the following:

```
$ puppet apply manifests/hello.pp

Notice: Compiled catalog for <hostname> in environment production in 0.
01 seconds
Notice: Hello, World!
Notice: /Stage[main]/Main/Notify[greeting]/message: defined 'message' a
s 'Hello, World!'
Notice: Finished catalog run in 0.01 seconds
```

And we've completed our first Puppet manifest.

Lab 1: Hello, Puppet World!

Complete Lab 1.

Lab 2: Creating a User

Complete Lab 2.

Lab 3: Parser Validation

Complete Lab 3

CLI: `puppet resource`

What's really cool is that behind the scenes Puppet is cataloging resources of all types on the system, and allows you to see them in Puppet manifest format!

```
$ puppet resource <type> <title>   # query a specific resource
```

Or for all such resources of given type:

```
$ puppet resource <type>           # query all resources of specified t
ype
```

If you forget what types of resources are availiable, you can always:

```
$ puppet resource --types          # returns a list of types
```

Example: `puppet resource`

Find the user we ensured existed before:

```
$ puppet resource user joe  # type=user, name=joe
```

What's even cooler is behind the scenes Puppet is doing this for all resources of all types on the system, and allows you to see them in Puppet manifest format!

```
$ puppet resource user   # get list of all resources of type=user!
$ puppet resource group  # same, but with groups
```

Core Resource Types

Just a few: `notify, file, package, service, exec, cron, user, group`

Here's a handy cheatsheet (http://docs.puppetlabs.com/puppet_core_types_cheatsheet.pdf) (I've already included this in your repo though :)

Resources: Review

- Smallest unit of declaration in Puppet
- Services, files, pacakges, etc
- Many types
- `$ puppet resource <type> <title>`

Lab 4: `puppet resource` Practice

Complete Lab 4.

Lab 5: Hosts in `/etc/hosts`

Complete Lab 5.

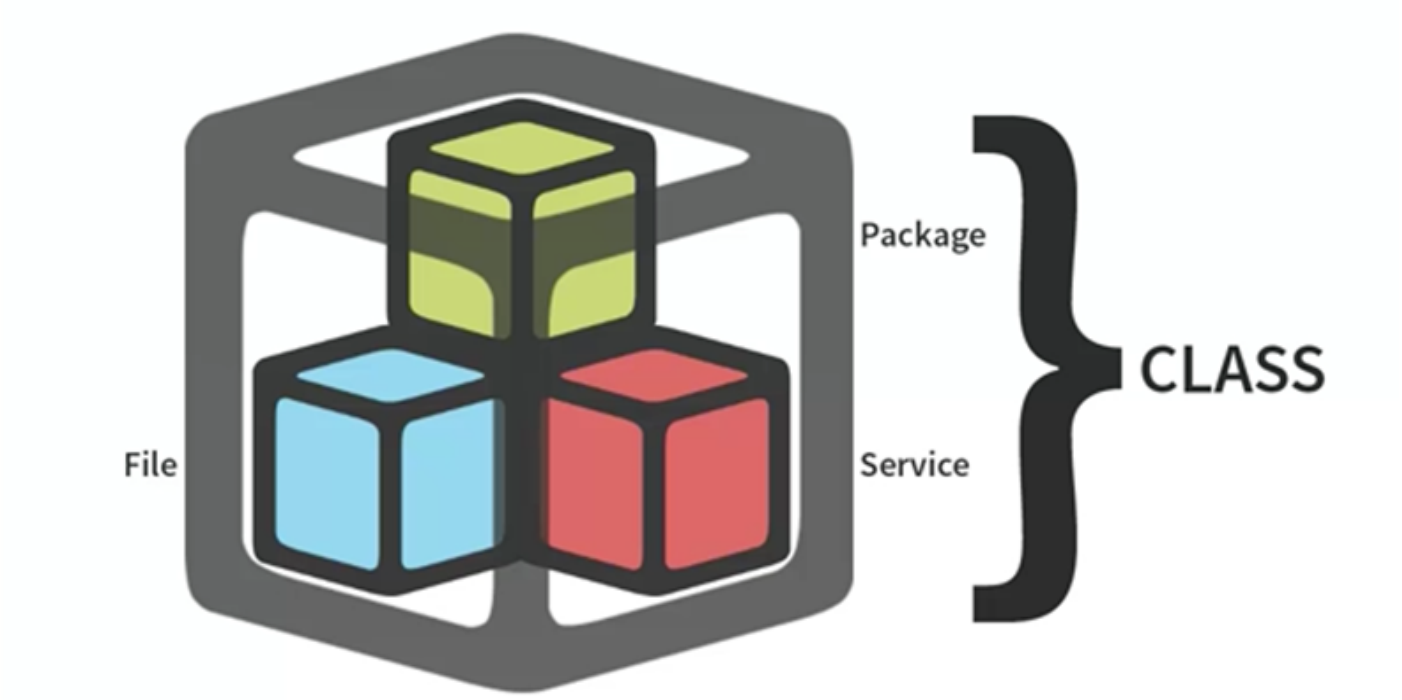
Lab 6: Recovering Files with Filebucket

Complete Lab 6.

Classes

A collection of resources managed together as a single unit.

Defining a class makes it availiable but does **not** apply it to anything.

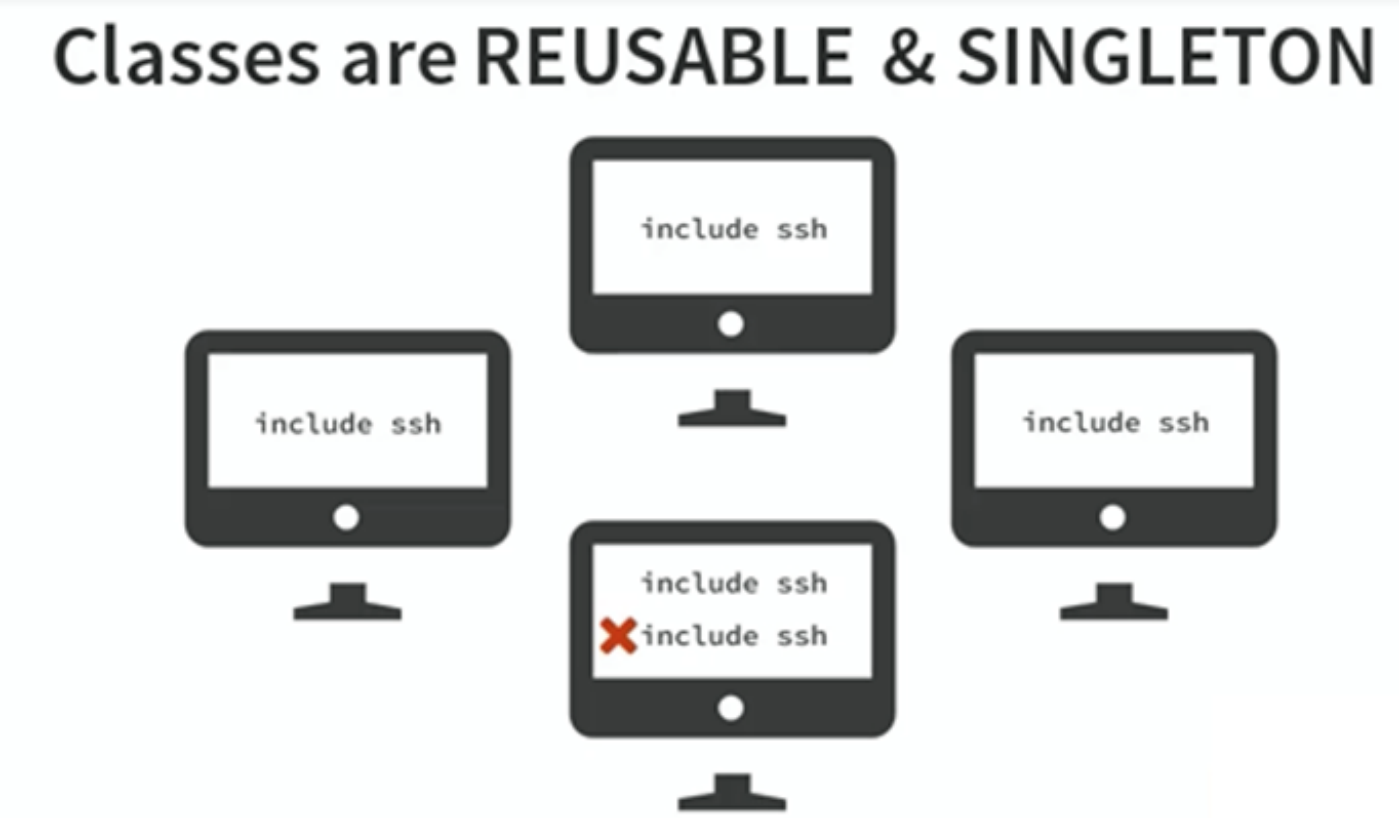


How do we define a class?

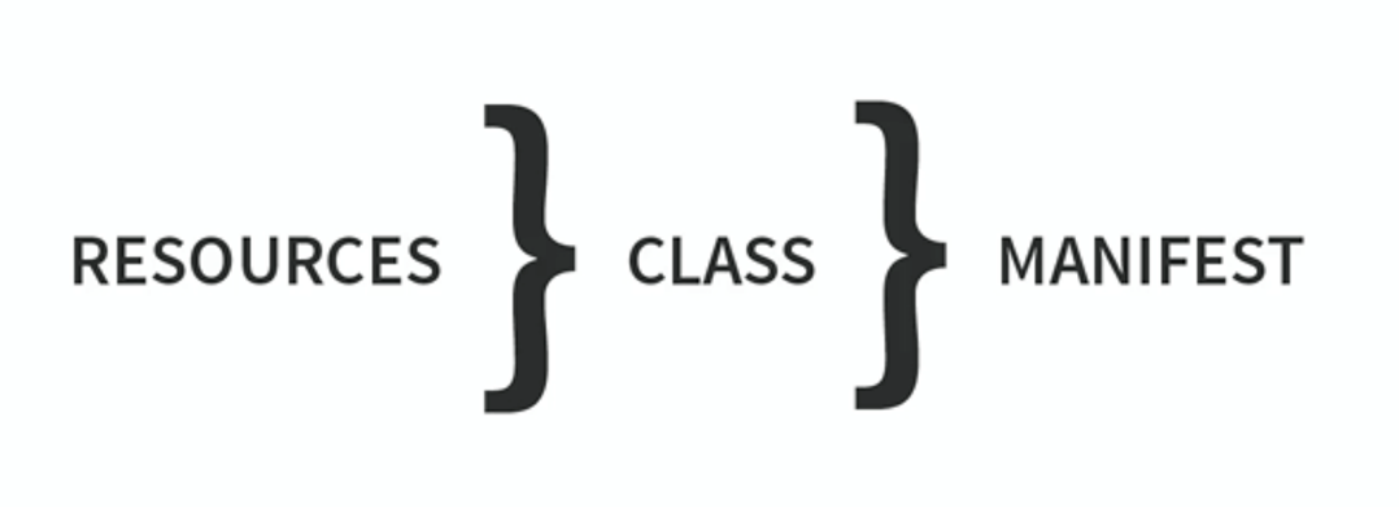
```
class {
    type1 { 'title' :
        ...
    }

    type2 { 'title2' :
        ...
    }
}
```

Running this won't cause anything to happen, you are only **defining** it.

Class Properties



Hierarchy of Puppet Organization



Modules

- Self-contained, reusable Puppet bundles of code/data
- Can be installed from community or you can make your own
- Like an "app" you can add to your Puppet setup

Think of them as add-ons or extensions to Puppet.

Notes on Modules

Puppet can ONLY find classes that are part of modules.

All manifests also belong in modules (with the exception of the special `site.pp`).

What is in a module?

A module is essentially a folder with a predictable inner structure:

```
/etc/puppet/modules/<MODULE NAME>/
    manifests/
        init.pp  # contains class definition
    files/
    templates/
    lib/
    facts.d/
    examples/
    spec/
```

We'll explain all these parts later. You can also double check where Puppet will look for modules by checking your shell environment with:

```
$ puppet config print modulepath
```

The Puppet Forge

Visit the Forge (https://forge.puppetlabs.com/)

Resources, Classes, Manifests, and Modules?

What is the difference?!?

- **Resources**: building blocks of systems
- **Classes**: collections of resources that can be applied to servers
- **Manifests**: groups of resources and classes to be applied together
- **Modules**: reusable, configurable Puppet classes

Package-File-Service

The Puppet "Trifecta"

A **package** requires a **file** (like a configuration file) and needs to be running on the system as a **service**.

A surprising number of configurtion or devops flows can fit into this simple paradigm.

Example: Package-File-Service

```
class apache {
  package { 'httpd':
    ensure => present,
  }
  file { '/etc/httpd/conf/httpd.conf':
    ensure => file,
    owner  => 'root',
    group  => 'root',
    mode   => '0644',
    source => 'puppet:///modules/apache/httpd.conf',  # relative filepa
th - good!
  }
  service { 'httpd':
    ensure => running,
  }
}
```

Ordering in Puppet

If you've ever written a deployment script, you know that ordering is difficult and tedious:

- Ensuring you created that directory before you tried to write a filepath containing it
- Started a service and inserted a "sleep" wait and (hopefully) it will be done
- Wrote functions/scripts to detect if a particular step was completed so you can continue to the next
- Wrote monitoring capabilities to restart services based on configuration changes

Well, no more. Puppet to the rescue!

Ordering in Puppet (con't)

We can use special attributes, like `before`:

```
package { 'openssh-server' :
    ensure => present,
    before => File['/etc/ssh/sshd_config'],
}
```

Here, when the resource `'openssh-server'` is being evaluated and applied, the `before` attribute says "Wait a minute, BFEORE you complete this, you need to create this File ('/etc/ssh/sshd_config')!".

Ordering: Special Attribute Names

A "target resource" with one of the following attributes:

- **before** — Applies a resource before the target resource.
- **require** — Applies a resource after the target resource.
- **notify** — Applies a resource before the target resource. The target resource refreshes if the notifying resource changes.
- **subscribe** — Applies a resource after the target resource. The subscribing resource refreshes if the target resource changes.

```
package { 'openssh-server' :
    ensure => present,
    notify => File['/etc/ssh/sshd_config'],
}
```

Example: Two Equivilent Orderings

This:

```
service { 'sshd' :
    ensure => running,
    require => [
        Package['openssh-server'],
        File['/etc/ssh/sshd_config']
    ],
}
```

is the same ordering as:

```
package { 'openssh-server':
    ensure => present,
    before => Service['sshd'],
}

file { '/etc/ssh/sshd_config/' :
    ensure => file,
    mode => '0600',
    source => 'puppet:///modules/sshd/sshd_config',
    before => Service['sshd'],
}
```

What is the ordering?

Lab 7: Install & Use: `puppet-lint` Package

Complete Lab 7.

Lab 8: Custom Classes & Modules

Complete Labs 8a and 8b.

Ordering: Chaining Arrows

We can also use a special syntax for this in Puppet:

```
Package['ntp'] -> File['/etc/ntp.conf'] ~> Service['ntpd']
```

Translation:

1. First, ensure `ntp` installed, then
2. Create `/etc/ntp.conf`, then finally
3. Start the `ntpd` service, but monitor the configuration file and restart the service whenever it changes.

Lab 9: Package-File-Service: NTP

Complete Lab 9.

Review: Modules

- Self-contained bundles of code/data
- Puppet Forge vs. writing your own
- Only way to load classes is as part of module
- Package-File-Service Trifecta
- Ordering constraints

Recap

- Why use Puppet?
- Declarative & idempotent
- Resources
- Classes
- Manifests
- Modules