Getting Started with Puppet

Student Guide

Puppet Education

http://learn.puppet.com/

Objectives

After completing this course, participants will be able to:

- Identify use cases for adopting Puppet Enterprise in their environment.
- Differentiate between imperative and declarative techniques for configuration management.
- Locate, download, wrap, and deploy Puppet modules from the Puppet Forge.
- Gain visibility into the current state of their systems.
- Use Bolt & Puppet tasks to run scripts or initial automation tasks on multiple nodes.
- Model declarative state for ongoing platform management.
- Automate application delivery when the platform is stable.

Schedule - Day 1

9:00-9:15	Introduction & Warm Up
9:15-9:45	Puppet Ecosystem Overview
9:45-10:15	Lab 1.1: Puppet Product Overview
10:15-10:30	**Break**
10:30-11:15	Bolt and Puppet Tasks
11:15-12:00	Labs 2.1-2.2: Installing and Using Bolt
12:00-1:00	**Lunch**
1:00-1:30	Task Plans
1:30-2:15	Introduction to Declarative Puppet Concepts
2:15-2:30	**Break**
2:30-3:15	Set Up Your Lab Environment
3:15-4:00	Lab 5.1: Agent Deployment and Development Tools Setup

Schedule is subject to change and may be updated as needed

Schedule - Day 2

9:00-9:15	Review of Day 1
9:15-9:45	Lifecycle of a Puppet Agent Run
9:45-10:15	Lab 6.1: Puppet Resources
10:15-10:30	**Break**
10:30-11:00	Lab 6.2: Using and Extending Facter
11:00-11:30	Using Puppet Modules from the Forge
11:30-12:00	Lab 7.1 Puppet Forge
12:00-1:00	**Lunch**
1:00-1:30	Creating Wrapper Modules
1:30-2:15	Lab 8.1: Create a Wrapper Module
2:15-2:30	**Break**
2:30-3:15	Basic Module Testing
3:15-4:00	Lab 9.1: Test Module Syntax and Style

Schedule is subject to change and may be updated as needed

Schedule - Day 3

9:00-9:15	Review of Day 2
9:15-9:45	Creating Role and Profile Modules
9:45-10:15	Lab 10.1: Create Roles and Profiles
10:15-10:30	**Break**
10:30-11:00	Data Separation
11:00-11:30	Create a Baseline with Puppet
11:30-12:00	Lab 12.1: Expand Initial Roles and Profiles to a Baseline
12:00-1:00	**Lunch**
1:00-1:30	Creating and Accepting Parameters in the Baseline
1:30-2:15	Lab 13.1: Class Params
2:15-2:30	**Break**
2:30-3:15	Define an Application Stack with Puppet
3:15-3:45	Lab 14.1: Define and Deploy an Application Stack
3:45-4:00	Course Conclusion

Schedule is subject to change and may be updated as needed

Training & Certification

Getting Started with Puppet is the first step in the certification curriculum for the Puppet Professional Certification.

For more information about Puppet Training, please visit: http://learn.puppet.com.

For more information about the Puppet Certification Program, please visit: http://puppet.com/certification.

Student data policy

Student data is processed in accordance with Puppet's privacy policy posted at www.puppet.com/legal.

Presentation Primer

Controls and navigation.

Move to the next slide.	`[space]`, `[down]`, `[right]`, `[pagedown]`
Move to the previous slide.	`[up]`, `[left]`, `[pageup]`
Show the table of contents.	`[c]`, `[t]`
Toggle follow mode.	`[f]`
Show help dialog.	`[h]`, `[?]`

Use the menu in the upper left to:

- Browse directly to any slide in the deck.
- Download classroom support files.
- Provide pace feedback.
- Ask anonymous questions of the instructor.
- Provide content feedback to the Education department.

The presentation will normally track along with the instructor. If you turn off *follow mode*, then you can navigate separately to any slide you like.

We use the Showoff web-based presenter. It allows for many interactivity features, some of which we use in the classroom. You are welcome to use it in your own presentations.

https://github.com/puppetlabs/showoff

Making Acquaintances

Help me tailor the classroom experience towards your needs.

How long have you been using Puppet? ----

What roles do you serve at work?

- [ ] Technical Support
- [ ] Sysadmin
- [ ] DB Admin
- [ ] Developer
- [ ] Management

Which operating systems do you have experience with?

- [ ] Linux
- [ ] MacOS
- [ ] Solaris
- [ ] Windows

Have you used Puppet Enterprise? ○ Yes ○ No

Save

Puppet Portfolio Overview

Lesson 1: Puppet Portfolio Overview

Objectives

At the end of this lesson, you will be able to:

- Identify the portfolio of products Puppet has to offer.
- Identify use cases and roadmap considerations for adopting Puppet Enterprise.

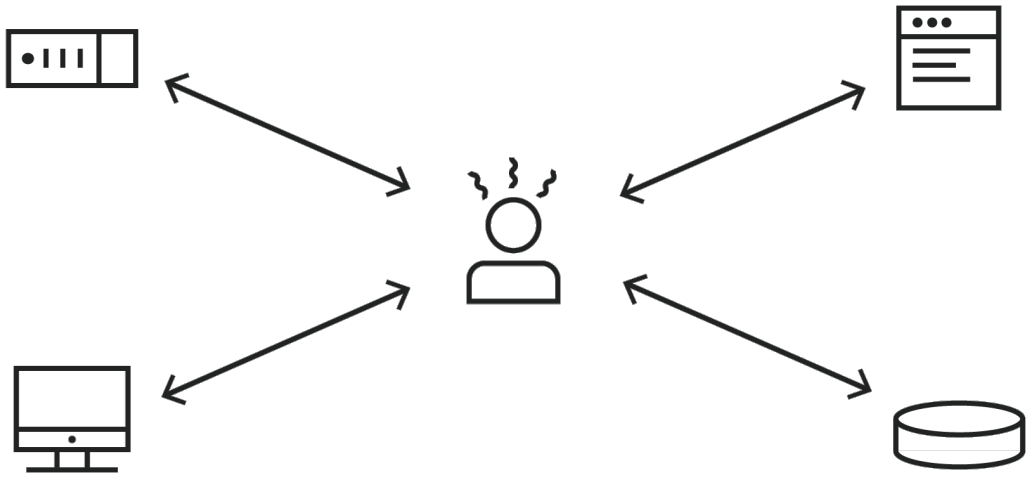
While this class will focus on configuration management, our portfolio addresses a wide range of use cases beyond the core.

About Puppet

- Founded in 2005
- Globally distributed in Portland, London, Plzen, Sydney, Tokyo and elsewhere
- Deep partnerships with leading cloud, container, and datacenter brands
- 1,000+ enterprise customers, including 75 of the Fortune 100
- 5,000+ community-contributed modules, 7.5M lines of code
- 37,000+ organizations using Puppet
- Backed by VMware, Cisco, Google Ventures and others

Legacy Infrastructures

One-off problem solving.



In legacy infrastructures, or those not yet using configuration management, system administrators often run from machine to machine correcting and resolving issues one at a time. Often the tasks are repetitive and involve repeatedly debugging the same issues. This can lead to crushing levels of firefighting in organizations that may already be understaffed.

Organizations will typically go through several iterations in their search for solutions provided by configuration management:

- Manual Configuration
  (literally logging in to every node to configure it)
  Difficult to scale. Practically impossible to maintain consistency between nodes.

- Golden Images
  (Using a complete template to create new node installations)
  Need separate images for different environments, configurations, or roles. Very difficult to maintain consistency across multiple image versions and virtualization platforms. Monolithic images are rigid and difficult to update as the business needs change.

- Custom One-off Scripts
  (custom code written to address a specific, tactical problem)
  Scripts are difficult to reuse for different applications or different deployments. They are brittle: as needs change, the entire script must often be rewritten. They are difficult to maintain when the original author leaves the organization. They can be less reliable because scripts are typically used and tested only by your organization and not by a complete community.

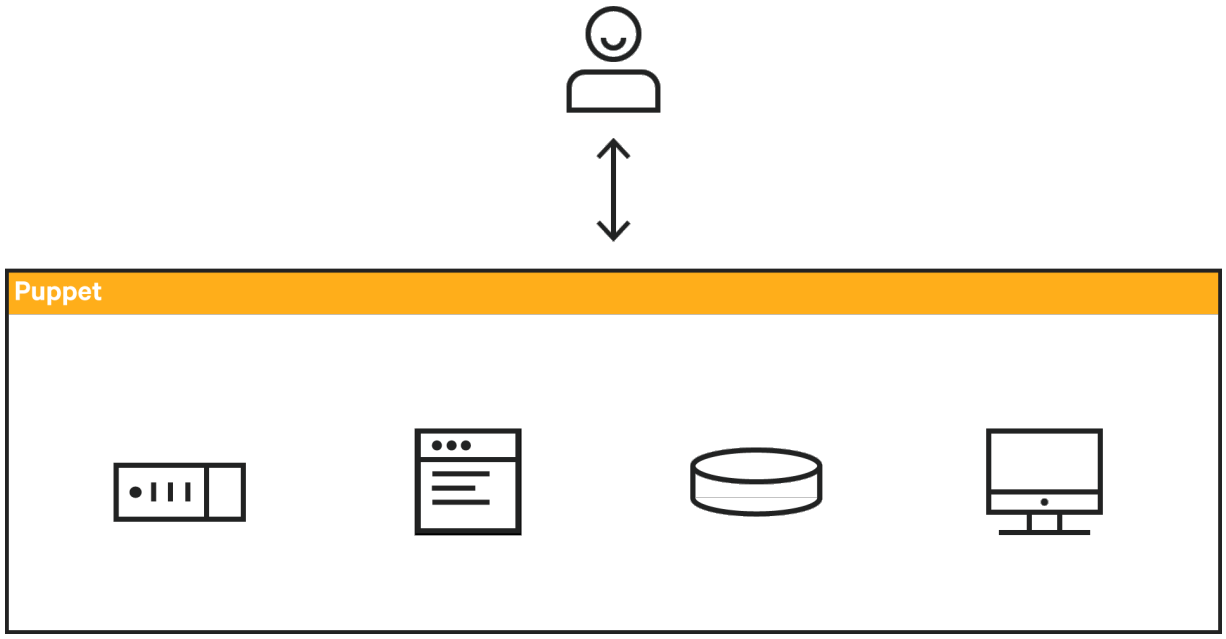
Legacy Infrastructure Problems

- Inconsistent deployments
- Slow rollouts
- Putting out fires instead of being proactive
- Business as usual for most IT operations teams

Legacy infrastructure is plagued by many problems. Change control attempts to mitigate these issues, but in defining workflows and processes, it can reduce the ability to respond quickly and efficiently.

Configuration Management

Using Puppet to enhance your problem solving ability.



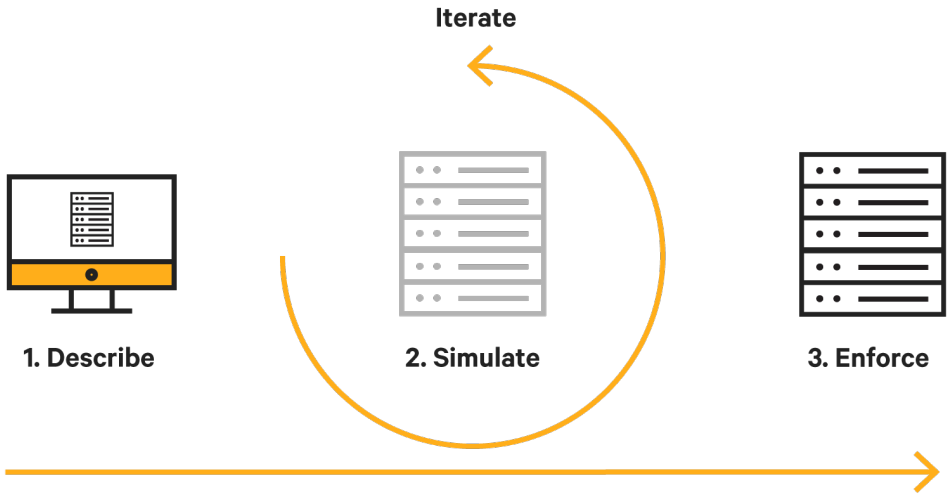
Instead, configuration management allows the system administrator to solve a problem once, in a repeatable fashion, describe that solution in Puppet code committed into the organization's codebase, and then deploy that solution automatically everywhere it's relevant.

Puppet helps you to solve problems universally and keeps your infrastructure consistent and predictable. This means that you can solve more problems and support a larger infrastructure without staffing up to unsustainable levels.

An interesting tool to help your organization evaluate your own performance is the Operations Report Card, located at http://www.opsreportcard.com

Model Based Approach

Describe your desired state and let Puppet enforce it



1. Describe your infrastructure and its desired state

Use Puppet to describe the attributes of resources. Manage as much or as little as you'd like and progressively roll out configuration management.

1. Simulate the enforcement of these resource definitions

Understand the impact of changes before they affect production.

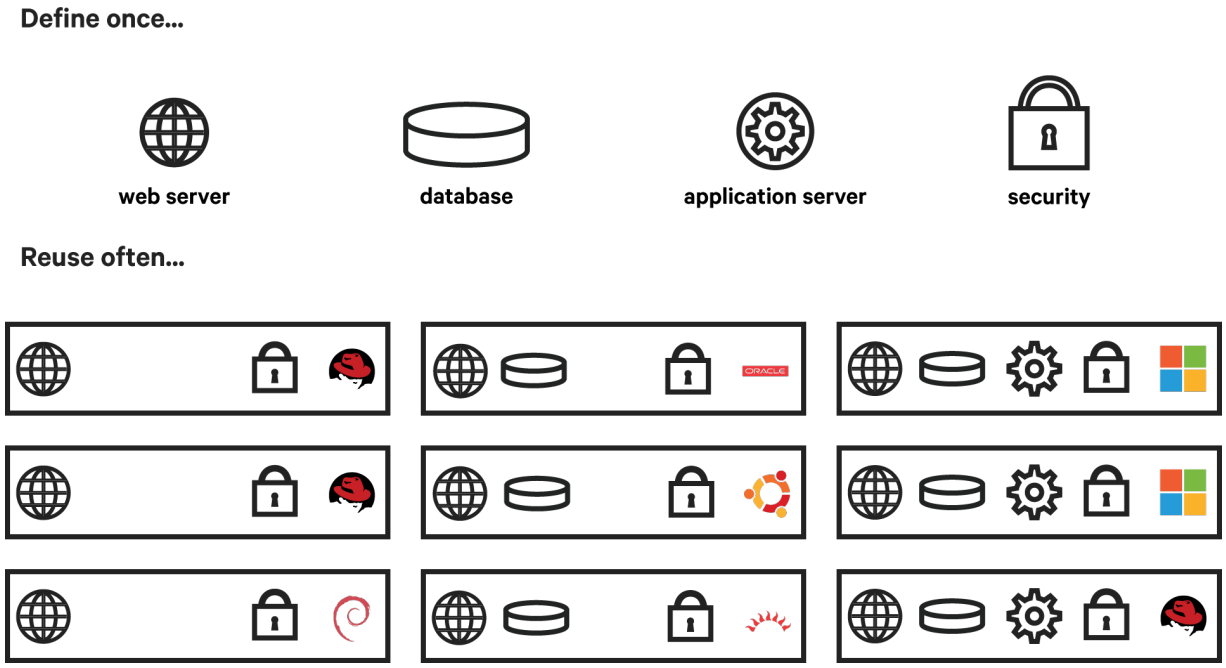
Iterate as often as you need to make sure that you get your actual desired state.

1. Enforce the desired state of your infrastructure

Periodically check each node for compliance with these definitions and automatically resolve drift to maintain infrastructure-wide configuration consistency.

Composable Configurations

Build configuration models from smaller components



Puppet's human-readable DSL enables you to specify and manage your infrastructure with defined models of your infrastructure, not procedures.

Complete services and applications--web servers, database servers, application services--can be built from collections of modules or reusable "building block" components.

Because these models are centrally managed, you can make changes once, test them, and then deploy configuration consistently to multiple nodes.

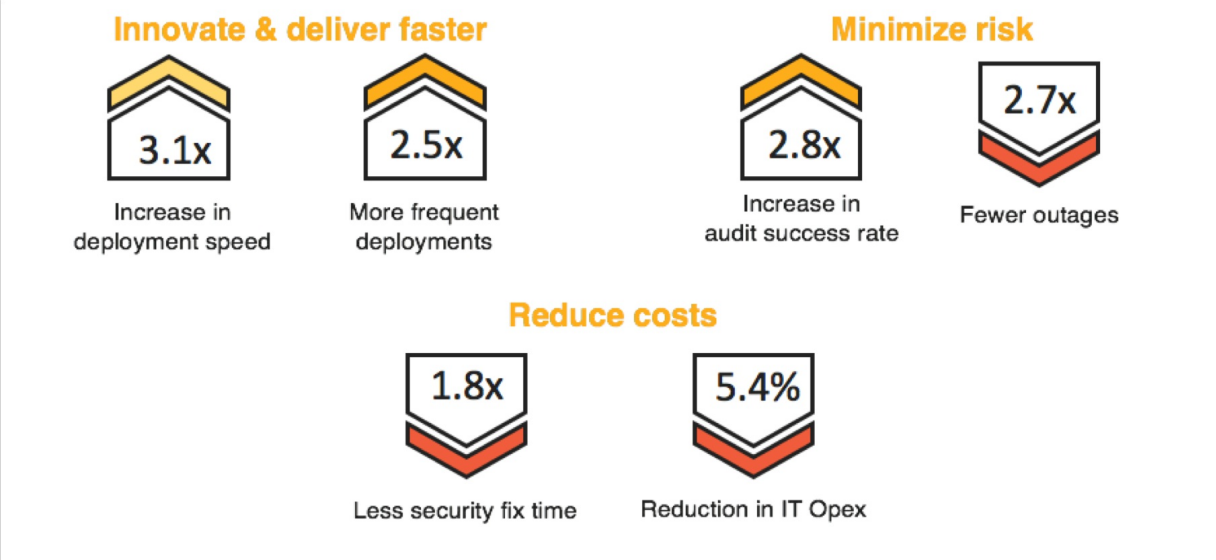
Puppet's resource abstraction layer enables reusable and portable configurations across any supported platform.

To help users get started, Puppet has thousands of freely downloadable modules for resources, applications, and services on the Puppet Forge community: http://forge.puppet.com.

How Can I Use Puppet in My Environment?

- Puppet is cross-platform so it can be used with:
  - Linux
  - MacOS
  - Windows
  - Many network devices
  - Solaris
  - AIX

Automation Delivers Results



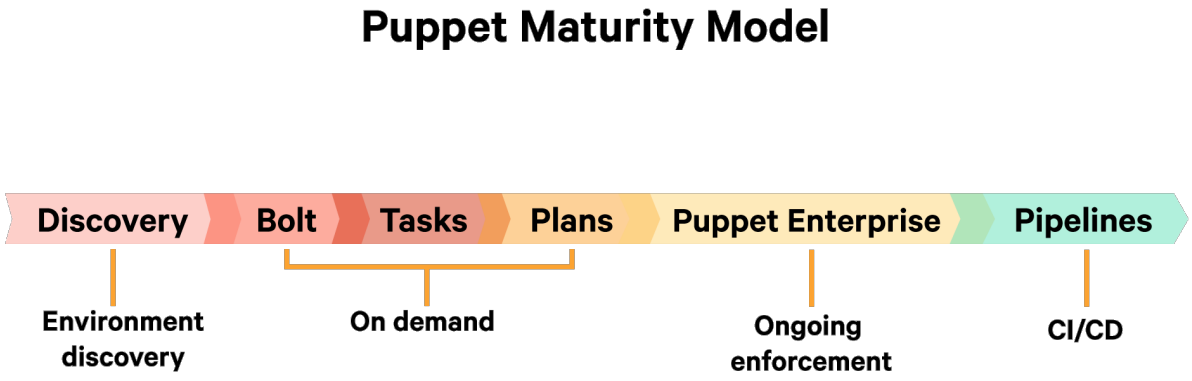
Deliver better software faster, and deliver value to the company. For our customers who do this across their IT organization and get it right, their results are staggering. On average, they:

- Move from infrequent deployments to delivering quality software 3.1X faster and 2.5X more frequently
- Move from outages and firefighting to experiencing 2.7X fewer failures and recovering from downtime faster
- Move from security bottlenecks and failed compliance to spending 1.8X less time fixing security problems and passing audits 2.8x more often
- Go from months-long cloud migration projects to moving to the cloud faster and adopting containers faster and more reliably
- Go from bloated budgets and IT as a cost center to reducing IT operating expenses by 5.4% while increasing digital revenue by 1.42%

Source: ESG Economic Value Validation Report. Results are avg of >200 Puppet Enterprise customers.

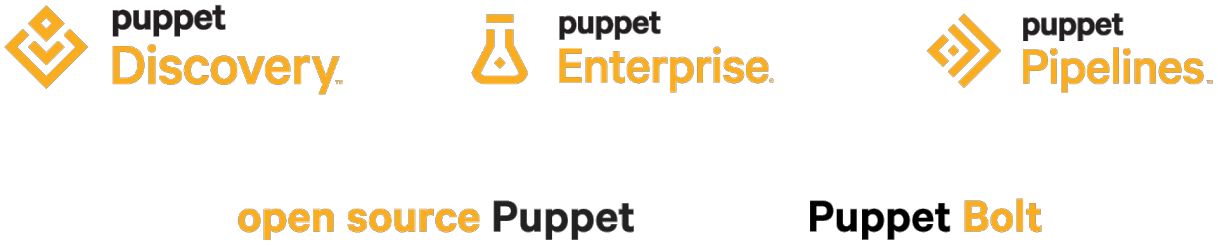
Also see the State of DevOps Report at http://www.puppet.com/devops-report

Where are you in the journey?



The Puppet Portfolio

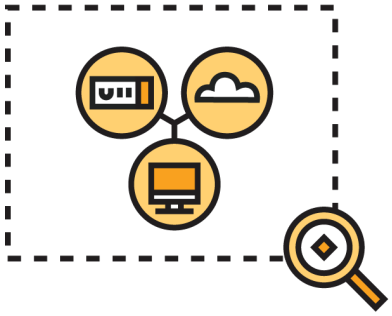
Discover. Manage. Deliver.



The Puppet portfolio includes a blend of open source and licensed products:

- Open source Puppet includes a collection of more than 40 open source projects, ranging from the core components for automating infrastructure management, to development and testing tools, to service frameworks.
- Puppet Bolt is an open source, agentless task runner that helps deploy one-off changes, troubleshoot systems, and execute sequenced actions in a deployment workflow. If you're just getting started with automation, you can use Puppet Bolt to execute tasks on systems and devices remotely.
- Puppet Enterprise gives you additional powerful capabilities that you can use right out of the box to automate delivery and operation of enterprise-scale infrastructure, securely and with full reporting. These capabilities — orchestration, role-based access control, full portability, reporting, and compliance — allow you to automate an entire application stack, and every element of your data center and cloud infrastructure.
- Puppet Discovery allows you to identify and inventory all of your IT resources, from on-prem to public cloud. Puppet Discovery is purpose-built for resource discovery, and it's your first step toward pervasive automation.
- Puppet Pipelines simplifies continuous delivery and unifies workflows across your Dev and Ops teams. It automates the build and deployment of your applications — whether they're traditionally packaged or container-based apps running in Kubernetes — and gives you deep visibility and audit trails for every action taken.

Puppet Discovery

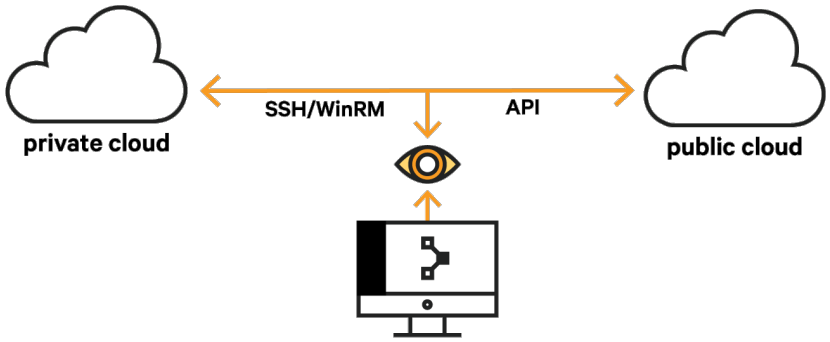


- Provides visibility into what's running on your entire hybrid infrastructure
- Allows you to install Puppet agents and perform other tasks

Puppet Discovery gives you an easy to understand dashboard that provides insights about your infrastructure. It shows node-specific information in context and allows you to start and stop services, install Puppet agents, and perform other tasks.

Puppet Discovery gives you a real-time picture of your infrastructure.

How Puppet Discovery Works



- Frictionless
- Uses SSH, WinRM, and public cloud APIs
- May trigger security alerts if network management and security sensors are not properly configured

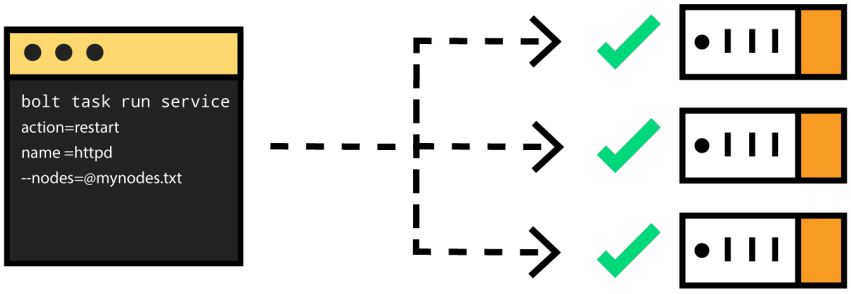
> Note: we will demonstrate Puppet Discovery in the first lab at the end of this module.

Puppet Discovery is a frictionless software application that uses SSH, WinRM, and public cloud APIs to connect to and forage a wide range of resources. Simply add your data sources and your machine credentials to start foraging physical hosts; AWS, Azure, and Google cloud instances; VMware virtual machines; and containers.

CAUTION: Prior to installing and running Puppet Discovery, it is recommended that you work with your network security administrators to ensure network management and security sensors are appropriately configured. Using each SSH and WinRM credential you provide, Puppet Discovery attempts to authenticate with each discovered host until a successful authentication is achieved. This process is repeated every 30 minutes. Depending on the configuration of your network management and security sensors, Puppet Discovery's activities may trigger alerts or active response.

Note: we will demonstrate Puppet Discovery in the first lab at the end of this module.

Puppet Bolt



- Allows you to execute one-off tasks or entire plans and scripts
- Tasks run with Bolt are reusable and shareable via the Puppet Forge

Sometimes, you just need to run some scripts—but wouldn't it be great if you could run and manage your scripts from a central console?

That's what Puppet Bolt provides. Bolt is driven through a command line interface (CLI) and connects to remote systems via SSH or WinRM, so it doesn't require you to install any agent software. The tasks you run with Puppet Bolt are reusable and shareable via the Puppet Forge, and can be written in any scripting or programming language.

Puppet Bolt is great for deploying on-demand changes, troubleshooting, distributing scripts to run across your infrastructure, or automating changes that need to happen in a particular order as part of an application deployment. Bolt is also great if you are just getting started with infrastructure automation. You can think of it as a bridge between the adhoc scripting you might be doing today and full-fledged infrastructure automation. We'll talk more about Bolt later in this course.

Open Source Puppet

- **Puppet language (DSL)**
- **Server components & database**
- **Unified agent**
- **Community support**

Open source Puppet is great for individuals managing a small set of servers. It provides the core configuration automation and provisioning capabilities: the Puppet domain specific language (DSL), the core server components and PuppetDB, and the Puppet agent. Companies and individuals who use open-source Puppet can get support from a rich and engaged community of users. Visit http://slack.puppet.com/ to sign up for the free Slack community.

Puppet Enterprise

Puppet Enterprise adds powerful capabilities that you can use right out of the box to automate delivery and operation of enterprise-scale infrastructure, securely and with full reporting. These capabilities allow you to automate an entire application stack and every element of your data center and cloud infrastructure:

- Orchestrate infrastructure and application deployments in a graphical interface. Grouping nodes by function, hardware and software properties, or any distinction you can express with your own code.
- Puppet models infrastructure as code, so code management becomes critical as you scale your infrastructure. Puppet Enterprise provides additional code management features such as supported out-of-the-box workflows, multi-master file sync, and command line and API interfaces.
- Puppet Enterprise is designed for scale, with a Web UI for managing infrastructure nodes.
- With a graphical interface to reporting, you know exactly how your infrastructure and applications are configured. You know what has changed in real time, and who is running Puppet tasks on your nodes.
- With Puppet Enterprise, you have access to live support when you need it, a private support portal with email and phone support, all product updates, training, and attention to your bug reports and feature requests.

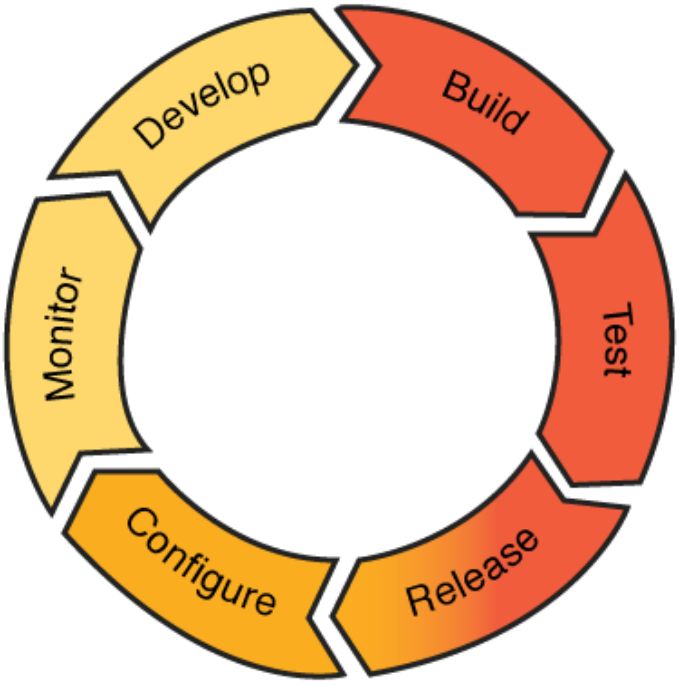
Puppet Enterprise Console

- Provides an interface to adhoc tasks on managed nodes
- Offers RBAC to restrict functions to appropriate users

The Puppet Enterprise console is a graphical interface to the Puppet infrastructure. The Puppet Enterprise console:

- Presents an overview of your systems
- Displays detailed information about each node
- Collates and displays statistics
- Provides an interface for node classification
- Enables report browsing and viewing
- Can invoke Puppet run on nodes
- Can run abitrary, or adhoc, tasks

Puppet Pipelines



Automates the build and deployment of your applications.

- Continuous Delivery for Puppet Enterprise
- Puppet Pipelines for Applications
- Puppet Pipelines for Containers

Puppet Pipelines enables powerful, customizable CI/CD workflows for teams of all sizes. Attaching a Pipelines application to your software repository enables you to automatically build and deploy whenever code is checked in. It also provides an audit trail and a one-click rollback. It can use BitBucket, GitLab, Github, and other code repositories.

Workflow Consolidation with Puppet Pipelines

- Continuous delivery and release automation in any cloud and on-prem
- Intuitive UI to create full CD pipelines with a few clicks
- Set approvals, gates, triggers, and revert changes with one-click rollbacks
- Fine-grained access control and detailed audit trails

Puppet Pipelines connects your preferred source control repository (from GitHub, BitBucket, GitLab or others) in one click and automatically triggers application builds for every code commit or pull request. Simply configure a few build steps and Puppet Pipelines automatically builds a release package and makes it ready for testing.

Continuous Delivery for Puppet Enterprise

- Automatically build, test, and promote infrastructure changes
- Create continuous delivery pipelines for infrastructure code

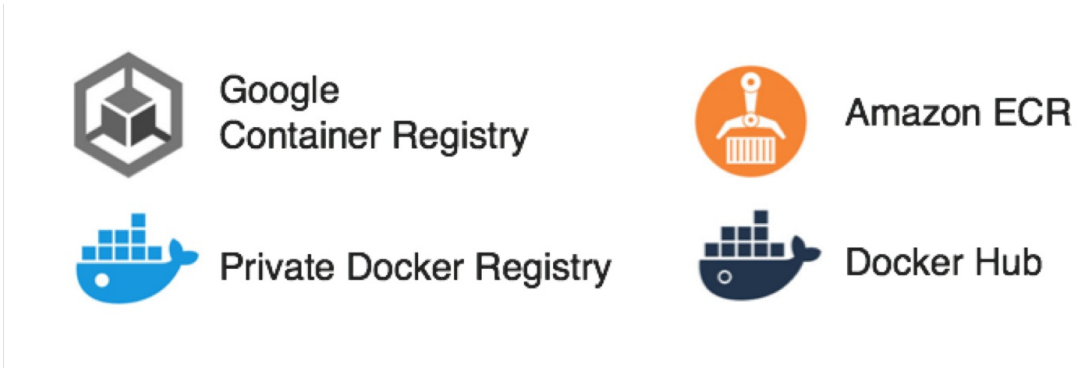
Continuous delivery isn't just for your application deployment. With Continuous Delivery for Puppet Enterprise, operations staff can use the same workflows as their devs: build, test, and deploy infrastructure with repeatable, templated, easily managed, code. It makes continuous delivery simple for infrastructure managed with Puppet Enterprise. Admins gain both control and visibility into how infrastructure is tested and deployed so businesses can deliver rapid change with confidence.

Automatically build, test and promote infrastructure changes. Apply continuous integration practices to your infrastructure code. Connect your preferred source control repository in one click, and quickly create and run jobs to build and test Puppet code before you deliver it to your environments.

Create continuous delivery pipelines for infrastructure code. Apply continuous integration practices to your infrastructure code by creating and running jobs to build and test Puppet code before you deploy it to your environments. Use visual dashboards to create full continuous delivery pipelines for your Puppet code.

Puppet Container Registry

- Host Docker images
- Connect to local or remote registries
- Audit and automate application and container deployment



Puppet Container Registry (formerly known as Europa from Distelli) makes it easy for software teams to host Docker images within their infrastructure along with a unified view of all their images stored in local and remote repositories. With powerful features like single sign-on and access control, Puppet Container Registry is designed to be easy for developers to use, and engineered for the enterprise.

Local Registries

Puppet Container Registry hosts local repositories backed by storage in your cloud or on-premises servers. Local registries implement the Docker V2 API and support the complete range of operations from push and pull to listing tags and repositories.

Remote Registries

Puppet Container Registry also connects to your remote repositories hosted in third party registries such as Amazon Elastic Container Registry, Google Container Registry, or Docker Hub registries. Puppet Container Registry will scan these registries and send webhooks on every push detected on these remote repositories.

Audit Trails

Puppet Container Registry ensures that you have a complete audit trail of every action on every local and remote repository in your infrastructure. View every push/pull along with the image tag, SHA, and the user who completed the action.

Automated push pipelines

Puppet Container Registry is designed with advanced automation and allows software teams to set up automated push pipelines from one repository to multiple downstream repositories. This allows teams to easily set up dev, test, and prod repositories and automatically make images and tags available in multiple repositories with a single Docker push.

Access control

Puppet Container Registry Premium and Enterprise editions support fine-grained access control for users. Authorization and access controls ensure that users can only push / pull or view repositories that they have been granted access to. Multiple groups ensures powerful control over who can access a repository.

Image locality

Pipeline features allow you to ensure that Docker images are always available in the repository closest to your servers and cloud infrastructure. Every push to a local repository ensures that the same image and tag is available for pull from a remote repository within your cloud infrastructure, thereby reducing costs and increasing speed of delivery.

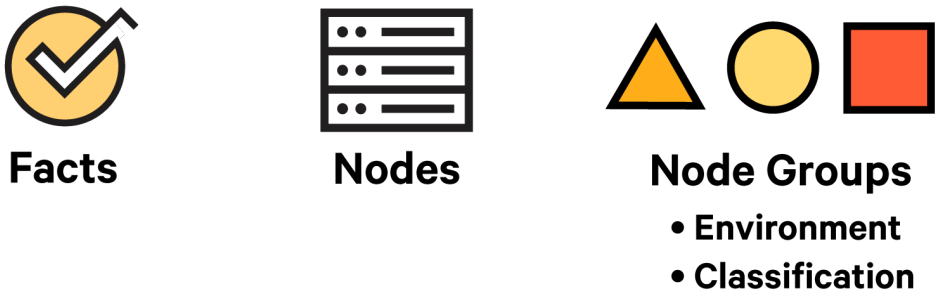
Single sign-on

Puppet Container Registry Enterprise edition supports single sign-on integrated with your on-premises SAML Identity Provider (IDP). A unified login ensures that teams don't have to manage individual passwords and accounts within Puppet Container Registry.

Teams

Puppet Container Registry is multi-tenant and allows enterprises to host a single instance of Puppet Container Registry and create multiple teams for better isolation. Users can be added to multiple teams and access controls will define the level of access a specific user has to a specific repository in a specific team.

Preparing for the Lab: Basic Concepts



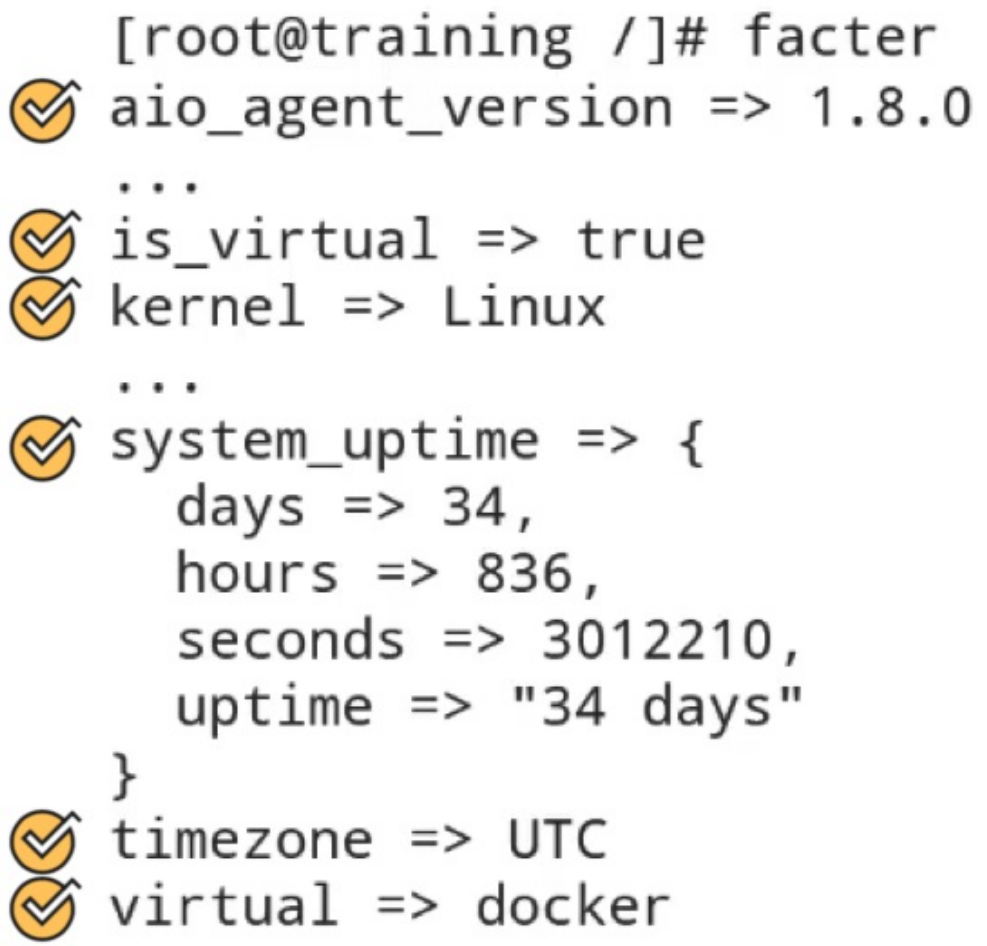
In the lab you will learn a few basic components of Puppet including facts, nodes, and node groups

- Facts are information about nodes.
- Nodes are servers or devices in your infrastructure.
- Node Groups allow you manage related nodes as a single unit.

Facts

Facts can be used to generate inventory reports or they can be used by Puppet to make decisions about how to configure a given node.

- Facts are information (about each node)
- Key-value pairs
- Provided by Puppet agent



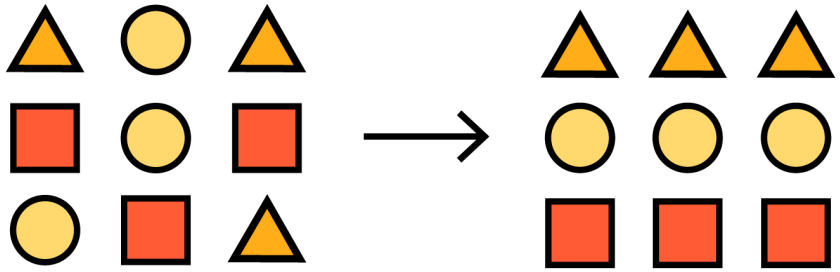
Facts are information collected about a node and reported to the Puppet master on each run. They can be key value pairs like environment variables or structured data. Any information about a node that the master needs to know, in order to make configuration choices, must be communicated as a fact.

Facts are collections of normalized system information about target nodes used by Puppet. For example:

- timezone
- IP address
- hostname
- whether the system is virtual
- uptime
- disk space
- EC2 metatdata
- ... and anything you can express in custom code.

Node Groups

Node groups are a collection of related systems that are managed as a unit.



- Node groups make it easier to assign configuration and management to a category of servers.
- Node groups exist in a hierarchy of parent and child relationships.
- Nodes inherit classes, class parameters, variables, and rules from all ancestor (parent) groups.

Common node management tasks include adding and removing nodes from your deployment, grouping and classifying nodes, and running Puppet on nodes. You can also deploy configuration changes to nodes using an environment-based testing workflow.

The main steps involved in classifying nodes are:

1. Create node groups.
2. Add nodes to groups, either manually or dynamically, with rules.
3. Assign classes to node groups.

Nodes can match the rules of many node groups. They receive classes, class parameters, and variables from all the node groups that they match.

Node groups exist in a hierarchy of parent and child relationships. Nodes inherit classes, class parameters and variables, and rules from all ancestor groups.

Classes – If an ancestor node group has a class, all descendant node groups also have the class.

Class parameters and variables – Descendant node groups inherit class parameters and variables from ancestors unless a different value is set for the parameter or variable in the descendant node group.

Rules – A node group can only match nodes that all of its ancestors also match. Specifying rules in a child node group is a way of narrowing down the nodes in the parent node group to apply classes to a specific subset of those nodes.

Why Node Groups Matter

Node groups are useful because configuration is applied to many different sub categories of device.

For example:

- Servers in production use particular package versions.
- Servers not in production use newer package versions.
- Servers in development set different root/Administrator passwords.
- Network devices in a particular datacenter enable link-layer discovery.

Node groups allow configuration to be selectively applied.

Node groups simplify configuration management, giving you consistency across disparate environments. For example, on both Windows and Linux platforms you might need to enable network time synchronization, configure a static route, block a port, or disable running web daemons/services. Node groups make it easy to select a specific set of nodes and then deploy configuration as appropriate.

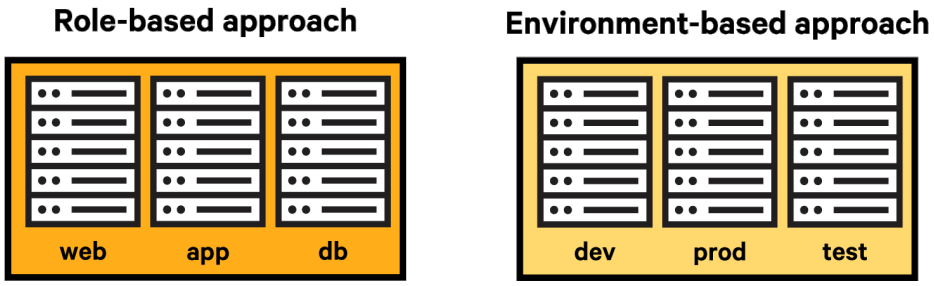
Configuring the correct servers is critical when applying automation. With node groups you can selectively apply configuration to the systems you want it to affect.

Using Node Groups

Typically administrators have either a role based approach (www/app/db) or an environment based approach (dev/prod/test) to grouping servers.

With Puppet, both approaches are possible!

You can use the ‘Prod' node group to push some changes and other changes can push to the ‘Web' servers node group, regardless of whether they are in production, testing, or development node groups.

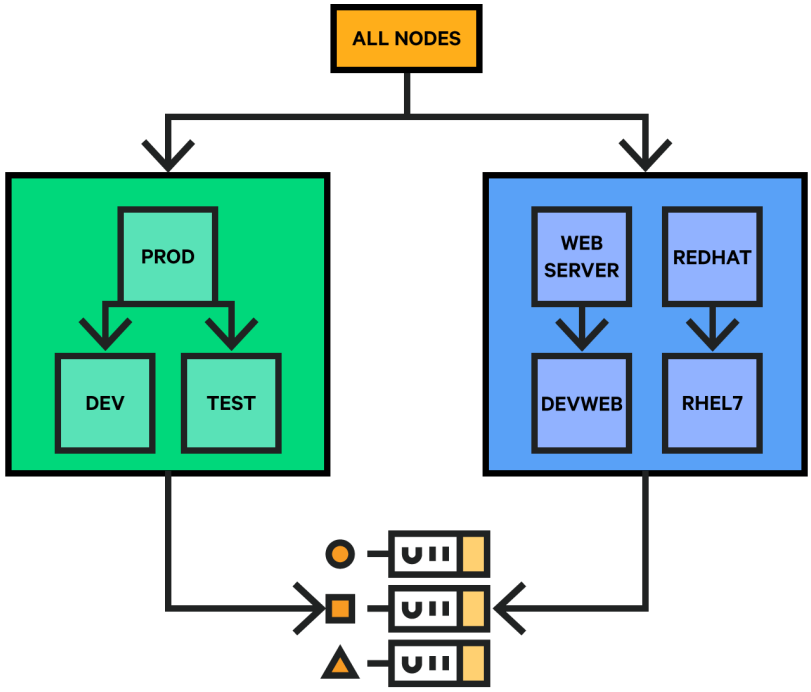


The two types of groups differ in purpose. Classification groups are used to indicate which configuration should be enforced on a node, whereas environment groups indicate which version of the Puppet code should be applied to nodes in a group.

Nodes may be members of multiple classification groups. For instance, a node may be in individual groups that enforce a baseline, add a web server, and include extra firewall rules for a DMZ. These groups indicate which items to manage.

Nodes may only be members of a single environment group. For instance, servers may be members of a group that receives the version of Puppet code that you are internally developing now, or members of a staging group that applies Puppet code deemed almost ready for production. Nodes default to the "production" environment group.

Environment vs. Classification Node Groups



It is up to you to maintain design separation between environment node groups and classification node groups.

Classification: **what** a node is (web server, db server, etc.).

Environment: **where** a node is (dev, test, production).

This diagram shows a design pattern for architecting and laying out node groups to implement the recommended environment-based workflow. The differences illustrated here between environment node groups and classification node groups are purely conventional; the node classifier makes no such distinction on a technical level, and does not show environment and classification groups separately.

Lab 1.1: Puppet product overview

Objectives:

- Puppet Discovery
  - Learn how to get a high-level summary of your IT infrastructure
  - Learn how to invoke operations through Puppet Discovery
- Puppet Enterprise
  - Learn how to assign configuration to nodes
  - Learn how to report on infrastructure state
  - Learn how to invoke remote operations on managed nodes

No handouts for this lab.

Checkpoint: Puppet Portfolio

Which Puppet product provides a dashboard showing which systems have agents and which do not?

- ○ Puppet master
- ○ Puppet Discovery
- ○ Puppet Pipelines
- ○ Puppet Container Registry

Puppet Enterprise is based on the same core as open source Puppet.

- ○ True
- ○ False

Which Puppet component gathers information about nodes that can be used to make decisions about what configuration is sent to the node?

- ☐ Puppet Container Registry
- ☐ Puppet Enterprise Console
- ☐ Puppet Pipelines
- ☐ Facter

Save

Bolt and Puppet Tasks

Lesson 2: Bolt and Puppet Tasks

Objectives

At the end of this lesson, you will be able to:

- Evaluate existing processes and/or scripts to define how they can be executed and scaled more efficiently using Puppet tasks.
- Use Bolt and tasks to distribute and run executables on multiple nodes from a central location.

Using Bolt for Ad-hoc Management

`bolt command run`

With one command, remotely execute arbitrary commands on multiple systems and have them return results to one place.

`bolt script run`

Transfer and run arbitrary scripts on multiple systems and collect the results.

`bolt task run`

Tasks add metadata to describe a script, and confine the allowed input parameters to only those values and data types that you choose. After validation, transfer and run scripts, passing variables to them to customize their behavior.

`bolt plan run`

Plans chain tasks, scripts, and commands in one place, then allow you to issue them to multiple sets of systems. In a plan, the results of one task can be used to form the input parameters for a subsequent task on other nodes.

`bolt apply`

Apply arbitrary Puppet code to any node that has a Puppet agent installed. (And can automatically install an agent!)

Executing Ad-hoc Commands and Scripts

`bolt command run`

`bolt script run`

Run existing tools on multiple systems at once

- One-time software package update
- On-demand restart of a service
- Flush caches
- Troubleshooting

Administrators, operators, and developers often have a collection of commands and scripts to automate common tasks. These usually need to be uploaded, or made available otherwise to the target system.

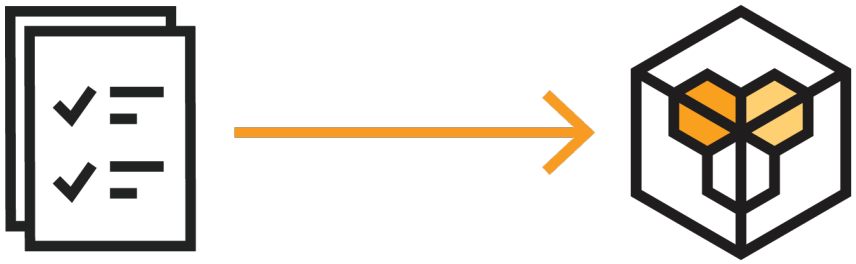
Running scripts from a central location on many target nodes is a powerful amplification of traditional scripting.

Bolt gives admins the flexibility of ad-hoc script usage, with the power of centralized management.

Sometimes you simply need to execute an ad-hoc command or script. Bolt requires no agent or pre-existing scripts on target systems. It can execute ad-hoc commands, run scripts, upload files, and run tasks or plans on remote nodes from a controller node.

The next few slides explain how to install Bolt without Puppet.

Running Puppet Tasks with Bolt



- Similar to scripts
- Execute commands or code on target nodes
- Can include metadata, and may be stored in modules making reuse and sharing easier
- May be any language available on the target nodes
- Task files should use the appropriate extension for the language they are written in (such as `.rb` for Ruby)
- Place them in the top level of your module's `tasks/` directory.

We will not go in depth on what tasks look like as it can get very advanced. For now, it is important that you understand certain aspects of tasks.

- Tasks are similar to scripts in that they both execute commands or code on target nodes. However, tasks can have metadata, and will be stored in modules. This makes reuse and sharing easier.
- Write tasks in any language, as long as that language interpreter is already installed on target nodes.
- Give task files the extension for the language they are written in (such as `.rb` for Ruby), and place them in the top level of your module's `tasks` directory.

Bolt 1.x features

- Apply Puppet manifest code
  - Easy to start
  - Expand automation
  - Test Puppet code
  - Masterless/Agent-less
- Redacted passwords for printed Target objects
- Share code between tasks
- Support for public-key signature system ed25519

Example Bolt Use Cases

- Restart a hung service
- Perform emergency package updates
- Clear a proxy's cache
- Force a resynchronization of date and time
- Run a script to gracefully reload an application stack
- Identify installed version of software across multiple nodes
- Install an ad-hoc patch across many systems

How Bolt Works

- Uses SSH or WinRM transports (no agents!)
- Can be used in conjunction with or independent of a Puppet master and agent.
- Uses scripts in any language
- Can be distributed in modules on the Puppet Forge.

Bolt will use ssh or WinRM to connnect to the node, run the given command, and then clean up after itself returning the output to you.

The commands are simple, as shown below:

```
bolt script run <name of script> --nodes
```

Bolt Supported Protocols

Bolt supports several protocols: * ssh:// Secure Shell * winrm:// Windows Remote Management * local:// Execute commands locally * pcp:// Puppet Communications Protocol (Puppet Enterprise Orchestrator) * docker:// Run commands on container instances via Docker API

Installing Bolt

Windows

- Bolt is distributed as an MSI.
- Simply download and run.

Linux

- Bolt is available in apt and yum repositories.
- Simply add a repository definition and use the package manager.

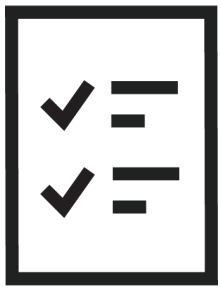
MacOS

- Bolt is distributed as a DMG.
- Simply download, mount, and run the installer.

Detailed instructions for installing Bolt on Windows, Linux, MacOS, or as a Ruby Gem are available at the Puppet docs site:

https://puppet.com/docs/bolt/latest/bolt_installing.html

Running Tasks



- Bolt can run Puppet tasks on remote nodes without requiring any Puppet infrastructure.
- Task names are based on the filename of the task, the name of the module, and the path to the task within the module.
- Use SSH/WinRM to run tasks on target nodes that do not have the Puppet agent installed.

Task names are composed of one or two name segments indicating: * The name of the module where the task is located. * The name of the task file, without the extension.

To execute a task, run `bolt task run` specifying:

- The full name of the task, formatted as `<MODULE::TASK>`, or as `<MODULE>` for a module's main task (the init task).
- Any task parameters, as `parameter=value`.
- The nodes on which to run the task and the connection protocol, with the `--nodes` flag.
- The module path that contains the plan's module, with the `--modulepath` flag.
- If credentials are required to connect to the target node, the username and password, with the `--user` and `--password` flags.

Example: `bolt task run mysql::sql database=mydatabase sql="SHOW TABLES" --nodes neptune --modulepath ~/modules`

Learn more about running remote tasks via SSH in the Puppet docs

Command Line Options

- Bolt commands can accept several command line options, some of which are required to run certain Bolt commands.
- You must specify the target nodes that you want to execute Bolt commands on.
  - For most Bolt commands, specify the target nodes with the `--nodes` flag. For plans, specify nodes as a list within the task plan itself or specify them as regular parameters, like `nodes=neptune`.

To specify multiple nodes with the `--nodes` flag, use a comma-separated list of nodes, such as `--nodes neptune,saturn,mars`.

To pass nodes to Bolt in a file, pass the file name and relative location with the `--nodes` flag and an `@` symbol:

```
bolt command run --nodes @nodes.txt
```

To pass nodes on stdin, on the command line, use a command to generate a list of nodes, and pipe the result to `bolt` with `-` after `--nodes: <COMMAND> | bolt command run --nodes -` For example, if you have a list of nodes in a text file, you might run `cat nodes.txt | bolt command run --nodes -`

Command Line Options

- To pass nodes to Bolt in a file, pass the file name and relative location with the `--nodes` flag and an `@` symbol

To run Bolt commands on target nodes that require a username and password, pass credentials as options on the command line. Examples:

`--nodes, -n` Required when running. Nodes to connect to.

`--query, -q` Query PuppetDB to determine the targets.

`--noop` Execute a task that supports it in no-operation mode.

`--description` Add a description to the run. Used in logging and submitted to Orchestrator with the pcp transport.

`--params <parameters>`, passed as a JSON object on the command line, or as a JSON parameter file, prefaced with `@` like `@params.json`.

For Windows PowerShell, add single quotation marks to define the file: `'@params.json'`

Lab 2.1: Install

- Objectives:
  - Install Bolt.
  - Validate installation.

Lab 2.2: Running commands

- Objectives:
  - Use Bolt to run simple commands.
  - Use Bolt to run existing scripts.

Checkpoint: Bolt and Tasks

Bolt uses UDP to communicate with nodes.

- True
- False

How do you use Bolt to run a script on multiple nodes?

- bolt command run ../sync_time.sh --nodes=my_group
- bolt script run ../sync_time.sh --nodes @nodes.txt
- bolt script run ../sync_time.sh --nodes=my_group
- bolt command run ../sync_time.sh --nodes @nodes.txt

Bolt must be installed on each node before use.

- True
- False

Save

Plans

Lesson 3: Plans

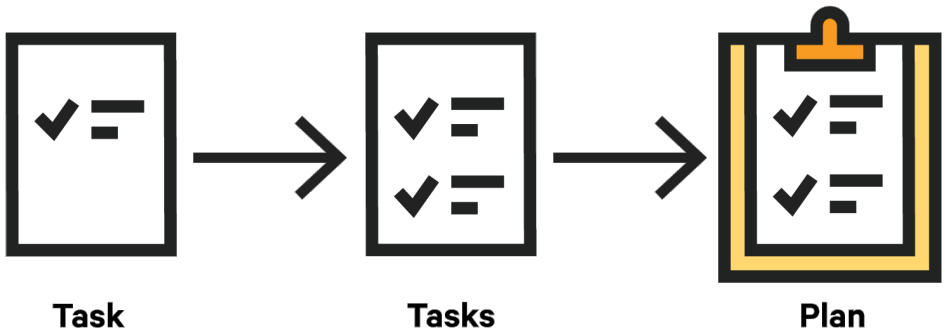
Objectives

At the end of this lesson, you will be able to:

- Describe the relationship between Puppet tasks and plans.
- Recognize a plan, then compare tasks and plans to legacy management techniques.
- Identify when it is appropriate to use plans versus tasks.

Review: What is a Task?

- A single, adhoc action that you can run on target machines in your infrastructure
- Allows you to make on-demand, immediate changes to remote systems.



Bolt can run Puppet tasks and plans on remote nodes without requiring any pre-existing Puppet infrastructure.

Recall from a few slides ago: You can write tasks in any programming language that can run on the target nodes, such as Bash, Python, or Ruby. Tasks may be packaged within modules, so you can reuse, download, and share tasks on the Forge. Task metadata describes the task, validates input, and controls how the task runner executes the task.

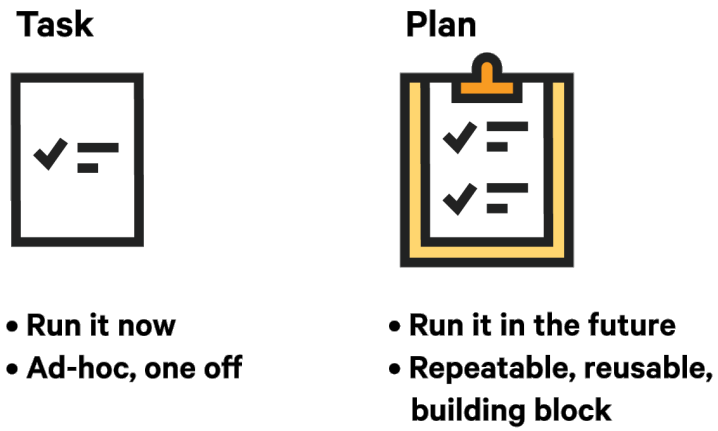
What is a Plan?

- Plans are sets of tasks.
- Can be combined with other logic allowing more complex operations.
- Allow multiple tasks to be run with one command.
- Can compute values for the input for a task.
- Can run tasks based on results of another task.
- Are written in the Puppet language.
- Use the `.pp` extension.
- Are placed in a module's `plans/` directory.

Like tasks, plans may be packaged in modules which can be shared on the Forge. Plans are written in the Puppet domain specific language (DSL).

Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

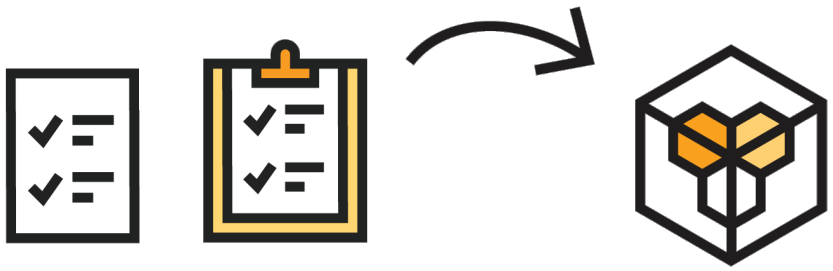
Tasks Versus Plans



Task logic runs on a single target, can be written in any language, and returns the results of a single action on target.

Plans run on a controller, written in Puppet, run multiple tasks or plans, collect results from multiple actions.

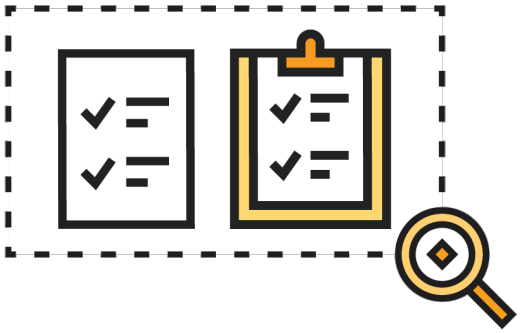
Installing Tasks and Plans



- Tasks and plans may be distributed in Puppet modules, so they are installed when you install the module containing them.

One simple command to install: `puppet module install`

Installing Tasks and Plans



- Test before you commit.
- Always inspect tasks and plans before running them.
- Run in no-operation (`--noop`) mode to execute the task without making changes.

Before you run tasks or plans in your environment, inspect them to determine what effect they will have on your target nodes.

Run in no operation mode

```
bolt task run package name=vim-minimal action=install --noop -n example.com
```

Show a task list

```
bolt task show
```

Show documentation for a task

```
bolt task show <TASK NAME>
```

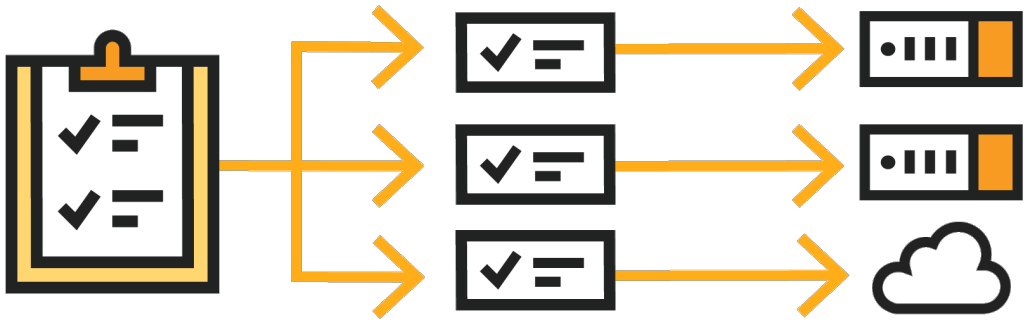
Discover plans

```
bolt plan show
```

Show documentation for a plan

```
bolt plan show <PLAN NAME>
```

Deployment with tasks and plans



- Consistency across all deployments
- Fast deployments
- Run all tasks with a single command
- Reduce repetition
- Manage scripts in one place
- Agentless management of legacy infrastructure

Automate your workflow with tasks and plans.

Sometimes you need to do work in your infrastructure that isn't about monitoring and enforcing the desired state of machines. You might need to restart a service, run a troubleshooting script, or get a list of the network connections to a given node. You perform actions like these with Puppet tasks and plans.

Tasks
Tasks are single actions that you run on target machines in your infrastructure. You use tasks to to make as-needed changes to remote systems.

You can write tasks in any programming language that can run on the target nodes, such as Bash, Python, or Ruby. Tasks are packaged within modules, so you can reuse, download, and share tasks on the Forge. Task metadata describes the task, validates input, and controls how the task runner executes the task.

Plans
Plans are sets of tasks that can be combined with other logic. This allows you to do more complex task operations, such as running multiple tasks with one command, computing values for the input for a task, or running certain tasks based on results of another task. You write plans in the Puppet language. And, like tasks, plans are packaged in modules and can be shared on the Forge.

Inspecting tasks and plans
Before you run tasks or plans in your environment, inspect them to determine what effect they will have on your target nodes.

Running tasks
Bolt can run Puppet tasks on remote nodes without requiring any Puppet infrastructure.

Running plans
Bolt can run plans, allowing multiple tasks to be tied together.

Installing tasks and plans

Tasks and plans are packaged in Puppet modules, so you can install them as you would any module and manage them with a Puppetfile.

Writing tasks

Tasks are similar to scripts, but they are kept in modules and can have metadata. This allows you to reuse and share them more easily.

Writing plans

Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

Example: Deployment with Tasks

- A DBA might want to drop a given user `joe` from all databases after they leave the company.

Instead of going to every MySQL instance on each node and running `DROP USER`, the DBA will run a task to do it.

Bolt can run plans, allowing multiple tasks to be tied together.

To execute a plan, run `bolt plan run`, specifying: * The full name of the plan, formatted as `<MODULE>::<PLAN>`. * Any plan parameters, as `parameter=value`. * The path that contains the plan's module, with the `--modulepath` flag. * If credentials are required to connect to the target node, pass the username and password with the `--user` and `--password` flags.

Conclusion

Tasks and plans make life easier in the following ways:

- Enable automation of ad hoc actions across servers that need immediate changes
- Allow you to run more than one task with a single command
- Allow computed values to be used for the input to a task
- Process the results of tasks
- Make decisions based on the result of running a task

Always remember to only automate configuration and settings that you understand.

Ideally, create tasks and plans that are repeatable and have the ability to scale.

Checkpoint: Bolt, Tasks, and Plans

What has been covered so far:

- What is a task?
- What is a plan?
- Why use Bolt?

Introduction to Declarative Puppet Concepts

Lesson 4: Introduction to Declarative Puppet Concepts

Objectives

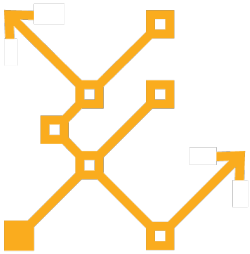
At the end of this lesson you will be able to:

- Discuss when it is appropriate to use imperative (Bolt and tasks) versus declarative (agent) styles of definition.
- Demonstrate how each is best applied in the overall automation process.

Imperative vs. Declarative Puppet

Imperative

- **Now**
- **Ad-hoc, one-off**
- **Bolt**



Declarative

- **Future**
- **Repeatable, reusable, building block**
- **Puppet agent**

Imperative use of Puppet details each step (the collection of which can achieve a desired end state).

Declarative use of Puppet focuses on the end state, defining a contract with a node.

Think of it this way: imperative instructions are focused on an ordered list of specific tasks; declarative instructions define the desired end state.

Solving Real Problems

Use case: managing an *admin* user.

> **Goal**: All servers in our infrastructure should have a non-root *admin* user for common system administration and troubleshooting tasks.

We care that the user *admin*:

- exists
- belongs to the *wheel* group
- is configured with an NFS shared home directory

Solving Real Problems

- Managing a node use cases
  - Manage cron jobs
  - Enforce permissions
  - Set IE enhanced security configuration
  - Ensure services are running

Every time a new server or new application stack is deployed, sysadmins and developers run the same commands over and over again. It is better to let automation do that consistently and reliably.

Existing Management Utilities

Tools for writing your own solution.

> "Some people, when confronted with a problem, think 'I know, I'll write a script.' Now they have two problems."
>
> -- inspired by Jamie Zawinski

- Unix:
  - `useradd` / `usermod`
  - `groupadd` / `groupmod`
  - `mkdir`
  - `chmod`
  - `chown` / `chgrp`
- Windows:
  - `net user`
  - `net localgroup`

These are just some of the built-in commands that would help you solve this problem. For the purpose of this thought exercise, we're looking at built-in system tools, not dedicated user management solutions.

On a Microsoft Windows system, you might use the *Local Users and Groups* snap-in to the Microsoft Management Console, PowerShell scripting methods, or you might use the `net` commands above, such as:

- `net user /add puppet 'puppet8#labs'`
- `net localgroup administrators /add puppet`

The Puppet Way

A light at the end of the tunnel.

```
user { 'admin':
  ensure     => present,
  gid        => 'sysadmin',
  home       => '/mnt/home/admin',
  managehome => true,
}
```

- Tell Puppet the things you care about.
- Let Puppet take care of the details.
- Allow the code itself to serve as documentation.
    - Descriptive and readable language.

This is a built-in Puppet resource that describes the state that we would like this user to exist in. Puppet will bring the resource into compliance by performing any required actions to make the user match this desired state.

This code is descriptive enough that most people can read it and have a pretty decent idea of the configuration applied to this system. This is much better than the alternative of having many different forms of documentation and tracking down which set of documentation is closest to reality.

This code is:

- Descriptive
- Straightforward
- Transparent
- Portable across platforms

To be perfectly accurate, this isn't completely platform independent, because Windows doesn't have the concept of a *primary group* and doesn't allow creation of users without passwords. We'll talk about ways to handle that later in the course.

Puppet Resources

Something that Puppet can manage

resource
A Puppet resource is a single item that Puppet knows how to manage or to interact with. This can be a package, a user, a file, or even a single line in a file--like a host entry.

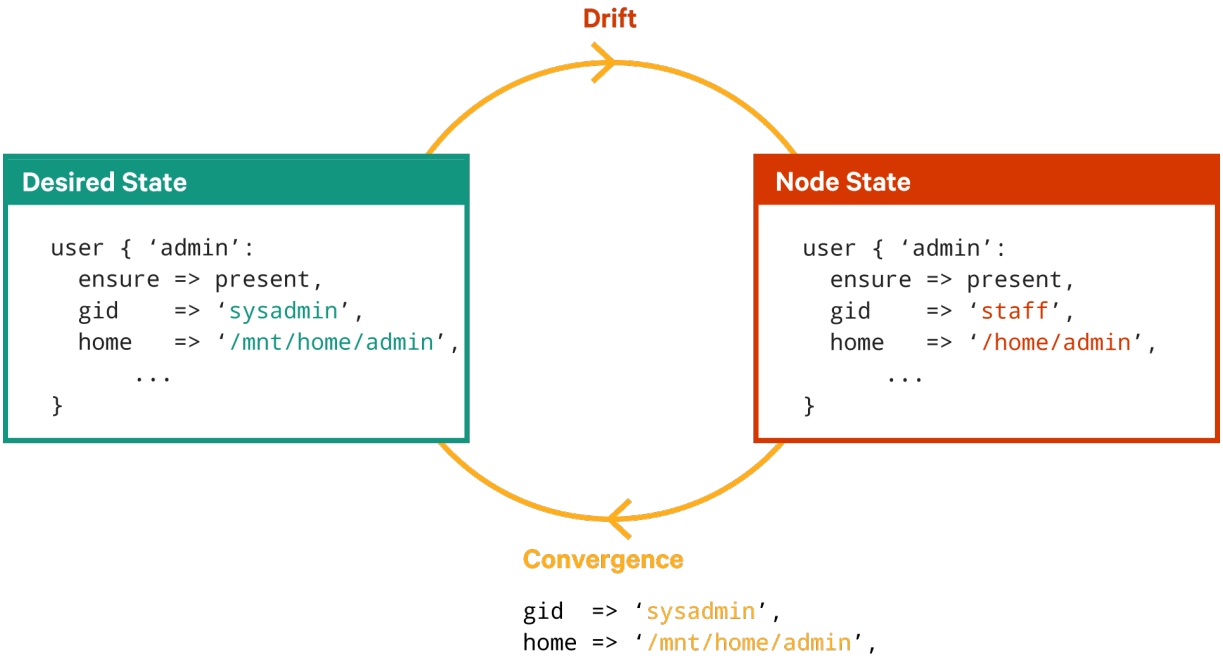
Inspect resources with `puppet resource`

The `puppet resource` command:

- Is a command line tool for inspecting Puppet resources on the system.
- Interacts directly with the Resource Abstraction Layer (RAL).
- Returns the Puppet DSL representation of the current state of a resource.

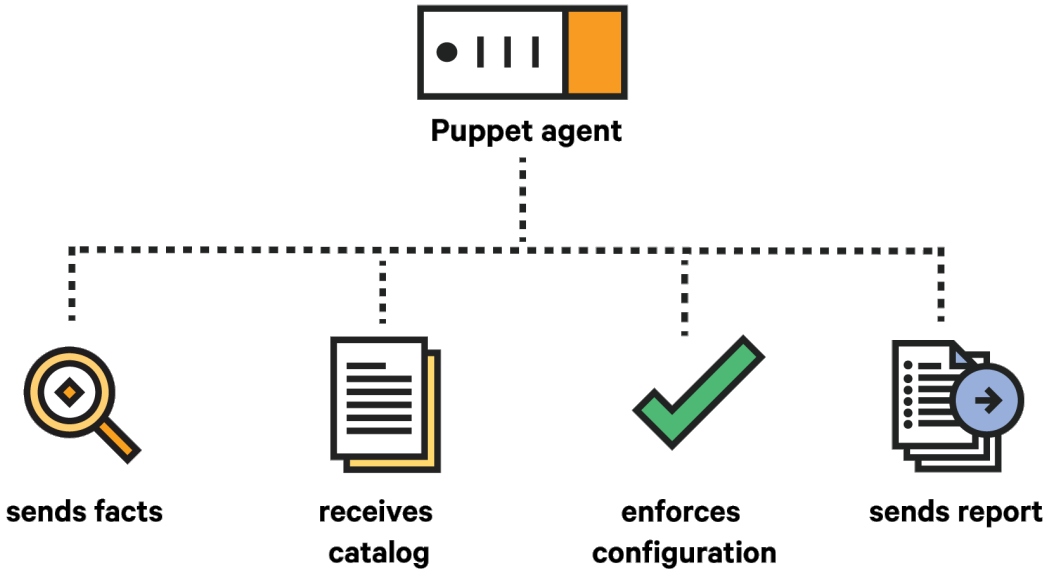
Desired State

Describe the state you want.



drift
Changes from the desired state. Drift can be unauthorized changes, it can be the difference between desired state and a base OS during initial provisioning, it might even be updates to the Puppet codebase changing the desired state, which will cause changes in the managed nodes to match.

What is the Puppet Agent?



- The Puppet agent is a service or daemon that runs on managed nodes.
- Contains Puppet's main code and all of the dependencies needed to run it.
- Distributes the load of bringing nodes in to compliance

It includes Facter, Hiera, and bundled versions of Ruby and OpenSSL. It also includes the PXP Agent which allows you to run tasks through the Puppet orchestrator, over the pcp transport. Once it's installed, you have everything you need to run the Puppet agent service and invoke `puppet` from the command-line.

Puppet agent or Bolt

Puppet agent	Puppet Bolt
Agent installed on managed nodes	Remote control without an agent
Reports results to a central DB	Reports results to standard out
Manage entire system state	Issue ad-hoc commands and scripts
Runs continually to enforce state	Run once, as needed
Declarative	Imperative

These tools solve different problems.

The Puppet agent continuously manages configurations on your nodes. It uses a central Puppet master server to compile and distribute configuration catalogs. The Puppet agent run as a service or daemon on your nodes.

Because the Puppet agent runs in the background, it is always enforcing the desired state of your systems. Using to a central infrastructure, it is able to report changes made, and the current state of all facts on your systems, into a single data store. This information can be queried, or automatically processed as it arrives.

Puppet agent or Bolt snippet examples

Puppet enforced every agent run

```
package { 'openssh':
  ensure => installed,
}
file { '/etc/ssh/sshd_config':
  ensure  => file,
  source  => 'puppet:///ssh/sshd_config',
  require => Package['openssh'],
}
service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

Bolt enforced when `bolt task run` executed

```
URL='https://target.example.com/configs/sshd_config'
yum install openssh
wget $URL -O /etc/ssh/sshd_config
systemctl start sshd
```

Most resources will not be this static, this just a simplified example

Introduction to Bolt Apply

Apply Puppet code to nodes using Bolt

Features:

- No pre-installed Puppet agent needed
  - (Automatically installs one on first run on the target)
- All required code is copied to target
- Full access to facts and Puppet DSL
- Leverage existing Puppet Forge modules

Applying Puppet Code with a plan

Apply Puppet code to nodes using Bolt

```
plan profiles::nginx_install(
  TargetSpec $servers,
  TargetSpec $lb,
  String $site_content = 'hello!',
) {
  if get_targets($lb).size != 1 {
    fail("Must specify a single load balancer, not ${lb}")
  }
  # Ensure puppet tools are installed and gather facts for the apply
  apply_prep([$servers, $lb])

  apply($servers) {
    class { 'profiles::server':
      site_content => "${site_content} from ${$trusted['certname']}",
    }
  }

  # ...
  # ...
}
```

This is the first half of a plan that uses the `apply` function to apply Puppet manifest code to a set of target nodes. The plan is too long to fit on one slide, and it's continued on the next slide.

The `apply_prep` function installs the Puppet agent on the target nodes and gathers facts for compiling the node catalog.

This plan is designed to configure both a load balancer and backend web servers. When it is invoked with `bolt plan run`, the plan applies the `profiles::server` class to the web servers and installs and configures HAProxy on the load balancer. A sample command might be:

```
bolt plan run profiles::nginx_install servers=web1.mydomain.com,web2.mydomain.com
lb=www.mydomain.com
```

Notice how the plan code combines conditionals, use of the `apply` function to apply Puppet code to target nodes, function calls, iterators and other Puppet language features.

The `TargetSpec` parameters to the plan correspond to command line arguments that can be supplied to the `bolt plan run` command. Notice the `TargetSpec $servers` parameter to the plan and the `servers=web1.mydomain.com,web2.mydomain.com` command line argument. Plan parameters may also include a default value which will be assigned if the parameter is not supplied on the command line.

Applying Puppet Code with a plan

Apply Puppet code to nodes using Bolt (continued)

```
plan profiles::nginx_install(
  TargetSpec $servers,
  TargetSpec $lb,
  String $site_content = 'hello!',
) {
  # ...
  # ...

  apply($lb) {
    include haproxy
    haproxy::listen { 'nginx':
      collect_exported => false,
      ipaddress        => $facts['ipaddress'],
      ports            => '80',
    }

    $targets = get_targets($servers)
    $targets.each |$target| {
      haproxy::balancermember { $target.name:
        listening_service => 'nginx',
        server_names      => $target.host,
        ipaddresses       => $target.facts['ipaddress'],
        ports             => '80',
        options           => 'check',
      }
    }
  }
}
```

Conclusion

- Puppet lets you define the desired state of resources. (The **what**)
- The Puppet agent will enforce the desired state on each node. (The **how**)

Checkpoint: Declarative Puppet

Tasks are an example of an imperative workflow.

- True
- False

Which configuration is best suited for declarative use of Puppet?

- Temporarily disable a service during maintenance
- Install the Puppet agent on nodes
- Execute ad-hoc commands on multiple nodes
- Ensure all nodes are using the same NTP configuration

An imperative set of instructions describes the desired state and relies on the tools to enforce state.

- True
- False

Save

Set Up Your Development Environment

Lesson 5: Set Up Your Development Environment

Objectives

At the end of this lesson, you will be able to:

- Describe the tools used to install the Puppet agent.
- Use Bolt to deploy the Puppet agent to multiple nodes.
- Connect and authorize agents to the Puppet master.
- Use a cross-platform editor for creating and editing Puppet code.

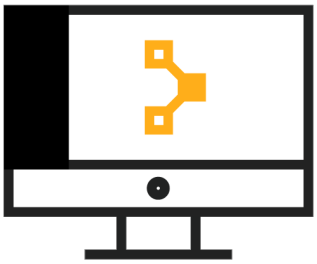
Creating Your Own Puppet Development Environment

Environment prerequisites:

- Name Resolution – DNS or `hosts` file
- Centralized time (NTP) – For certificate support, logs, and event correlation
- A development workstation – (Windows, Linux or MacOS)
  - A code editor to write Bolt tasks and Puppet code
  - With source code management support - (git, etc.)
  - Preferably multiplatform
  - Puppet Development Kit
  - Bolt
- Puppet master – The central puppet server which also functions as a certificate authority
- Node(s) – to manage with the Puppet agent installed
- Source code repository (Github, bitbucket, etc.)

Agents use DNS to resolve the address of the master. By default, they look for a host named "puppet". If your master is named differently, simply set it in the agent's `puppet.conf` file. Nodes installed using the Puppet Enterprise curl- or PowerShell-based installers automatically set this value for you.

The Lab Environment



- Network Access
  - Windows: RDP over port 3389
  - Linux: SSH over port 22
- Node running Puppet agent
  - Dedicated AWS VM instance
- Puppet agent node is preconfigured with the following tools installed:
  - pe-client-tools
  - Git
  - Visual Studio Code (Windows only. Linux will use vi)

The Puppet master is already set-up, in the classroom.

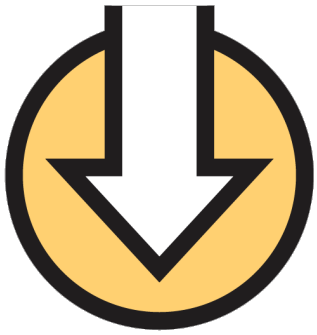
The Agent Service

Puppet agent runs on all managed nodes

- It is responsible for:
  - Gathering facts via `facter` and sending them to the Puppet master
  - Enforcing configuration state assigned by the Puppet master
  - Sending runtime reports after each run
- Supported agent platforms and devices include:
  - Linux (RHEL, Debian, and several other distributions)
  - Windows
  - Solaris
  - MacOS
  - AIX
  - Select network devices (Arista EOS, Cumulus Linux, etc.)

There is only a small set of devices that allow us to run the Puppet agent natively.

Downloading & Installing the Puppet Agent



Puppet agent

- Works with Windows or *nix systems
- No particular hardware requirements

Puppet server

- Pre-installed in our class lab
- 16+ cores and 32 GB of RAM for a monolithic Puppet master for up to 4,000 nodes

Always refer to the official documentation for the latest requirements.

Puppet system requirements (Select the target version from the drop-down)

Linux systems

- Install a release package to enable Puppet Platform repositories.
- Confirm that you can run Puppet executables.

The location for Puppet's executables is `/opt/puppetlabs/bin/`, which is not in your `PATH` environment variable by default.

The executable path doesn't matter for Puppet services — for instance, `service puppet start` works regardless of the `PATH` — but if you're running interactive puppet commands, you must either add their location to your `PATH` or execute them using their full path.

To quickly add the executable location to your `PATH` for your current terminal session, use the command `export PATH=/opt/puppetlabs/bin:$PATH`. You can also add this location wherever you configure your `PATH`, such as your `.profile` or `.bashrc` configuration files.

Install the `puppet-agent` package on your Puppet agent nodes using the command appropriate to your system: * Yum – `sudo yum install puppet-agent` * Apt – `sudo apt-get install puppet-agent` * Zypper – `sudo zypper install puppet-agent`

(Optional) Configure agent settings.

For example, if your master isn't reachable at the default address, `server = puppet`, set the `server` setting to your Puppet master's hostname. For other settings you might want to change, see a list of agent-related settings.

Start the puppet service: `sudo /opt/puppetlabs/bin/puppet resource service puppet ensure=running enable=true`

(Optional) To see a sample of Puppet agent's output and verify any changes you may have made to your configuration settings in step 5, manually launch and watch a Puppet run: `sudo /opt/puppetlabs/bin/puppet agent --test` Sign certificates on the certificate authority (CA) master.

On the Puppet master:
PE 2018.X or earlier - Run `sudo /opt/puppetlabs/bin/puppet cert list` to see any outstanding requests. Run `sudo /opt/puppetlabs/bin/puppet cert sign <NAME>` to sign a request.

PE 2019.X or later - Run `sudo /opt/puppetlabs/bin/puppetserver ca list` to see any outstandindg requests. Run `sudo /opt/puppetlabs/bin/puppetserver ca sign <NAME>` to sign a request.

As each Puppet agent runs for the first time, it submits a certificate signing request (CSR) to the CA Puppet master. You must log into that server to check for and sign certificates. After an agent's certificate is signed, it regularly fetches and applies configuration catalogs from the Puppet master.

Windows:

Download the Windows `puppet-agent` package Puppet's Windows packages can be found here. You need the most recent package for your OS's architecture:

- 64-bit versions of Windows must use `puppet-agent-<VERSION>-x64.msi`
- 32-bit versions of Windows must use `puppet-agent-<VERSION>-x86.msi`

Note: Puppet agent will prevent you from running the 32-bit version on a 64-bit Windows system, this functionality was deprecated in Puppet 4, and removed in Puppet 5.

These packages bundle all of Puppet's prerequisites, so you don't need to download anything else. The list of Windows packages might include release candidates, whose filenames have something like `-rc1` after the version number. Use these only if you want to test upcoming Puppet versions.

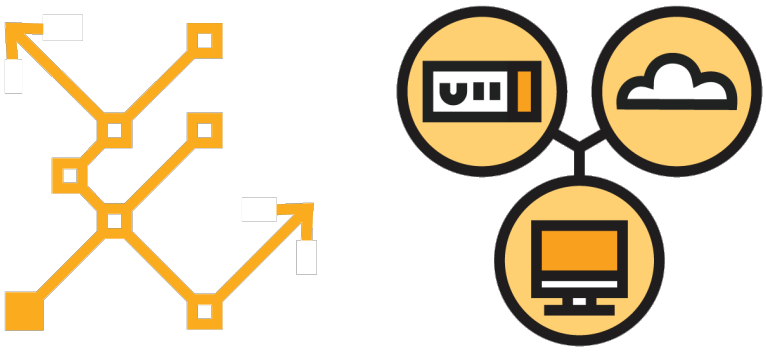
Install Puppet
You can install Puppet with a graphical wizard or on the command line. The command line installer provides more configuration options.

Graphical installation:
Double click the MSI package you downloaded and follow the graphical wizard. The installer must be run with elevated privileges. Installing Puppet does not require a system reboot.

During installation, Puppet asks you for the hostname of your Puppet master server. This must be a *nix node configured to act as a Puppet master. For standalone Puppet nodes that won't connect to a master, use the default hostname (puppet). You might also want to install on the command line and set the agent startup mode to Disabled.

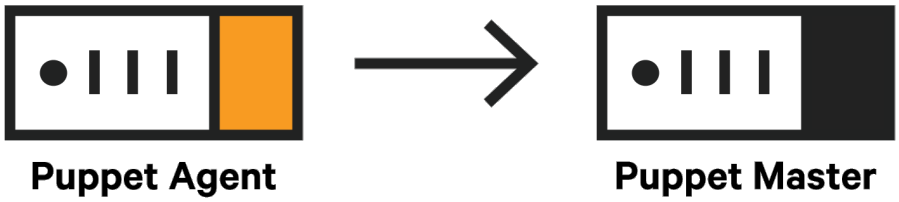
Once the installer finishes, Puppet will be installed, running, and partially configured.

Deploying the Agent on Existing Infrastructure



- You can install the Puppet agent on existing infrastructure with Puppet Bolt.
- Puppet has published a task on Github. https://github.com/puppetlabs/puppetlabs-bootstrap

Registering the Agent with the Puppet Master



The Bolt installation method will initiate the agent registration with the master.

You can accept the agent's newly-generated certificate from the console UI or from the command-line of the master.

On the Puppet master (Puppet Enterprise 2018.1):

`sudo puppet cert list` `sudo puppet cert sign <puppet_agent_hostname>`

On the Puppet master (Puppet Enterprise 2019.x):

`puppetserver ca list` `puppetserver ca sign --certname <puppet_agent_hostname>`

On the Puppet agent's next run, it will fetch the signed certificate:

`[sudo] puppet agent --test`

Puppet authorizes agents and secures communication between the agent and master via SSL. To manage nodes with Puppet master, the agent must send a certificate signing request to the master and you must approve the node's certificate signing request on the master.

When you install a new Puppet agent, during the first run, the agent automatically submits a certificate signing request **(CSR)** to the master.

Certificate requests can be signed from the console or the command line.

After approving a node request, the node doesn't show up in the console until the next Puppet run, which can take up to 30 minutes. You can manually trigger a Puppet run if you want the node to appear immediately.

To accept or reject CSRs in the console or on the command line, you need the permission "Certificate requests: Accept and reject". To manage certificate requests in the console, you also need the permission "Console: View".

Command line:

You can view, approve, and reject node requests using the command line.

To view pending node requests on the command line:

For Puppet Enterprise 2018.1 or earlier

```
$ sudo puppet cert list
```

To sign a pending request:

```
$ sudo puppet cert sign <hostname>
```

For Puppet Enterprise 2019.x or later

```
$ sudo puppetserver ca list
```

To sign a pending request:

```
$ sudo puppetserver ca sign <hostname>
```

The Master Service

`pe-puppetserver` runs on the central server.

It is responsible for:

- Authenticating agent connections.
- Signing certificates.
- Serving a compiled catalog to the agent.
- Serving files.
- Processing posted reports.

Supports several Linux distributions.

Runs under JRuby on the JVM for increased performance at scale.

> Note: Running the Puppet master on the JVM allows it to scale linearly and behave more predictably.

Scaling linearly means that supporting 2,000 nodes requires roughly twice the resources that supporting 1,000 nodes would. This was not true with the legacy Passenger based Puppet master.

See the Puppet system requirements in the official documentation and select the desired Puppet version.

Visual Studio Code

- FREE cross-platform opensource editor made by Microsoft
- Runs on Windows, MacOS, and Linux
- Includes built-in Git integration, and also connects to many other hosted SCM services
- For Puppet language support, install the official Puppet Visual Studio Code extension `jpogran.puppet-vscode` from the Marketplace at: https://marketplace.visualstudio.com/items?itemName=jpogran.puppet-vscode

Most other popular editors also have varying levels of Puppet syntax highlighting available.

Using a common code editor across your team has numerous advantages. It supports a number of different programming languages. It can highlight syntax and help you recognize syntax errors. There are numerous code editors, but we are just discussing Visual Studio. Feel free to check out different editors to see which one is best for you.

Some of the benefits of Visual Studio Code:

- Multiplatform - Runs on Windows, MacOS, and Linux.
- Git integration – Easily use Git and other SCM providers to review diffs, stage files, and make commits from the editor. Push and pull from many hosted SCM services.
- Extensible - Install extensions to add new languages, themes, debuggers, and to connect to additional services.
- IntelliSense - beyond syntax highlighting and autocomplete, IntelliSense provides smart completions based on variable types, function definitions, and imported modules.

Lab 5.1: Agent deployment

- Objectives:
  - Validate lab environment
  - Install Puppet agent
  - Register with master server
  - Validate installation

No handouts for this lab.

Lifecycle of a Puppet Agent Run

Lesson 6: Lifecycle of a Puppet Agent Run

Objectives

At the end of this lesson, you will be able to:

- Describe the role of Facter.
- Describe the order of operations that occur when the Puppet agent runs.
- Identify Puppet resources.

A look at how definitions are used to automatically configure and manage IT infrastructure:

1. The Puppet agent on the node tells the Puppet master information about itself (hostname, node name, operating system, etc.).
2. The Puppet master looks up the configuration for that node, compiles, and sends a catalog representing that intended configuration back to the node.
3. The node reports back any actions that were taken to enforce that configuration.
4. The Puppet master server aggregates all the reports from all the nodes and provides a single overview on the state of your infrastructure.

A real world analogy might be that of an architect designing a new building. First, information (facts) about the site are gathered. This would be information such as the water table, or the annual snowfall, etc.

Then the architect will use that information along with the building specs to create a blueprint (catalog). This blueprint describes how the building should be constructed, but it doesn't have step-by-step instructions. Instead, it trusts that the workers know how to build a wall and lay the foundation.

That blueprint is delivered to the construction company (the Puppet agent), which uses it to build as needed to specifications.

Agent Scheduling

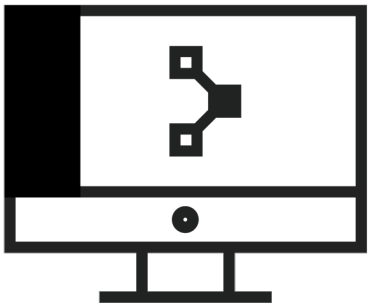
- The agent process performs an automatic run every 30 minutes (by default) to enforce configuration
- Trigger an immediate run on a node's command-line with `puppet agent -t`
- Or, trigger immediate runs through the Enterprise console, in the "Run > Puppet" tab

---

The default interval of thirty minutes can be changed in the `puppet.conf` file on an agent node. Simply add or update the `runinterval` setting. Each agent may have its own distinct setting, if desired.

https://puppet.com/docs/puppet/latest/configuration.html#runinterval

Running Jobs from Puppet Enterprise Console



- The Puppet Enterprise Console can save you time when running jobs.
- No need to be in the Jobs section of the console to run a Puppet job.

You can create jobs from the following pages:

Overview: This page shows a list of all your managed nodes and gathers essential information about your infrastructure at a glance.

Events: Events let you view a summary of activity in your infrastructure, analyze the details of important changes, and investigate common causes behind related events. For instance, let's say you notice run failures because some nodes have out-of-date code. Once you update the code, you can create a job target from the list of failed nodes to be sure you're directing the right fix to the right nodes. You can create new jobs from the **Nodes with events** category.

Classification node groups: Node groups are used to automate classification of nodes with similar functions in your infrastructure. If you make a classification change to a node group, you can quickly create a job to run Puppet on all the nodes in that group, pushing the change to all nodes at once. You must have permissions for running jobs and PQL queries.

In the console, in the **Run** section, click **Puppet**. At this point, the list of nodes is converted to a new Puppet run job list target.

In the **Job description** field, provide a description that will be shown on the job list and job details pages.

Select an environment:

- **Run nodes in their own assigned environment**: Nodes will run in the environment specified by the Node Manager or their puppet.conf file.
- **Select an environment for nodes to run in**: Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

> If your job target includes application instances, the selected environment will also determine the dependency order of your node runs.

Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:

- **No-op**: Simulate a Puppet run on all nodes in the job without enforcing a new catalog.

- **Override noop = true configuration**: If any nodes in the job have `noop = true` set in their puppet.conf files, Puppet will ignore that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.

Do not change the Inventory from **Node list** to **PQL query**. This will clear the node list target. Click **Run job**.

Task Scheduler in Puppet Enterprise Console

Starting with PE 2019:

- Schedule Puppet tasks in the PE console.
- Run tasks on a one-time schedule.
- Set schedules in a graphical interface.

With future releases, look for improvements in this feature and extended capabilities.

Stopping a job

When stopping jobs from the "Job details" page:

- Jobs already underway will finish.
- Job runs that have not yet started will be cancelled.

Stopping tasks depends on how many nodes are targeted in the job versus the concurrency limit you've set. If the concurrency limit is higher than the number of nodes targeted, all nodes will complete the task, as those nodes will have already started the task.

To stop a job: In the console, navigate to the **Job details** page for the job you want to stop, and click **Stop**. On the command line, press **CTRL + C**.

Puppet Resources

Something that Puppet can manage

resource
A Puppet resource is a single item that Puppet knows how to manage or to interact with. This can be a package, a user, a file, or even a single line in a file--like a host entry.

Inspect resources with `puppet resource`

- A command line tool for inspecting Puppet resources on the system.
- It interacts directly with the Resource Abstraction Layer (RAL).
- Returns the Puppet code representation of the current state of a resource.

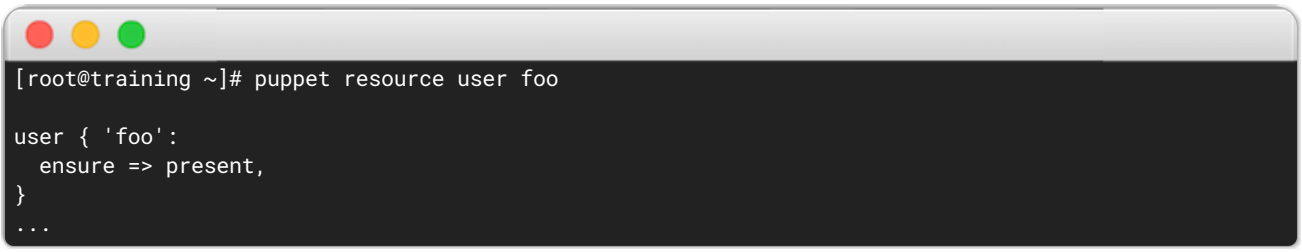
Puppet Resource Query

Retrieve the state of a resource

The Puppet resource command takes one or two arguments:

1. `<resource type>`
2. `<resource title>` (optional)

Returns the current state of a resource.



Origin of Resource Types

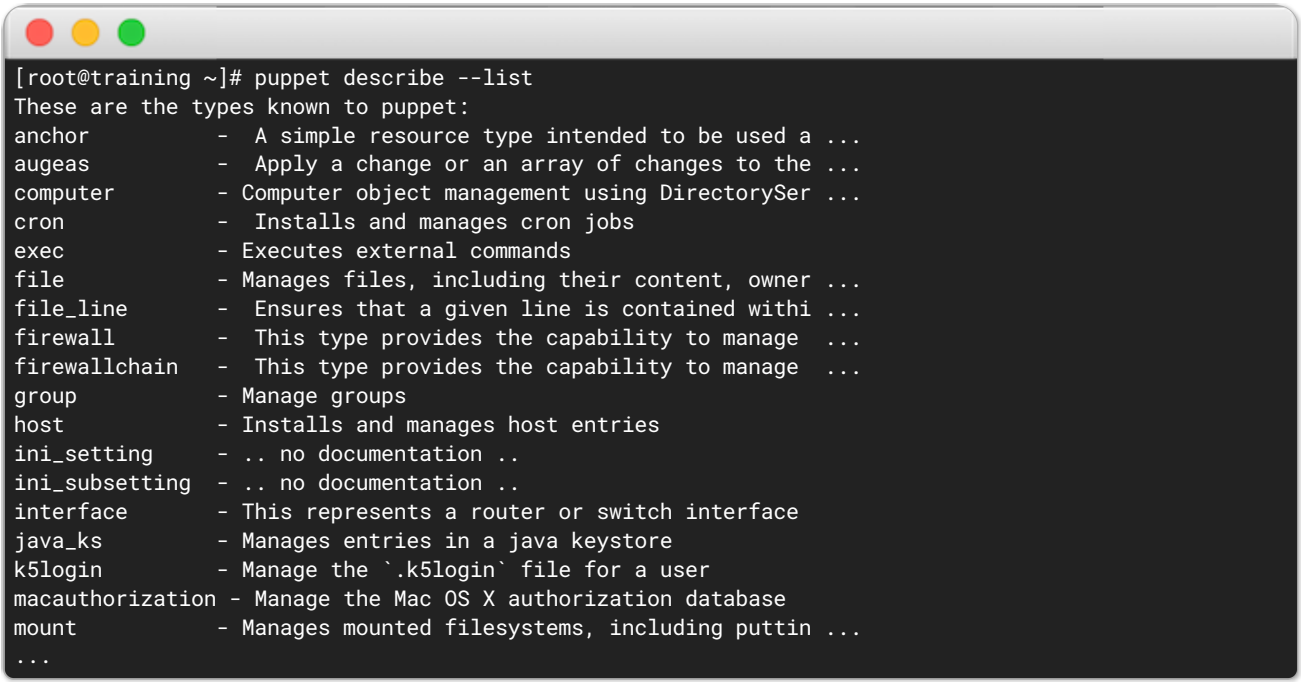
- Sample Core Resource Types
  - `user`
  - `file`
  - `package`
  - `service`
  - `yumrepo`
- Resource Types may come from modules
  - `file_line`
  - `ini_setting`
  - `java_ks`
  - `mysql_database`
  - `reboot`

The Puppet Forge is a great community site for sharing modules. You'll be able to find modules others have written to manage things as disparate as Linux `sysctl` settings to the Nginx webserver or the Drupal content management system.

We will explore the Puppet Forge later in this course.

Resource Type Listing

Display all the installed resource types.



The first step when trying to manage something with Puppet is to figure out what resource type to use. List out the resource types you've already got installed and see if there's something that meets your needs. If not, then you'll search the Forge for types, such as MySQL database management. You will often find that types to manage the resources you need have already been written for you.

This is a very abbreviated example of some interesting resource types.

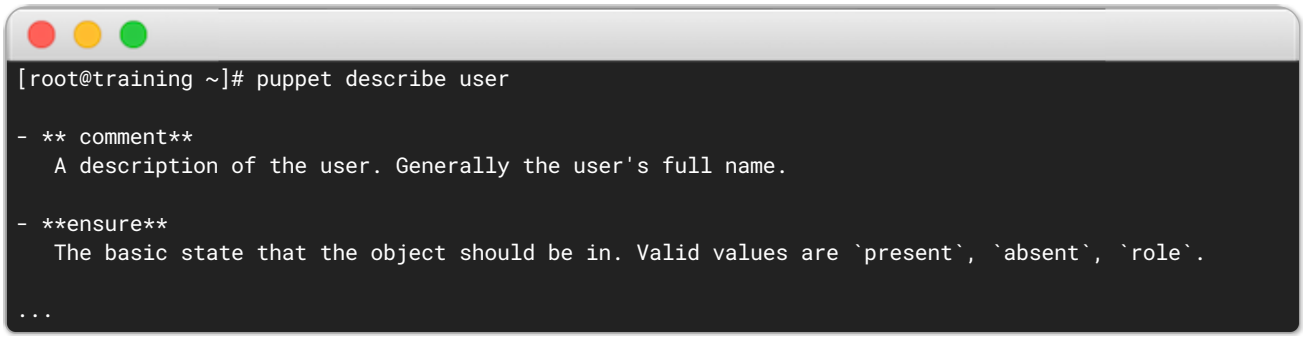
We will cover the Forge community site in a later lesson.

User Resource

Sample attributes

- `uid` : The user's uid number.
- `groups` : List of groups that this user belongs to.
- `home` : The user's home directory.
- `shell` : The user's login shell.

Want to know more?



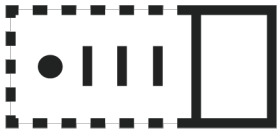
```
[root@training ~]# puppet describe user

- ** comment**
   A description of the user. Generally the user's full name.

- **ensure**
   The basic state that the object should be in. Valid values are `present`, `absent`, `role`.

...
```

Puppet describe takes a resource type as an argument. It returns detailed documentation on that specific resource type and is generated from the same source that we use for http://docs.puppet.com/references/latest/type.html.

Modify Puppet Resources

- Modify resources using the `puppet resource` command.
- We demonstrate this in the Puppet resource lab.

Simulating Change with Puppet



- `--noop`, as in 'no operation' runs in a dry-run mode.
- Allows you to simulate change without enforcing it.
- Reports back a list of potential changes that would have needed to take place if an actual agent run executes.

`--noop` gives you an opportunity to test the effect of an update without actually applying it.

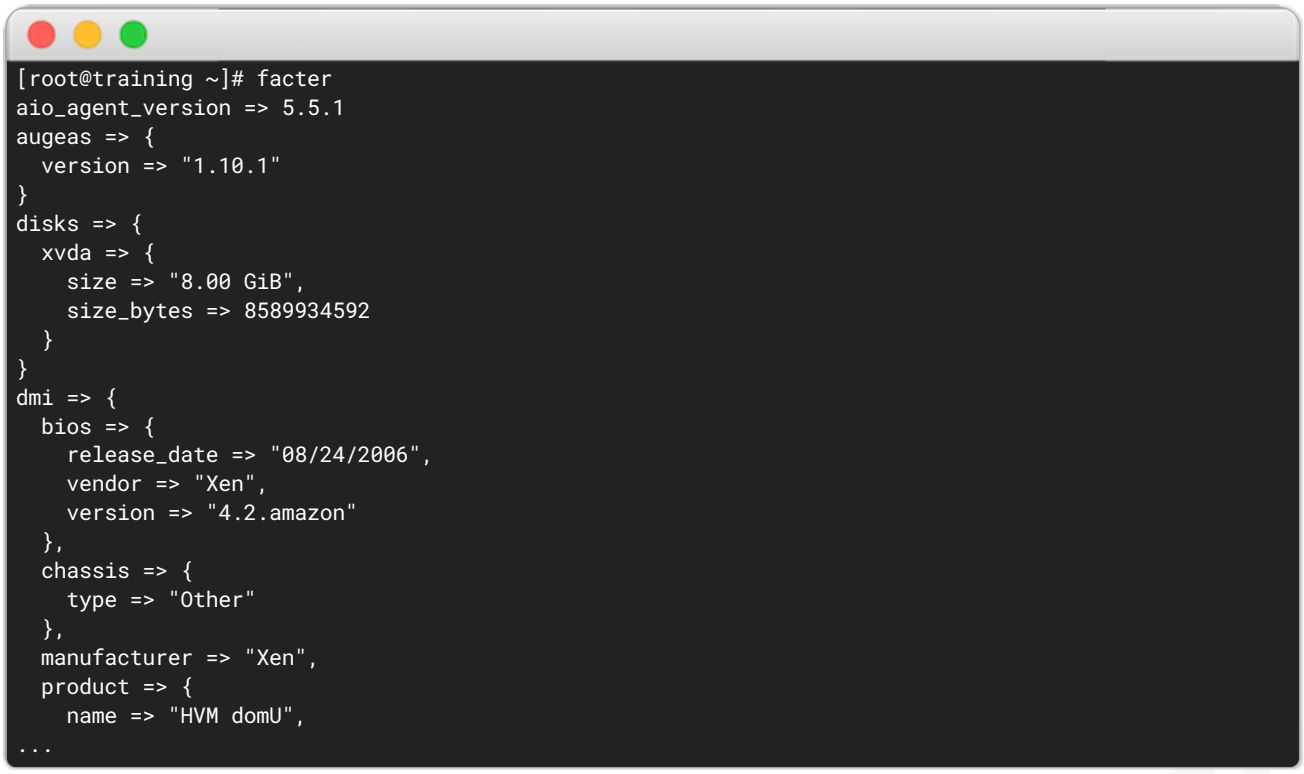
Lab 6.1: Puppet resources

- Objectives:
  - Run the `puppet resource` command to inspect local user accounts.
  - Run the `puppet resource` command to inspect local user groups.
  - Change a local user or group with the `puppet resource` command.
  - Inspect the state of the time service configured in a previous lab.

Facter

Gathers information about a node

Puppet gathers facts using `facter` as one of the first steps of a run. Facts are used by the master during catalog compilation.



```
[root@training ~]# facter
aio_agent_version => 5.5.1
augeas => {
  version => "1.10.1"
}
disks => {
  xvda => {
    size => "8.00 GiB",
    size_bytes => 8589934592
  }
}
dmi => {
  bios => {
    release_date => "08/24/2006",
    vendor => "Xen",
    version => "4.2.amazon"
  },
  chassis => {
    type => "Other"
  },
  manufacturer => "Xen",
  product => {
    name => "HVM domU",
...
```

Facter is Puppet's system inventory tool. Facter discovers facts intrinsic to a node (such as its hostname, network interfaces and IP addresses, operating system, etc.) and makes them available to Puppet.

Facter includes a large number of built-in facts. You can view their names and values for the local system by running `facter` at the command line. In agent/master Puppet arrangements, agent nodes send their facts to the master, and the master compiles the catalog using these facts.

Facts are always generated prior to the agent run. You cannot change facts during compilation and your catalog cannot use facts to make conditional decisions on the agent during application. We will talk later about how to use conditionals to change how the catalog is built.

Newer versions of `facter` present facts as structured data, meaning that some facts will return array or hash data objects instead of just simple strings.

To see the older simple string facts, pass the `--show-legacy` command line option.

External Facts

External facts are simple ways to customize the facts for a system.

- Static files of information
- Scripts that return certain data structures when called

External facts can be used like inventory tags (what data center is this node in?) or to lookup additional information from a system that you want to expose on the master or use to decide what catalog information to include.

There are two main types of external facts: text files and scripts.

The first allows you to add your own facts from a static file -- either plain text key=value pairs, YAML, or JSON format. On Linux, these static files should be placed under `/etc/puppetlabs/facter/facts.d`. On Windows, they go in `C:\Program Data\Puppetlabs\facter\facts.d`.

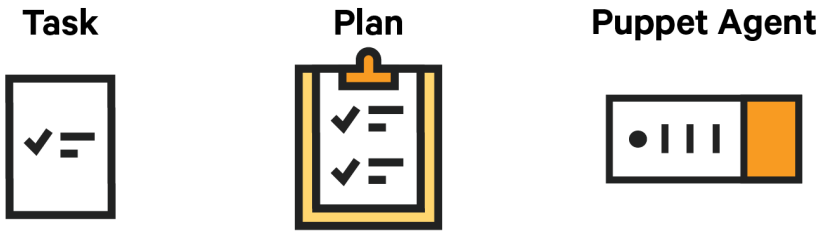
```
$ sudo mkdir -p /etc/puppetlabs/facter/facts.d
$ echo 'external_fact=yes' | sudo tee /etc/puppetlabs/facter/facts.d/external_test.txt
external_fact=yes
$ facter external_fact
yes
```

Additionally, scripts may be used as external facts as long as the node can execute them, and they return a key=value pair, YAML, or JSON. Linux scripts must include a "shebang" on the first line and be executable. Windows looks for files ending in .bat or .ps1 to determine if an external fact is a script to be executed.

Lab 6.2: Using and extending facter

- Objectives:
  - Run `facter` to display core fact values.
  - Install an external fact script.
  - Run `facter` to display an external fact value.
  - Display fact values in the PE console.

Task vs. Agent



- Bolt executes commands across multiple systems through:
  - Bolt `run`: one time/ad hoc
  - Bolt `apply`: one time desired state using Puppet DSL
- Puppet agent is used for continuous application of desired state using Puppet DSL

Bolt and tasks use an imperative model to execute commands.

The agent uses a declarative model.

Imperative models revolve around ad hoc tasks, meaning if you want to generate 15 items, it may create 15 new objects. Now you may wind up with 30 objects. In a declarative model (the Puppet agent), you have 15 objects and only make changes necessary to increase or decrease the objects to ensure you have exactly 15.

Checkpoint: Puppet Agent Lifecycle

What did we learn?

- What are the installation requirements for the Puppet agent?
- What are the installation requirements for Puppet server?
- At what interval does the Puppet agent report to the puppet server?

Using Puppet Modules from the Forge

Lesson 7: Using Puppet Modules from the Forge

Objectives

At the end of this lesson, you will be able to:

- Define a Puppet module.
- Explain how to locate existing Puppet modules.
- Explain how to evaluate Puppet modules.

Modules

Modules

- Are self-contained bundles of code and data
- May be downloaded as existing packages from the Puppet Forge
- May be custom written for your environment
- Are designed to encapsulate all of the components related to a given configuration in a single folder hierarchy

Modules have a pre-defined structure that enables the following:

- Auto-loading of classes
- File-serving for templates, tasks, and files
- Auto-delivery of custom Puppet extensions
- Easy sharing with others

Modules should be self-contained and should have well defined integration points for other modules to use. Each module should manage everything to do with the thing that it is managing, and--more importantly--should not manage things that don't fall within its scope. For example, a web app should not manage the MySQL or Apache configuration because then you could ever only use one at a time.

Learning how to appropriately define layers of abstraction is a skill that comes with practice.

The Puppet Forge

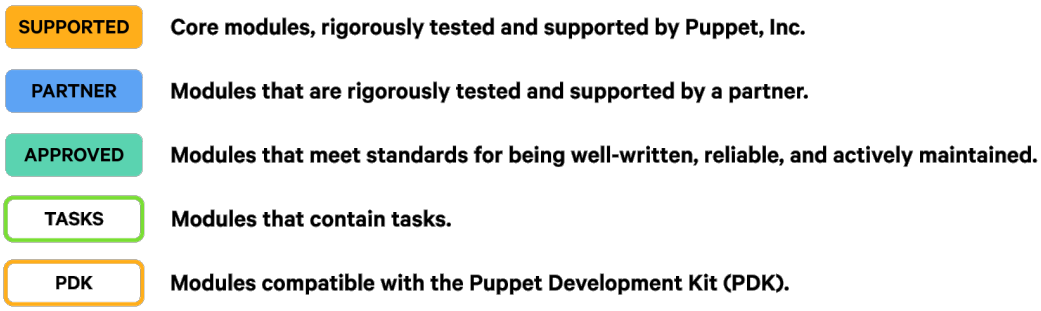
- The standard repository for pre-built Puppet modules
- Provides robust searching and filtering capabilities
- Features Puppet-supported and Puppet-approved modules
- Integrates with Puppet Development Kit (PDK)

http://forge.puppet.com

The Puppet Forge has component modules that are ready to use. You just have to download and, if needed, configure them for your invironment.

Puppet Module Badges

Modules on the Forge may have one or more badges to guide you when selecting which ones to use.



Puppet Approved and Supported modules are recommended by Puppet for use with Puppet Enterprise and meet our expectations for quality and usability.

Puppet ensures that Puppet Approved modules:

- Solve a discrete automation challenge and are developed in accordance with module best practices
- Adhere to Puppet's standards for style and design
- Have accurate and thorough documentation to help you get started quickly
- Are regularly maintained and versioned according to SemVer rules
- Provide metadata including license, issues url, and where to find source code
- Do not deliberately inject malicious code or otherwise harm the system they're used with

Puppet Approved is a designation given to modules that pass specific quality and usability requirements. These modules are recommended by Puppet, but not supported as part of a Puppet Enterprise license agreement.

Puppet Supported are modules that have been tested with Puppet Enterprise and are fully supported by Puppet and/or a third-party vendor where applicable.

Puppet Forge modules are pretty great at making your life easier. Puppet Supported modules take it one step further, making sure common services are easy to set up, implement, and manage with Puppet Enterprise.

Puppet guarantees that each supported module has been tested with Puppet Enterprise and support will be maintained over the lifecycle, with bug or security patches as necessary.

Module Quality and Community Scores

Puppet modules on the Forge display

- An automatically generated quality score
  - Includes linting, compatibility, metadata
- A community approval rating
- The number of downloads
- Version
- Most recent release date
- Vox Pupuli: https://voxpupuli.org/, https://forge.puppet.com/puppet
  - Collaborators group
  - Maintain 163 community modules
  - Group maintenance of modules

Installing Modules

- To install a module from the Puppet Forge, use the `puppet module install` command with the full name of the module you want.
- The full name of a Forge module is formatted as `username-modulename`.
- The Forge has the syntax for you.
- Modules can also be installed on your master through your control repo or by downloading and expanding the tarball in the `modulepath`.

To install or upgrade a Puppet Enterprise module with the `puppet module` command, you must:

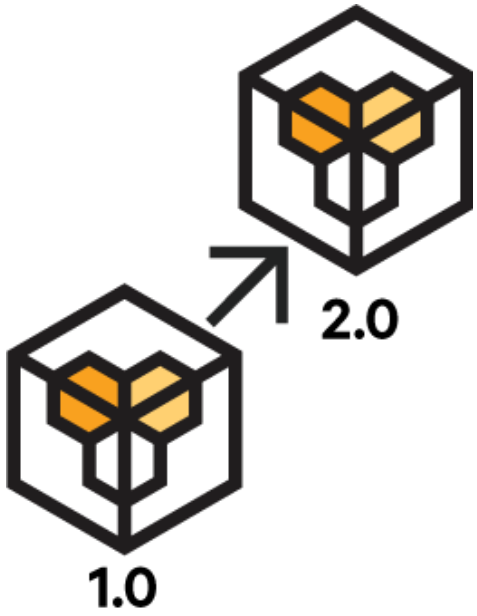
- Run the command as the root user.
- Have internet access on the node you are using to download the module.

Modules can also be installed on your Master through a Control Repo to define all of the modules you use in code for consistent, repeatable Master deployments. For more information on the `Puppetfile` in a control repo, see https://puppet.com/docs/pe/latest/puppetfile.html and an example at https://github.com/puppetlabs/control-repo

Finally, modules may be manually installed in your `modulepath` by downloading the tarball from the Forge and expanding it in the `modulepath`. Run `sudo puppet config print` to see the configured `modulepath` on your system.

> Using the `puppet module` tool will resolve any dependencies of the module. Installing with a Puppetfile or by downloading a tar archive, however, requires that you manage dependencies as needed.

Upgrading Modules



- Use the module tool's `upgrade` action to upgrade an installed module to the latest version.
- Identify the target module by its full name `username-modulename`.

Use the `--version` option to specify a specific version.

Use the `--ignore-changes` option to upgrade the module while ignoring and overwriting any local changes that might have been made.

Use the `--ignore-dependencies` option to skip upgrading any modules required by this module.

Uninstall Modules

- Use the module tool's `uninstall` action to remove an installed module.
- Identify the target module by its full name (`username-modulename`)

Use the module tool's uninstall action to remove an installed module. You must identify the target module by its full name (username-modulename).

```
$ sudo puppet module uninstall apache
Error: Could not uninstall module 'apache':
Module 'apache' is not installed
You may have meant `puppet module uninstall puppetlabs-apache`

$ sudo puppet module uninstall puppetlabs-apache
Removed /etc/puppetlabs/code/modules/apache (v0.0.3)
```

By default, the tool won't uninstall a module that other modules depend on, or whose files have been edited since it was installed. Use the `--force` option to uninstall even if the module is depended upon or has been manually edited. Use the `--ignore-changes` option to uninstall the module while ignoring and overwriting any local changes that might have been made.

Lab 7.1: Puppet forge

- Objectives:
  - Understand how to search for modules on the Puppet Forge.
  - Learn how to install modules from the Puppet Forge on the Puppet master.
  - Classify nodes with installed modules and configure class parameters.
  - Run the Puppet agent manually and observe node configuration changes.

Checkpoint: Puppet Forge

Which is not a module endorsement badge on the Forge?

- Approved
- Supported
- Tasks
- Classes

Community scores are automatically generated.

- True
- False

Modules encapsulate all of the components related to a given configuration.

- True
- False

Save

Creating Wrapper Classes

Lesson 8: Creating Wrapper Classes

Objectives

At the end of this lesson, you will be able to:

- Describe how to use Puppet classes to define custom business logic around existing modules.
- Understand how to customize upstream modules, reduce duplication and simplify node configuration.

Module theory and detailed syntax will be covered later.

Rather than declaring a class or resource from a Forge module and customizing it every time, you can create specify the settings the way you need them, once, then simply `include` that anywhere you need it going forward. This reduces duplication and potential errors.

To do this, we create what's called a wrapper class. At its most simple form, a wrapper class defines a class consisting of other resources and your customized business logic or specific parameters.

Module Customization

- Wrapper classes can save time and simplify configuration management.
- Puppet Forge component modules usually need customization.
- Rather than declaring a class or resource that needs constant updating, define a wrapper class.
- To simplify module creation we will use PDK to easily setup a module structure and then assign our new custom module to nodes.

Rather than declaring a class or resource from a Forge module and customizing it every time, you can create a wrapper class, specify the settings the way you need them, once, then simply `include` that anywhere you need it going forward. This reduces duplication and potential errors.

To do this, we create what's called a wrapper class. At its most simple form, a wrapper class defines a class consisting of other resources and your customized business logic or specific parameters.

Modules

Modules are directory structures of files that contain logic about a configuration. They are designed to encapsulate all of the components related to a given configuration in a single folder hierarchy.

They have a pre-defined structure that enables the following:

- Auto-loading of classes.
- File-serving for templates, tasks, and files.
- Auto-delivery of custom Puppet extensions.
- Easy sharing with others.

Modules should be self-contained and should have well defined integration points for other modules to use. Each module should manage everything to do with the thing that it is managing, and--more importantly--should *not* manage things that don't fall within it's scope. For example, a web app should not manage the MySQL or Apache configuration because then you could ever only use one at a time.

Learning how to appropriately define layers of abstraction is a skill that comes with practice.

Puppet Classes

Classes define a collection of resources that are managed together as a single unit. Wrapper classes are a special type of class.

```
# ${modulepath}/ssh/manifests/init.pp
class ssh {
  package { 'openssh':
    ensure => present,
  }

  file { '/etc/ssh/sshd_config':
    ensure  => file,
    owner   => 'root',
    group   => 'root',
    mode    => '0644',
    source  => 'puppet:///modules/ssh/sshd_config',
    require => Package['openssh'],
  }

  service { 'sshd':
    ensure    => running,
    enable    => true,
    subscribe => File['/etc/ssh/sshd_config'],
  }
}
```

- Stated another way, `package`, `file`, and `service` are individual Puppet resources bundled together to define a single idea, or class.
- Class definitions are contained in manifests. The `init.pp` file above is an example of a manifest written in Puppet DSL.

> Notice the trailing comma after the last attribute in each resource above. This is not required after the last parameter, but is a best practice because it reduces the chance of errors throughout the lifetime of the manifest file and makes comparing changes to modules easier over time.

A good design strategy is to make many smaller classes that represent logical configuration groupings and can be stacked together in different ways. This takes a little more design work up front, but becomes much more maintainable than large monolithic classes very quickly.

Learning how to structure your classes to make them composable in this way is an art that will be improve with practice.

Define and Declare

Now that we have built our class, how do we use it?

define:
To specify the contents and behavior of a class. Defining a class doesn't automatically include it in a configuration; it simply makes it available to be declared.

declare:
To direct Puppet to include or instantiate a given class. To declare classes, use the `include` function. This tells Puppet to evaluate the class and manage all the resources declared within it.

Defining a class is similar to defining a function in a language like Ruby, Python, or C. The function only ever has effect when it is invoked. Similarly, Puppet class definitions don't have any effect until we declare them. Besides the include function, the resource-like `class {'foo':}` syntax can be used. This is how we declare parameterized classes and will be covered in a later section.

You can declare a class, with all defaults using `include`. The `include <class_name>` statement may be used as many times as necessary and it is only applied to the catalog once. In contrast, multiple declarations of `class {'foo':}` will cause an error as a resource may only be included in the catalog exactly once.

Example ideas:

- You may want anyone who is a member of staff group to automatically get sudo access.
- You want to define some logic about logging or time: Network devices should send logs to A while servers should send logs to B.

- You want to enable ssh everywhere but want to customize the default settings... You can't just "include" it because you need to change something so you have to declare it with

  @@@ Puppet class { 'ssh::server': options => { 'PasswordAuthentication' => 'no', 'PermitRootLogin' => 'no', 'Port' => [22, 2222], }, }

Declaring a Class

A class is just another resource!

The `include` function is a shortcut that declares a class with default parameters:

```
include ssh
```

Classes can be declared just like any other resource:

```
class { 'ssh': }
```

Classes can be declared with parameter values:

```
class { 'ssh':
  allow_root => false,  # don't allow root to log in
  untrusted  => false,  # don't allow logins from untrusted hosts
}
```

The `include` function is *idempotent*, meaning that it will declare the class only if it's not already declared. This means that you can include a class any time you know it is needed. For example, if a defined type requires setup from a parent class, it should include that class itself.

Declaring a class with the resource syntax, however, is **not** idempotent. Just like any other resource, you can only declare classes once.

Best practices are to use the `include` function when you can. However, if you must customize parameters then you *should not* use `include` to include that class anywhere else in your codebase. To do so would put you in an indeterminate state that's difficult to debug.

One solution to this conundrum is to write a wrapper class that will declare the parameterized class with the required parameters, but not accept any parameters of its own. Even better would be to use automatic parameter lookup with Hiera, which will be covered in another lesson.

Using Declared Classes

Declare classes where needed.

Classes are unique and will only be used once on a given node.

```
class profile::base {
  # [...]
  include ssh
  include ntp
  include kernel_tuning
}

class profile::hardened {
  # [...]
  include ssh
  include profile::hardening::ssh
  include profile::hardening::network_stack
}
```

The catalog will only contain a single instance of the `ssh` class:

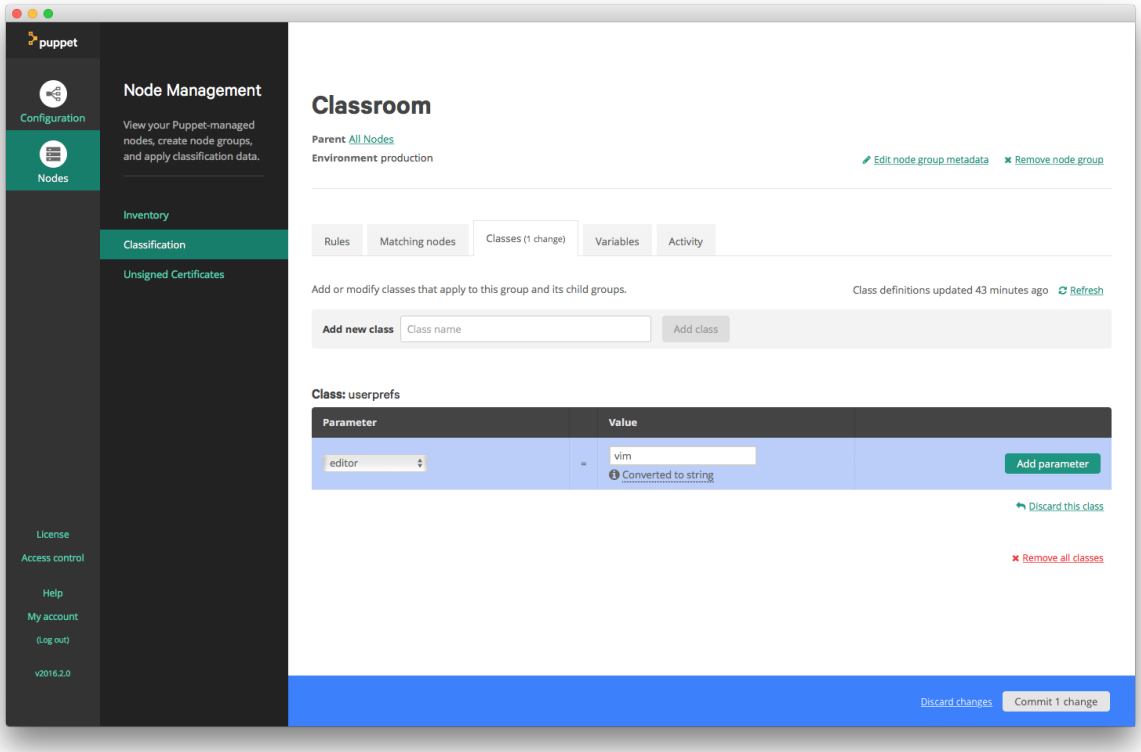
```
node 'example.puppetlabs.vm' {
  include profile::base
  include profile::hardened
}
```

Classes, just like resources, can only be declared once. There can only be one instance of a class in the catalog. The `include` function will declare a class if and only if it hasn't been declared already. It works similarly to the `require_once` function in other languages.

This means that best practices are to include a class when it's going to be used; even though the `include` function may be called many times, the class is only ever actually declared once.

> **Note:** Best practices entail using a `role` class to include `profile` classes. Node definitions should only include `role` classes. We'll talk more about this concept later.

Node Classification in Puppet Enterprise



Node classification is selecting which classes to apply to a node or group of nodes.

It is often much easier for people new to Puppet to use the PE graphical interface to classify nodes.

Puppet Development Kit

Anyone can easily create high-quality modules with Puppet Development Kit (PDK).

PDK tools assist with:

- Creating modules
- Converting modules
- Customizing module configuration
- Validating and testing modules
- Building module packages
- Not just for developers!

PDK provides integrated testing tools and a simple command line interface to help you develop, validate, and test modules.

PDK development workflow

- Create a module or convert an existing module to make it compatible with PDK.
- Validate your module to verify that it is well-formed.
- Unit test your module to verify that all dependencies and directories are present.
- Generate classes, defined types, or tasks in your module.
- Validate and test your module each time you add new functionality.

PDK generates an empty module based on a default template, but you can specify your own custom template with command line options. This saves you from having to create completely new modules with metadata, as well as creating classes, defined types, and tasks in your module! It’s all done for you! Including the directories and file structures Puppet requires. This greatly reduces errors, provides consistency, as well as saves lots of time.

The PDK infrastructure for validating and unit testing modules allows you to have sanity checks, ensuring code compiles correctly.

PDK Installation Requirements

- Before you install Puppet Development Kit (PDK), make sure you meet the system and language version requirements.
- By default, PDK installs to the following locations:
  - On *nix and MacOS systems: `/opt/puppetlabs/pdk/`
  - On Windows systems: `C:\Program Files\Puppet Labs\DevelopmentKit`

If you are using PDK behind a proxy, you must set environment variables to enable PDK to communicate. You can set these variables on the command line before each working session, or you can add them to your system configuration, which varies depending on the operating system.

RHEL or Centos systems: `sudo rpm -ivh pdk-<VERSION>-<PLATFORM>.rpm`

Ubuntu systems: `sudo dpkg -i pdk-<VERSION>-<PLATFORM>.deb`

Windows:

- Download the package for your operating system from the PDK download site: https://puppet.com/download-puppet-development-kit
- Double click on the downloaded package to install.
- Open a new PowerShell window to re-source your profile and make PDK available to your PATH.

MacOS:

- Download and install the package for Mac OS X systems from the PDK download site: https://puppet.com/download-puppet-development-kit
- Double click on the downloaded package to install.
- Open a terminal to re-source your shell profile and make PDK available to your PATH.

Verifying installation: Run `pdk --version` to ensure it is installed and working.

Creating Modules

- PDK generates a complete new module with metadata.
  - For example: `pdk new module <MODULE_NAME>`
- Can create classes, defined types, and tasks in your module.
- It sets up infrastructure for validating and unit testing your module.

To create your module's metadata, PDK asks you a series of questions. Each question has a default response that PDK uses if you skip the question. The answers you provide to these questions are stored and used as the new defaults for subsequent module creations. Optionally, you can skip the interview step and use the default answers for all metadata. For details about editing the metadata.json file, read about module metadata at

- https://puppet.com/docs/puppet/latest/modules_metadata.html

PDK generates an empty module based on a default template. You can specify your own custom template with command line options. To see the complete default module template, see the pdk-template project on GitHub at:

- https://github.com/puppetlabs/pdk-templates

Converting existing modules for use with PDK

- To get existing modules to work with PDK, run the `pdk convert` command.
- When you convert a module, PDK makes changes to it based on the default module template.
- This is the same template that PDK uses when it creates a new module.
- You can customize this template as needed.

If your module already has a `metadata.json` file, the metadata is merged with the default metadata information from the module template. If the metadata does not exist, PDK asks a series of interview questions to create the module's metadata.

PDK then displays a summary of the files that will change during conversion and prompts you to either continue or cancel the conversion. Either way, PDK generates a detailed change report, `convert_report.txt`, in the top directory of the module. This report is replaced by an updated version every time you run the `convert` command.

PDK Tools

PDK includes a variety of development and testing tools built by Puppet and by the Puppet open source community. Most are used via the high-level PDK commands.

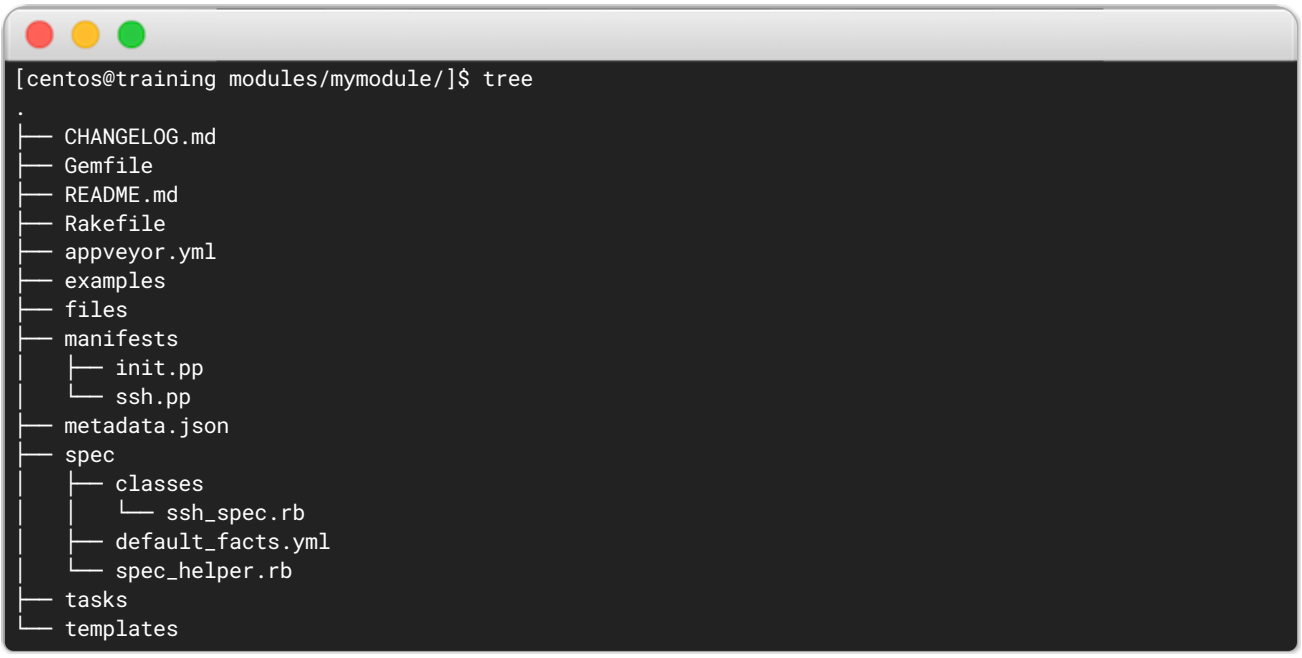
Command	Description
convert	Convert an existing module to be compatible with PDK.
new	create a new module, etc.
update	Update a module that has been created by or converted for use by PDK.
validate	Run static analysis tests.
test	Run tests.
build	Builds a package from the module that can be published to the Puppet Forge.
help	show help
bundle	(Experimental) Command pass-through to bundler
module	Provide CLI-backwards compatibility to the puppet module tool.

PDK includes a variety of pre-configured tools that are used to create, validate, and test modules in a consistent way, making it easier to collaborate with other module developers.

Tool	Description
pdk	Command line tool for generating and testing modules
metadata-json-lint	Validates and lints `metadata.json` files in modules against Puppet's module metadatastyle guidelines.
puppet-lint	Checks your Puppet code against the recommendations in the Puppet Language style guide.
puppet-syntax	Checks for correct syntax in Puppet manifests, templates, and Hiera YAML.
puppetlabs_spec_helper	Provides classes, methods, and Rake tasks to help with spec testing Puppet code.
rspec-puppet	Tests the behavior of Puppet when it compiles your manifests into a catalog of Puppet resources.
rspec-puppet-facts	Adds support for running rspec-puppet tests against the facts for your supported operating systems.

Module Structure

Modules are a file and directory structure that Puppet can use to store config information. Each module subdirectory has a specific function. Not all directories are required, but if used, they should be in the following structure.



For more information about the directory and file structure of modules, see the latest official Puppet documentation online.

PDK Templates

"When PDK creates a new module, how does PDK know what file structure to use for the new module?"

- Answer a series of questions for customization or use defaults.
- Defaults are generated based on PDK Templates.
- PDK Templates can give you a structure based on default settings and can be customized.
- https://github.com/puppetlabs/pdk-templates

There are templates for:

- `moduleroot`
- `moduleroot_init`
- `object_templates`

---

* `moduleroot` templates get deployed on `new module` and `convert`; use them to enforce a common boilerplate for central files. * `moduleroot_init` templates get only deployed when the target file does not yet exist; use them to provide skeletons for files the developer needs to modify heavily. * `object_templates` templates are used by the various `new ...` commands for classes, defined types, etc.

Packaging and Distributing (Sharing) Modules

- Build a module package with the `pdk build` command.
- This enables modules to be uploaded to the Forge.

From the command line, change into the module's directory with `cd <MODULE_NAME>`. Run `pdk build` and respond to any prompts. To change the behavior of the build command, add option flags to the command. For example, to create the package to a custom location, run `pdk build --target-dir=<PATH>`.

PDK builds a package with the naming convention `forgeusername-modulename-version.tar.gz` to the `pkg` directory of the module.

Map Classes to Nodes

Classes can be mapped to nodes with

- Node definitions in a `site.pp` file
- An external node classifier (ENC) such as the Puppet Enterprise Console "Node Classification"

Node Definitions in site.pp

Include node specific configuration

The node definition corresponding to the agent's name is enforced:

```
node 'oscar.example.com' {
  include role::web_app
}
```

- Only one node definition is ever enforced.
- Can match multiple nodes with a regular expression.
- Use a `default` node definition as a fallback.
    - Used when no other node definition matches.

Best practices: only include a single role class in each node definition.

Syntax Note: roles are covered in greater detail later in the course.

An agent node's `certname` is how it is identified in the Puppet network. It is set at install time but can be changed later. The `certname` is usually (but not always) the node's fully qualified domain name.

Best practices are to avoid any complex logic in node definitions and simply include the desired role class. This leads to a configuration model that is more readable and more composable. It also makes the transition to an External Node Classifier like the Enterprise Console a painless process.

That role class might look like:

```
class role::web_app {
  include profile::secure
  include profile::apache
  include profile::mysql
  include profile::web_app
}
```

If matching nodes by regular expression, your node definition might look like:

```
node /^web\d{3}\.example\.com$/ {
  include role::web_app
}
```

When a web application server, identified by a nodename of `webXXX`, connects to the Puppet master, it will be assigned the classes above.

Default Node Definition

If neither a node name or regex match, the default node will apply.

- In many environments, it may be desirable to have the default node fail without declaring any classes.

```
node default {
  fail('No node definitions matched.')
}
```

Node Definitions - Stacking Classes

Multiple classes may be declared together to represent a role

For example, to build a web application from Puppet classes on `oscar.example.com`:

```
node 'oscar.example.com' {
  include ssh
  include apache
  include mysql
  include web_app
}
```

This is a node definition which represents the agent machine and the classes that compose its Puppet configuration. When the node `oscar.example.com` requests a catalog from the master, these classes will be used to build it.

Node definitions can match based on simple strings, like above, or they can match based on regular expressions. Regular expressions are only used when no exact match is found, and they are compared in order until a regex matches, regardless of specificity.

Best practices are to avoid any complex logic in node definitions and simply include the required classes. This leads to a configuration model that is more readable and more composable. It also makes the transition to an External Node Classifier like the Enterprise Console a painless process.

Regular Expressions in Node Definitions

Configure nodes by nodename patterns.

- Regular expressions are only evaluated if no exact match is found.
- Regular expressions can be used to define nodes.
- The first match found is declared, regardless of specificity.

```
node /^web\d{3}\.puppetlabs\.com$/ {
  include ssh
  include apache
  include mysql
  include web_app
}
```

When a web application server, identified by a nodename of `webXXX`, connects to the Puppet master, it will be assigned the classes above.

Remember that regular expressions are not as readable as simple strings. Best practices are to, when possible, minimize the use of regular expressions to make it clear which node definition will be enforced. See https://puppet.com/docs/puppet/latest/lang_node_definitions.html for more information.

Lab 8.1: Create a wrapper module

- Objectives:
  - Use PDK to create a new class and default entries.
  - Fill in the class documentation.
  - Configure the `time` class.
  - Add your module to version control.
  - Publish your module on the classroom version control server.

Checkpoint: Wrapper Classes

Defining a class tells Puppet to manage the included resources on a node.

- True
- False

class { 'ssh': } is an example of

- Defining a class
- Deleting a class
- Validating a class
- Declaring a class

How do you use PDK to work with an existing module?

- pdk convert
- pdk import
- pdk build
- pdk validate

Save

Basic Module Testing

Lesson 9: Basic Module Testing

Objectives

At the end of this lesson, you will be able to:

- Verify that your Puppet module will parse correctly and meets recommended style guidelines before applying to a node.
- Describe how early style and syntax checking improve quality and collaboration as well as reduce errors.
- Explain how to use syntax and style tools included by PDK.
- Describe the benefits of unit tests and system tests when writing modules.

Puppet Style Guide

- The Puppet style guide promotes consistent formatting in the Puppet language.
- This consistency in code and module structure makes it easier to update and maintain the code.

Find the Puppet language style guide at https://puppet.com/docs/puppet/latest/style_guide.html

The Puppet language style guide can be found at https://puppet.com/docs/puppet/latest/style_guide.html

The purpose of the style guide is to promote consistent formatting in the Puppet language. This is especially important when contributing modules to the Forge or collaborating with others. It gives users and developers of Puppet modules a common pattern, design, and style to follow. Additionally, it provides consistency in code and module structure and makes continued development, contributions, updates, and maintenance easier.

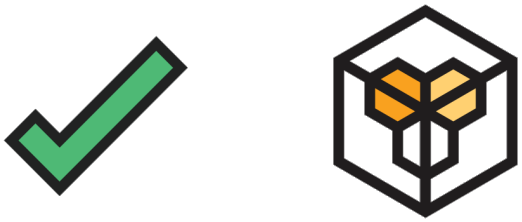
Style guides can never cover every case, so please keep in mind three principles:

- Readability matters.
- Scoping and simplicity are key.
- Treat modules as software.

Style and Syntax Checking (linting)

- Linting is the process of analyzing code for potential errors.
- Syntax checks ensure that the code is technically valid and will parse. It does not ensure it behaves as expected.
- Style checks ensure code meets generally accepted formatting and practices.

Validating Modules



- Puppet Development Kit (PDK) includes tools for syntax and style tests.
- The validations included in PDK provide a basic check of the well-formedness of the module.
- You do not need to write any tests for this validation.
- Run checks with `pdk validate`.

Puppet Development Kit (PDK) provides tools to help you run unit tests.

By default, the PDK module template includes tools that can:

- Validate the metadata.json file.
- Validate Puppet syntax.
- Validate Puppet code style.
- Validate Ruby code style.
- Run unit tests.

The validations included in PDK provide a basic check of the well-formedness of the module, and the syntax and style of the module's files. You do not need to write any tests for this validation.

By default, the pdk validate command validates the module's metadata, Puppet code syntax and style, and Ruby code syntax and style.

You can customize PDK validation with command line options. For example, you can pass options to have PDK automatically correct some common code style problems, to validate specific directories or files only, or to run only certain types of validation, such as metadata or Puppet code.

To validate against a specific version of the Puppet language, add a version flag. For example, to validate against PE 2018.1, run `pdk validate --pe-version "2018.1"`

Overview of Unit Testing

Unit tests:

- Check that components work as expected in isolation
- Run quickly
- Provide rapid feedback during development and refactoring

Testing Modules

PDK can run unit tests on a module's Puppet code to:

1. Ensure a catalog successfully compiles with the provided code.
2. Ensure the resources declared will be included in the catalog.
3. Test with multiple variations of facts such as `operatingsystem` to validate embedded logic.

When you generate a class, PDK creates a unit test file. This test file, located in your module's `/spec/classesfolder`, includes a basic template for writing your unit tests. PDK includes tools for running unit tests, but it does not write unit tests itself. However, if you are testing an empty module that you generated with PDK, you can run the unit test command to verify that all dependencies are present and that the spec directory was created correctly.

Note, you need to verify support on all platforms. `rspec-puppet` only simulates catalog compilation on different platforms by setting the "os" fact. It doesn't actually build on multiple platforms to run its test.

How to Unit Test a Module

- The `pdk test unit` command runs all the unit tests in your module.
- Ensure you have testing for your module unless it is a brand new module without any classes.

From the command line, change into the module's directory with `cd <MODULE_NAME>`

Run `pdk test unit`.

To change unit test behavior, add optional flags to the command. For example, to run only certain unit tests, run `pdk test unit --tests=<TEST1>,<TEST2>`

To unit test against a specific version of Puppet or PE, add a version option flag. For example, to test against PE 2018.1, run `pdk test unit --pe-version "2018.1"`

If there are no errors, the command returns successfully with exit code `0`, with no warnings.

System Testing

System tests

- Use test nodes to deploy a configuration, then validate the actual end-state of the node's configuration.
- Can verify idempotence by running the agent twice. No changes should be required on the second pass.
- Typically take longer to run than unit tests.

Introduction to Continuous Integration

- Continuous integration (CI) tools automatically run tests everytime code is pushed to the source code repository.
- Typically, a CI system will run syntax & style validation and unit tests for every commit. System tests may be run less frequently if they take a long time.
- Teams prioritize "fixing the build" when there are errors ensuring collaborators can see their changes pass a common set of tests and don't break existing things.
- Continuous Delivery for Puppet Enterprise (CD4PE) is one example of a continuous integration pipeline for Puppet code.

Think of developers on GitHub doing several commits a day. You then need to integrate all of the commits into production. A CI tool such as Jenkins may run a series of tests at different levels giving the team confidence in promoting code from development, through test environments, and finally, into production.

Puppet Pipelines Integration Overview

For full continuous integration (CI) and continuous deployment (CD), you will want to integrate your repository with Pipelines.

Integrate a repository

The master settings for integrating with a repository can be found in your account settings. To change these settings:

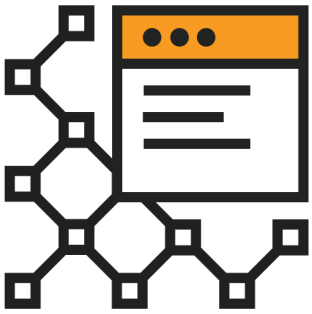
1. Click on the gear icon in the top right of the Pipelines web UI.
2. Click Integrations link on the left.
3. Click the repository you wish to integrate with.

   Note: If you are already connected to a repository, the button will say \Disconnect REPO Account. To disconnect (terminate) the integration, click the button. If you disconnect a repository that has repositories connected to applications in Pipelines for Applications, the applications will no longer auto build when a repository check-in occurs.

1. Click the "Connect REPO Account" button for the repository you wish to integrate with. You may be prompted to login to the repository and to approve access to the repository. If you have integrated Pipelines previously, you may not be prompted at all.

Once you have approved the integration on the repository, they are integrated.

Why use Puppet Pipelines for Integration?



Puppet Pipelines allow you to attach to application repositories and enable auto build and auto deployment with automated testing through the process.

Deploying applications in Puppet is a rather large and in depth topic. If you would like to learn how to do so the links below have all the steps:

- https://puppet.com/docs/pipelines-for-apps/enterprise/pfa-release-notes.html
- https://puppet.com/docs/pipelines-for-apps/enterprise/deployment-ci.html

To enable GitHub integration, follow these steps.

- In the Pipelines web UI click the gear icon at the top right.
- Click the Integrations link on the left.
- Click the GitHub icon.
- Click the Connect GitHub Account button.

Lab 9.1: Test module syntax and style

- Objectives:
  - Use the tools included by PDK to verify the syntax in your module.
  - Use the tools included by PDK to ensure your code meets style recommendations.
  - Iterate through running tests and updating the code to resolve issues found.

Checkpoint: Module Testing

Unit tests validate that the desired configuration was properly enforced on the target system.

- True
- False

Which type of tests ensure that module metadata is included?

- Validation, syntax & style
- Unit tests
- Integration tests
- System tests

How often should validation testing be run?

- Once a day
- Once a week
- Whenever you feel like it
- Before testing or committing changes

Save

Creating Role and Profile Modules

Lesson 10: Creating Role and Profile Modules

Objectives

At the end of this lesson, you will be able to:

- Combine wrapper classes and manifests into profiles, define roles from profiles, and apply roles to nodes.
- Describe how using common building blocks enables more consistent management across an entire organization while also providing granularity.
- Explain that the roles & profiles model is an implementation of wrapper classes for abstraction.
- Explain the implementation logic of a profile plus what should go into a profile and what should not.
- Use the PE console to apply roles / classes to nodes based on facts.

Good Module Design

Appropriate levels of abstraction

- Modules only manage their own resources.
  - `phpmyadmin` only manages phpMyAdmin, not Apache and MySQL.
- Classes should be designed to be reusable and composable.
  - Stack them together in multiple different combinations.
- Abstracted implementation details:
  - Configure for specific environments instead of re-writing each time.
- Classify nodes by business role.
  - Define nodes by *what they do*, not how you configure them to achieve that.

Classes that are designed to be reusable and composable means that you can take several general purpose classes and stack them together in the configuration you want. For example, you can use a module to manage a web application along with `puppetlabs/apache` and `puppetlabs/mysql` to create a complete application implementation for your site with a minimal amount of actual coding.

Rigorously keeping classes within scope also means that multiple applications may be managed on a single host without conflicts--as long as they don't attempt to manage common resources, such as Apache or MySQL, themselves!

Implementation Stack

This is called a `profile`.

- Site-specific customization of component classes.
- Define or retrieve configuration data.
- Declare application classes with parameters.
- Little to no logic and few resource declarations.

```
class profile::phpmyadmin (
  $docroot = '/var/www/phpmyadmin',
) {

  include apache
  include phpmyadmin

  phpmyadmin::server { 'default':
    port => 8080,
  }
  phpmyadmin::vhost { 'db.example.org':
    vhost_enabled => true,
    docroot       => $docroot,
    ssl           => false,
  }
}
```

Notice that values are retrieved from Hiera. The `$docroot` parameter lookup is namespaced, but because the `$ssl_cert` and `$ssl_key` parameter lookups are not namespaced with a class name, you can infer that these values might be used by multiple classes within the infrastructure.

Declaring the variables at the top of the class file makes it obvious on first glance what data is being resolved from Hiera and is recommended for clarity.

Business Role

This is called a `role`.

- Set of implementation stacks that make up a *logical* role.
- Composition of one or more profile classes.
- Defines a single complete role a node may serve.
- No logic at all.

```
class role::database_control_panel {
  include profile::base
  include profile::external_host
  include profile::phpmyadmin
}
```

Roles only declare profiles.

Profile as Customization Layer

A window into your infrastructure.

- Profile classes should be opinionated, but customizable.
  - Expose parameters for customization.
  - But only within well-defined guardrails.
- Profile classes are where all customization belongs.
  - Encapsulate all customization parameters exposed for use.
  - Encapsulate all parameters passed to component modules.

Constraining customization to a single layer makes it simpler to track down issues when they occur, often by orders of magnitude. Fewer sources of variation make for fewer configuration combinations to test.

Profile classes should limit the end user to well defined configuration choices. For example, rather than allowing full control over a group of options, the profile should allow the user to choose between pre-configured sets of values.

The profile layer is where all configuration takes place. The end user will never pass parameters to component modules. This means that reading the profile classes and looking at the parameters passed in to the profile classes should fully describe the infrastructure configuration. This makes configuration choices much more discoverable to those not intimately familiar with the codebase.

Customizing Profiles

Using profile parameters

Class: profile::wordpress

Parameter		Value	
Parameter name	=		Add parameter
wordpress_docroot	=	"/var/www/html/wordpr...	Edit Remove
wordpress_db_name	=	"knowledgebase"	Edit Remove
wordpress_db_password	=	"roygbiv"	Edit Remove
mysql_root_password	=	"hunter2"	Edit Remove

Or specify in Hiera data files:

```
---
profile::wordpress::mysql_root_password: hunter2
profile::wordpress::wordpress_db_password: roygbiv
profile::wordpress::wordpress_db_name: knowledgebase
profile::wordpress::wordpress_docroot: /var/www/html/wordpress/knowledgebase
```

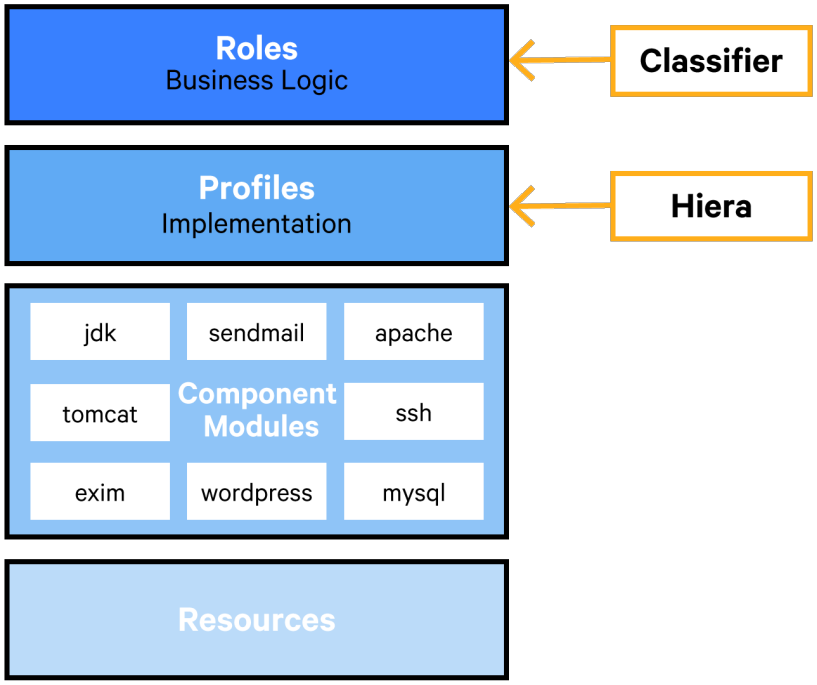
This slide shows two equivalent ways of specifying the same profile class parameters. The profile layer should be the only end user customization. We'll talk about Hiera data files in the Data Separation section. For now, consider it to be simply a data source from which variables can be retrieved.

Parameters can be set by:

- Setting profile parameters in the node classifier.
- Setting parameter keys in Hiera data files.

Roles and Profiles

Complete stack



* Components should be named after what they manage (`apache`, `ssh`, `mysql`) * Profiles should be named after the technology stack they implement (`database`, `bastion`, `mailserver`) * Roles should be named by business roles (`load_balancer`, `web_cluster`, `application`, `archive`)

Classification Using Code

Each node is assigned a single `role`.

- Nodes should only be assigned one role.
- Expose no implementation details at all.

```
node /^app\d{2,4}\.example\.com$/ {
  # matches app01.example.com, etc
  include role::application_server
}

node /^webdb\d{2,4}\.example\.com$/ {
  # matches webdb01.example.com, etc
  include role::database_control_panel
}
```

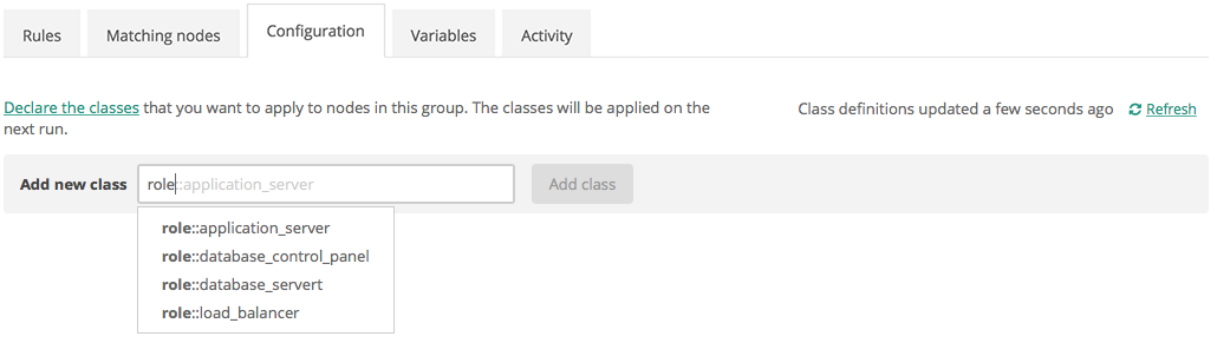
Instead of defining technology stacks at the node level, you should create roles and assign roles to nodes as required. The lack of implementation details at the node and role level means that you are free to redefine them as needed and easily refactor your complete infrastructure.

If you need to assign multiple roles to a node, that means that your role declarations are not complete. Create a new role that defines the appropriate profiles and include that role instead.

Classification Using the Node Classifier

Each node or group is assigned a single `role`

- Only use role classes in the node classifier.
- Graphically assign roles quickly and easily.



Why use Roles and Profiles?

- Without roles and profiles, people typically build system configurations in their node classifier.
- Using Hiera to handle tricky inheritance problems.

A standard approach is to create a group of similar nodes and assign classes to it, then create child groups with extra classes for nodes that have additional needs. Another common pattern is to put everything in Hiera, using a very large hierarchy that reflects every variation in the infrastructure.

Abstracting

- Ultimately, Puppet applies collections of managed resources to nodes under its control.
- To better manage the disparate resources, we create layers of abstraction:
  - Role classes declare profile classes.
  - Profile classes declare component module classes with site-specific parameters.
  - Component module classes declare Puppet resources.
- Modules are collections of related classes.

This is the key to abstraction as once you create a role, you don't 100% need to know what is in each module. The role defines the business logic but doesn't really need to know all the modules beneath the profiles. It just needs to work.

Why Add Abstraction?

Abstraction allows you to apply configuration without having to worry about the details of syntax or gritty details of how certain parts of the codebase work.

- This makes for easier refactoring.
- Do you have to use roles and profiles? No.
- Why use profiles and roles? It adds a level of abstraction.
- Why do I care about abstraction? It makes managing easier as things get more complex.

These extra layers of indirection might seem like they add complexity, but they give you a space to build practical, business-specific interfaces to the configuration you care most about. A better interface makes hierarchical data easier to use, makes system configurations easier to read, and makes refactoring easier.

Control Repo in Puppet

- To manage your Puppet code you need a control repository.
- This control repository is where code management stores code and data to deploy your environments.
- Your control repository must have a production branch.
- You'll base your other branches and environments on this production branch.
- If you are using multiple control repositories, do not duplicate branch names or you may have merge conflicts.

Code manager uses Git repository branches to create environments. Environments allow you to designate a node or node group to use a specific environment.

For example, you could designate one node group to use the development environment and another to use the production environment.

As you update the code in your control repo, code management tracks the state of that repo to keep each environment updated. Each branch of a connected repository is copied into a directory named after the branch. These environments are created in /etc/puppetlabs/code/environments on the master.

If your repository is called "mycontrolrepo", with branches named "production", "test", and "development", your production branch becomes a production directory, the test branch becomes a test directory, and the development branch becomes a development directory.

Puppetfile

Describes the modules to be installed in an environment.

```
# /etc/puppetlabs/code/environments/production/Puppetfile
forge "http://forge.puppetlabs.com"

mod "puppetlabs/razor"
mod "puppetlabs/ntp", "3.0.0"

mod "apt",
  :git => "git://github.com/puppetlabs/puppetlabs-apt.git",
  :ref => "testing_branch"

mod "stdlib",
  :git => "git://github.com/puppetlabs/puppetlabs-stdlib.git"
```

- A `Puppetfile` lives in the root of a Puppet environment
- The environment will be populated with all the modules listed
- When using a `git` source, the `:ref` argument allows you to specify a branch, commit, or tag (anything git can checkout).
- Each module should live in its own repository.

Each `mod` entry in the Puppetfile will provide a module in that environment. These typically come from the Forge or GitHub. Environments can "mix & match" by using the Puppetfile to describe only external modules and continue to maintain a large & monolithic repository containing all internal modules. However, sites who have fully embraced these dynamic ideals will have environments that consist solely of the Puppetfile.

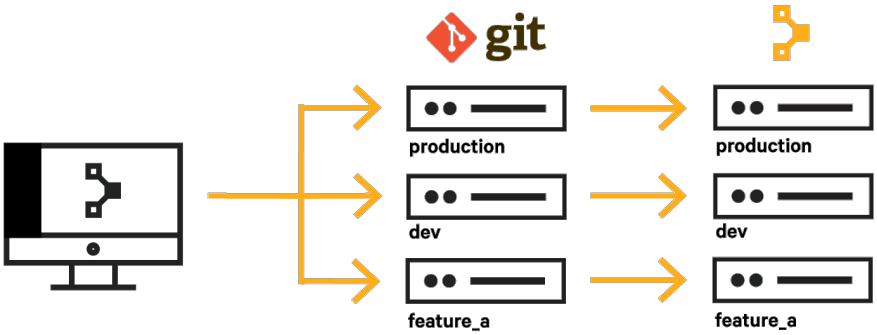
- `mod "puppetlabs/razor"`
  - Uses latest razor module from the Forge.
- `mod "puppetlabs/ntp", "3.0.0"`
  - Uses ntp version 3.0.0 from the Forgs
- The `:git` argument allow us to fetch module repositories directly from git. If we need a specific commit, release, or tag, we can specify that with the `:ref` argument.

continued...

People often ask why we choose to develop each module in a separate repository. This decouples modules from one another. One module can be developed and tagged with a release completely independently of another. This means that there are fewer chances for conflicting changes, and it helps developers stay in the modular mindset and develop more robust modules that are not tightly coupled to the internals of each other.

Several tools exist to populate environments based on the `Puppetfile`. Puppet-librarian is the originator of the Puppetfile format. We recommend the use of r10k, developed by Adrien Thebo based on his experiences with Puppet Labs Operations.

What is Code Manager?



Code Manager automates the management and deployment of your Puppet code.

Code Manager uses r10k and the file sync service to automate staging, commits, and sync your code.

To sync your code across multiple masters and to make sure that code stays consistent, Code Manager relies on file sync and two different code directories: the staging directory and the live code directory.

Create a control repository with branches for each environment that you want to create (such as production, development, or testing).

You'll also create a Puppetfile for each of your environments, specifying exactly which modules to install in each environment.

This allows Code Manager to create directory environments based on the branches you've set up.

When you push code to your control repo, you'll trigger Code Manager to pull that new code into a staging code directory (/etc/puppetlabs/code-staging).

File sync then picks up those changes, pauses Puppet Server to avoid conflicts, and then syncs the new code to the live code directories on your masters.

Lab 10.1: Create roles and profiles

- Objectives:
  - Clone the control repository.
  - Refactor the time module into a profile.
  - Add the time profile to the base profile.
  - Create a bastion role and add the base profile to it.

Checkpoint: Roles and Profiles

A profile

- Defines a technology stack
- Defines a node
- Should be applied directly to a node
- Combines several roles

The roles and profiles pattern reduces abstraction to manage nodes more effectively.

- True
- False

Code Manager creates environment groups from

- Directories in a file share
- Boxes you check in the Enterprise console
- Branches in a repository
- A list of nodes

Save

Data Separation

Lesson 11: Data Separation

Objectives

At the end of this lesson, you will be able to:

- Explain the benefits of using Hiera as a single source of truth.
- Describe a Hiera datasource hierarchy.
- Retrieve data from a Hiera datasource.

Single Source of Truth

Don't repeat yourself.

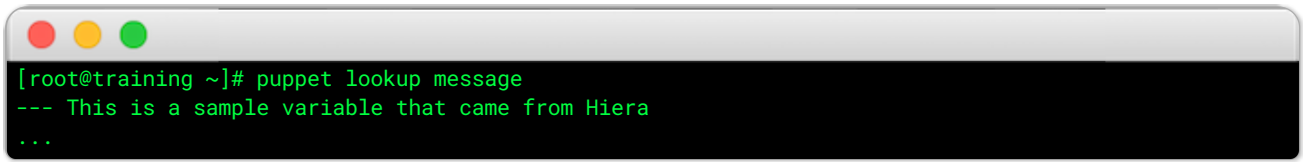
- Keep site-specific data out of your manifests.
- Puppet classes can request whatever data they need, *when they need it*.
- Benefits of retrieving configuration data from Hiera:
  - Easier to ensure that *all nodes* affected by changes in configuration data are updated in lockstep.
  - Infrastructure configurations can be managed without needing to edit Puppet code.
  - Easier to reuse or share modules.

Hiera

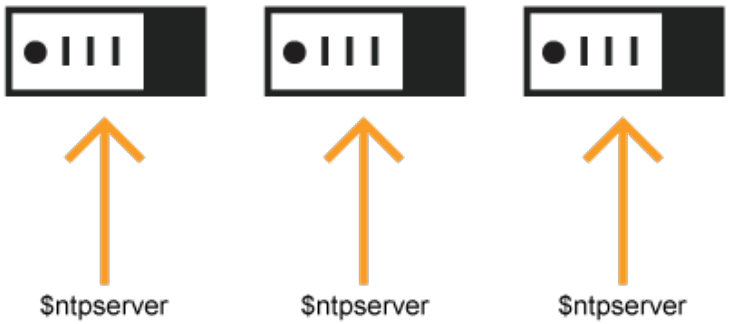
Flexible Data Lookup

- External data lookup tool.
- `key:value` data storage.
- Set values per node or for all nodes.

```
# /etc/puppetlabs/code/environments/production/hieradata/common.yaml
---
message: "This is a sample variable that came from Hiera"
```



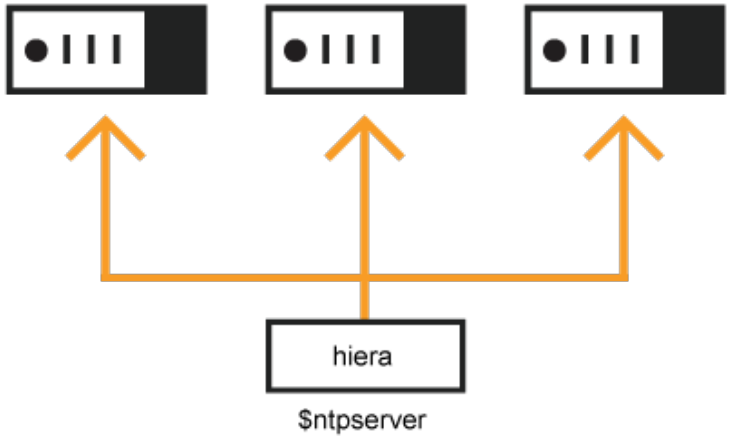
Configuration Data Without Hiera




```
class ntp {
  if ( $facts['fqdn'] == 'host4.example.com' ) {
    $ntpserver = '127.0.0.1'
  }
  elsif ( $::environment == 'test' or $facts['fqdn'] == 'test.example.com' ) {
    # Don't forget to update this to the new server on 8/17/2007
    $ntpserver = '192.168.2.1'
  } else {
    $ntpserver = 'us.pool.ntp.org'
  }

  class { 'ntp::client':
    server => $ntpserver,
  }
}
```

Consuming Hiera Data

Retrieve configuration data instead of hardcoding it.




```
class { 'ntp::client':
  server => lookup('ntpserver'),
}
```

> The `lookup()` function is part of Hiera 5 in the Standard Term Support Puppet Enterprise releases. Customers on the LTS release should use `hiera()` instead, with the same calling convention.

This provides a central location where all configuration data is kept separate from the implementation details. When updates need to be made, a single change will propagate across all the infrastructure reducing the chance of individual nodes being misconfigured. It makes the configuration specifics more clear, as well as reducing cut & paste configuration. It puts all site specific data in a single location, meaning that discoverability is greatly improved, and cuts down on required institutional knowledge. It also reduces the chances of unintended side effects, such as syntax errors breaking catalog compilations for other unrelated nodes.

The `lookup()` function has more functionality as well. For example, this invocation will validate that the data type returned is a string and will return a default value if none is found.

```
class { 'ntp::client':
  server => lookup('ntpserver', String, first, 'us.pool.ntp.org'),
}
```

Hiera Configuration

- Configured via `[environment]/hiera.yaml`
- Facts and other variables in scope are used for data resolution.

```
---
version: 5
hierarchy:
    - name: Yaml data
      datadir: data
      data_hash: yaml_data
      paths:
        - "%{facts.clientcert}.yaml"
        - "%{facts.datacenter}.yaml"
        - "common.yaml"
```

This hierarchy is resolved in order, based on:

1. `$facts['clientcert']`
2. `$facts['datacenter']`
3. `common` to return common values.

This `hiera.yaml` configures Hiera 5 in the Standard Term Support Puppet Enterprise releases. Customers running the LTS instead, will want to use the Hiera 3 config format. The concepts are similar, but Hiera 5 allows for more for more expressive hierarchies. Hiera 3 supports only a global `hiera.yaml`. See https://docs.puppet.com/puppet/latest/hiera_intro.html for more information.

With this configuration and an environment directory of `/etc/puppetlabs/code/environments/production`, the Hiera data files will be queried in this order:

1. `/etc/puppetlabs/code/environments/production/hieradata/%{clientcert}.yaml`
2. `/etc/puppetlabs/code/environments/production/hieradata/%{datacenter}.yaml`
3. `/etc/puppetlabs/code/environments/production/hieradata/common.yaml`

We have configured Hiera to look in `[environmentpath]/hieradata` for data files in `.yaml` format. Hiera will replace variables in the `:hierarchy` paths to construct filenames.

For example, if we used this configuration to retrieve the value of `ntpserver` for a node named `node1.example.com` in the `houston` data center, Hiera would look for the key `ntpserver` in the files below, in the order listed, and would return the first value found.

1. `/etc/puppetlabs/code/environments/production/hieradata/node1.example.com.yaml`
2. `/etc/puppetlabs/code/environments/production/hieradata/houston.yaml`
3. `/etc/puppetlabs/code/environments/production/hieradata/common.yaml`

continued...

You can see the hierarchy expanded by using `puppet lookup`.

```
root@training:/home/training # puppet lookup message --explain
[...]
Searching for "message"
  Global Data Provider (hiera configuration version 5)
    Using configuration "/etc/puppetlabs/puppet/hiera.yaml"
    Hierarchy entry "Yaml data"
      Path "/etc/puppetlabs/puppet/hieradata/training.puppetlabs.vm.yaml"
        Original path: "%{::trusted.certname}.yaml"
        Path not found
      Path "/etc/puppetlabs/puppet/hieradata/production.yaml"
        Original path: "%{environment}.yaml"
        Path not found
      Path "/etc/puppetlabs/puppet/hieradata/common.yaml"
        Original path: "common.yaml"
        Found key: "message" value: "This is a sample variable that came from a Hiera datasource"
```

Available Hiera functions:

> The functions below come with Hiera 5 in the Standard Term Support Puppet Enterprise releases. The LTS versions of the functions are listed for each.

`lookup($key)`

- Call out to hiera to look up a key using the configured datasource hierarchy.
- Returns the first value found.
- LTS version: `hiera($key)`

`lookup($key, { 'merge' => 'unique' })`

- Traverses the entire hierarchy and constructs an array of *all* values found.
- Elements can be any type.
- LTS version: `hiera_array($key)`

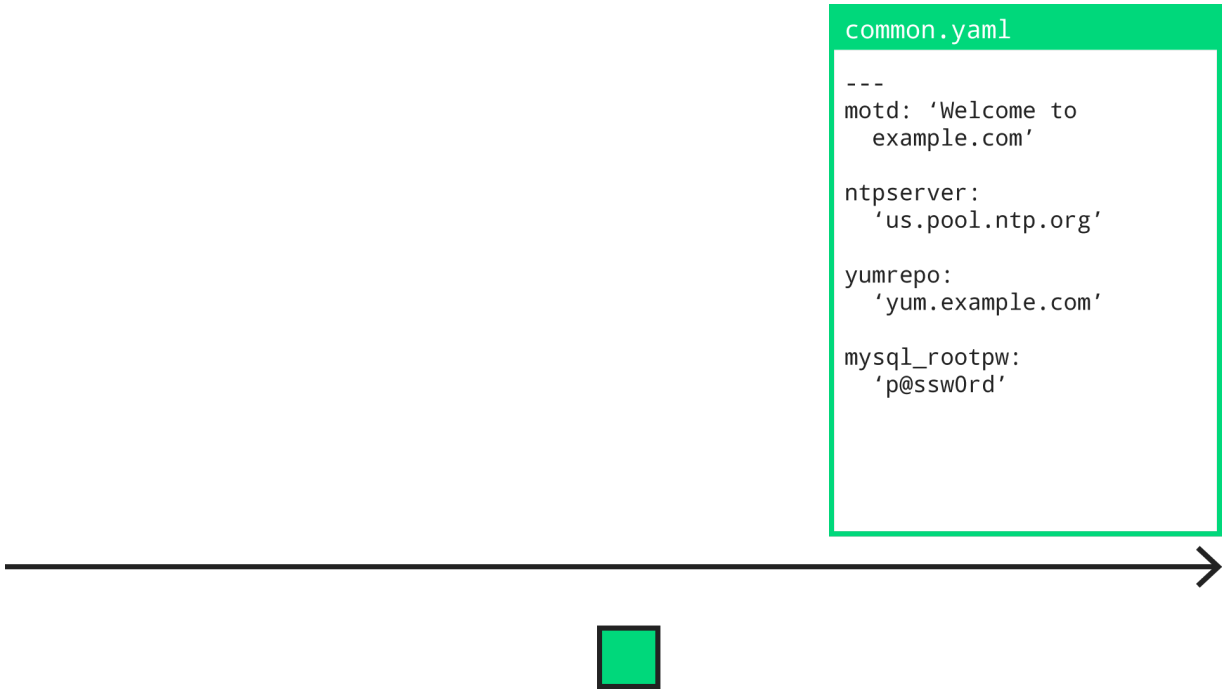
`lookup($key, { 'merge' => 'hash' })`

- Traverses the entire hierarchy and merges *all* values found into a single hash.
- All values found must be hashes.
- LTS version: `hiera_hash($key)`

`include(lookup($key, { 'merge' => 'unique' }))`

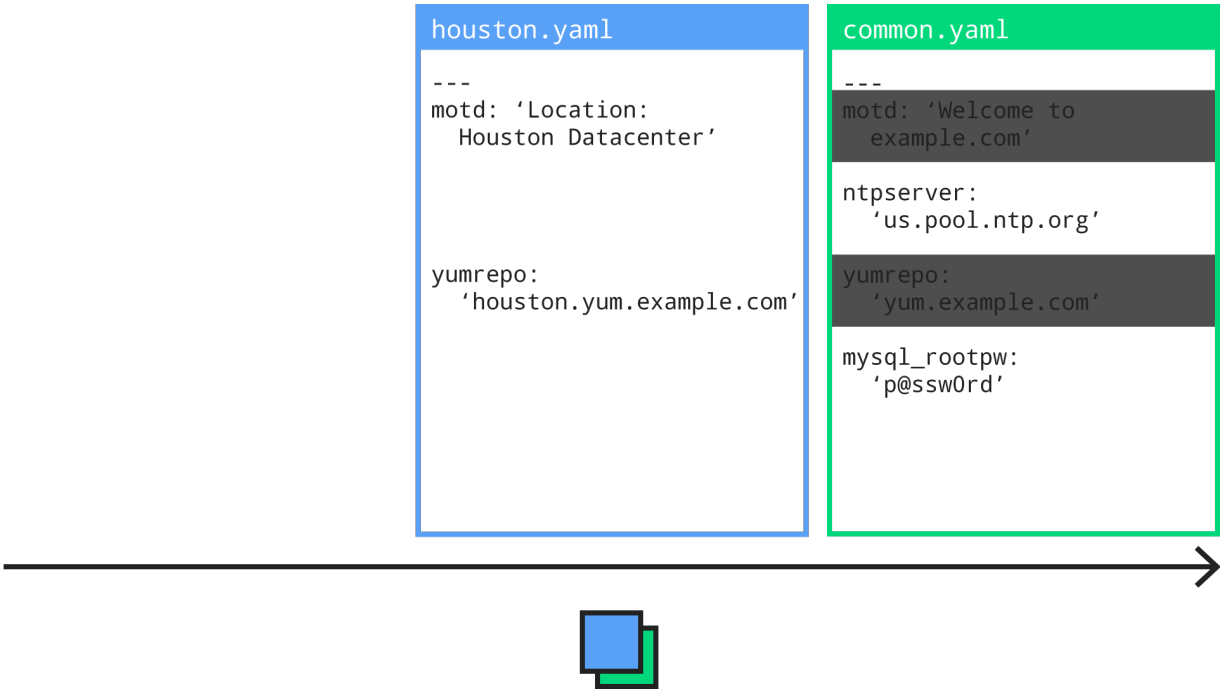
- Call `hiera_array()` on `$key` and include all values returned.
- `$key` can represent node, group, role, etc. and should resolve to a list of classes to include.
- LTS version: `hiera_include($key)`

Hiera Visualization



The next three slides visualize the Hiera query order. Here you can see the bottom level of the `:hierarchy`; common values that will applied if nothing else overrides them.

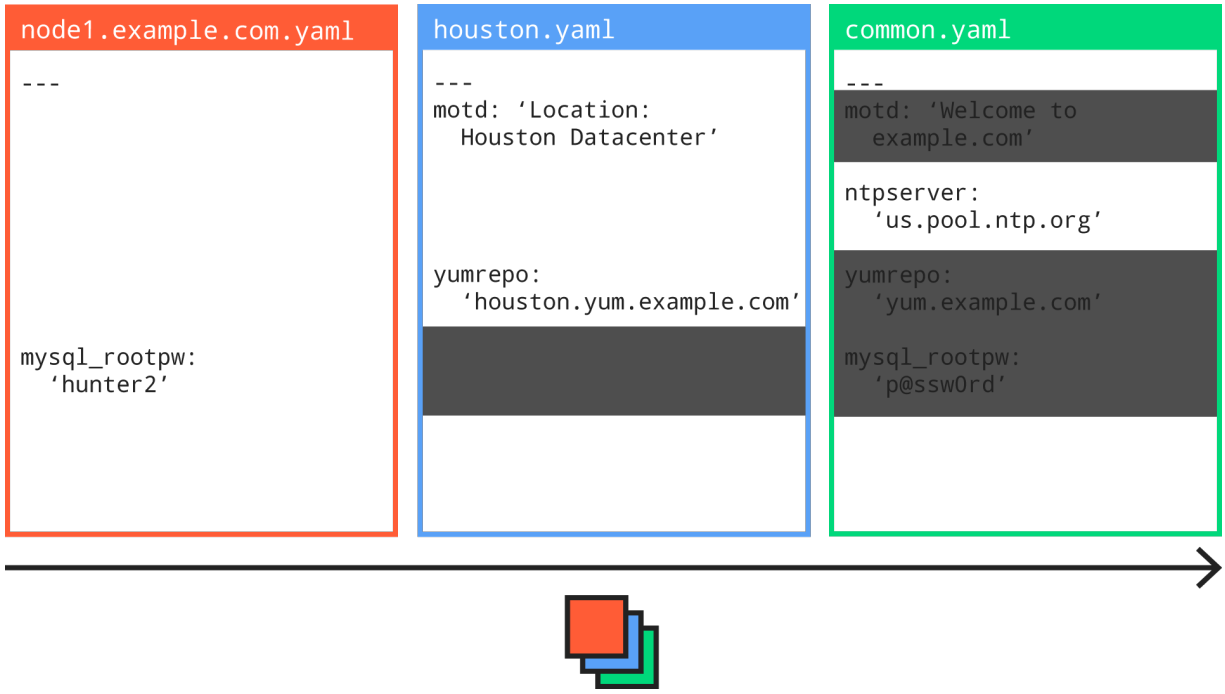
Hiera Visualization



On the second sheet are our datacenter overrides. This is the second level of the `:hierarchy` setting. You see that some variables are overridden, but that some of the common variables (`$ntpserver` and `$mysql_rootpw`) show through.

When `$facts['datacenter']` is set to 'houston' and we request a variable, these values are returned.

Hiera Visualization



Finally, our top level of the `:hierarchy` is the node's `$certname`. You see that only one variable is overridden at this level, and that only one variable shows through from the common layer.

When `$facts['datacenter']` is set to 'houston' and `$certname` is set to `node1.example.com` and we request a variable, these values are returned.

The final result is a composition of all the layers in the `:hierarchy`. This composition is constructed each time a variable is requested, so it will be different for each node.

Handling Sensitive Data

- Hiera allows multiple backends, such as `yaml` and `eyaml` in the same configuration.
- The `hiera-eyaml` backend supports encrypted data items.
- Encrypted and unencrypted data items may be stored in the same Hiera data file.
- The `binford2k/node_encrypt` module wipes sensitive data in catalogs and reports.

Sample Hiera data file with encrypted value

```
---
plain-property: You can see me

encrypted-property: >
    ENC[PKCS7,Y22exl+OvjDe+drmik2XEeD3VQtl1uZJXFFF2NnrMXDWx0csyqLB/2NOWefv
    [...]
    IZGeunzwhqfmEtGiqpvJJQ5wVRdzJVpTnANBA5qxeA==]
```

Sample use of the `node_encrypt::file` resource to scrub data from the catalog

```
node_encrypt::file { '/tmp/foo':
  owner   => 'root',
  group   => 'root',
  content => 'This string will never appear in the catalog.',
}
```

`hiera-eyaml` is the standard solution for encrypting sensitive data when it's at rest in a Hiera data file. Using the `binford2k/node_encrypt` module ensures that sensitive data does not appear in unencrypted files in the Puppet infrastructure, such as the Puppet catalog or agent report files.

The `puppet/hiera` module is also useful for configuring `hiera.yaml` with eYAML support.

More information may be found at:

- https://github.com/TomPoulton/hiera-eyaml
- https://forge.puppet.com/binford2k/node_encrypt
- https://forge.puppet.com/puppet/hiera

Checkpoint: Data Separation

Using Hiera requires the installation of a supported database.

- ○ True
- ○ False

The single source of truth design pattern allows you to:

- ☐ Point your finger at the dev who checked in bad code.
- ☐ Ensure that updates propagate everywhere they're needed.
- ☐ Cut and paste with confidence.
- ☐ Ensure that each application that uses a common setting has the same value for that setting.

Hiera returns the value of string based keys.

- ○ True
- ○ False

Hiera allows you to override data values based on fact variables.

- ○ True
- ○ False

Save

Create a Baseline with Puppet

Lesson 12: Create a Baseline with Puppet

Objectives

At the end of this lesson, you will be able to:

- Employ wrapper classes, profiles, and roles to compose a common baseline which can be applied to all or most of the nodes in your environment.
- Discuss the common resources managed in baselines.

Day Zero Configuration

Now that we have the ability to customize Forge modules to meet our unique datacenter requirements, we can begin to establish minimum configurations that are common to all nodes/devices/systems.

- Setup remote access via ssh or remote desktop
- Configure the default execution policy
- Add a login banner
- Enable network time synchronization
- Set the read-only SNMP community string
- Configure logging to a centralized destination
- Push unique digital certificates

Use Roles and Profiles to Create Your Unique Baseline

- Resource modeling is handled by the component modules.
- Profiles define the technology stack.
- Roles define the business logic.

Using this logic, you can create a baseline profile for all nodes within a group.

Combine with other technology-specific profiles to create roles but the core things only need to be defined once.

"Role" it all up

Roles:
business logic

Profiles:
implementation

Components:
modeling resources

This whole stack combined can create a baseline that can be sent across most nodes.

Introduction to Variables

Variables allow you to define values once and use them multiple places such as in logic statements or when constructing long strings.

Variables are prefixed with '`$`':

```
$httpd_dir = '/etc/httpd/conf.d'
```

Variables can be used as resource titles:

```
file { $httpd_dir:
  ensure => directory,
}
```

Variables can be used as attribute values:

```
$readme_content = "Please add project information to this file.\n"

file { "${httpd_dir}/README":
  ensure  => file,
  content => $readme_content,
}
```

Variables must be defined before they can be used. Because the Puppet DSL allows you to reference undefined variables, this is a common source of errors. If the `$readme_content` variable were to be defined after the file resource was declared, the file would be created with no content.

Variables Example

Using variables judiciously will reduce repetition in your code.

```
class apache {
  $httpd_dir = '/etc/httpd/conf.d'

  file { $httpd_dir:
    ensure => directory,
  }

  file { "${httpd_dir}/www1.conf":
    ensure  => file,
    content => "Configuring the ${httpd_dir}/www1.conf",
  }
}
```

Using a variable like this means that you can make updates in a single place and they are propagated throughout your codebase. A common practice is to put these variable assignments into a `params` class, such as `mymodule::params`, and then include that class and refer to the fully scoped name anywhere it's needed. Scope will be covered in the next few slides.

Variables are Immutable

Variables CANNOT be reassigned!

```
class apache {
  $httpd_dir = '/etc/httpd/conf.d'

  file { $httpd_dir:
    ensure => directory,
  }

  # Compilation will fail at the reassignment of $httpd_dir
  $httpd_dir = '/etc/site/httpd/conf.dir'

  file { "${httpd_dir}/httpd.conf":
    ensure => file,
  }
}
```

Variables cannot be reassigned, but local variables of the same name can be set to override global variables, including facts.

Immutable
Unchanging over time or unable to be changed.

Constructing Strings

Single-quoted strings are literal strings:

```
$string = 'My httpd_dir is ${httpd_dir}\n'
```

```
> My httpd_dir is ${httpd_dir}\n
```

Double-quoted strings allow variable interpolation:

```
$string = "My httpd_dir is ${httpd_dir}\n"
```

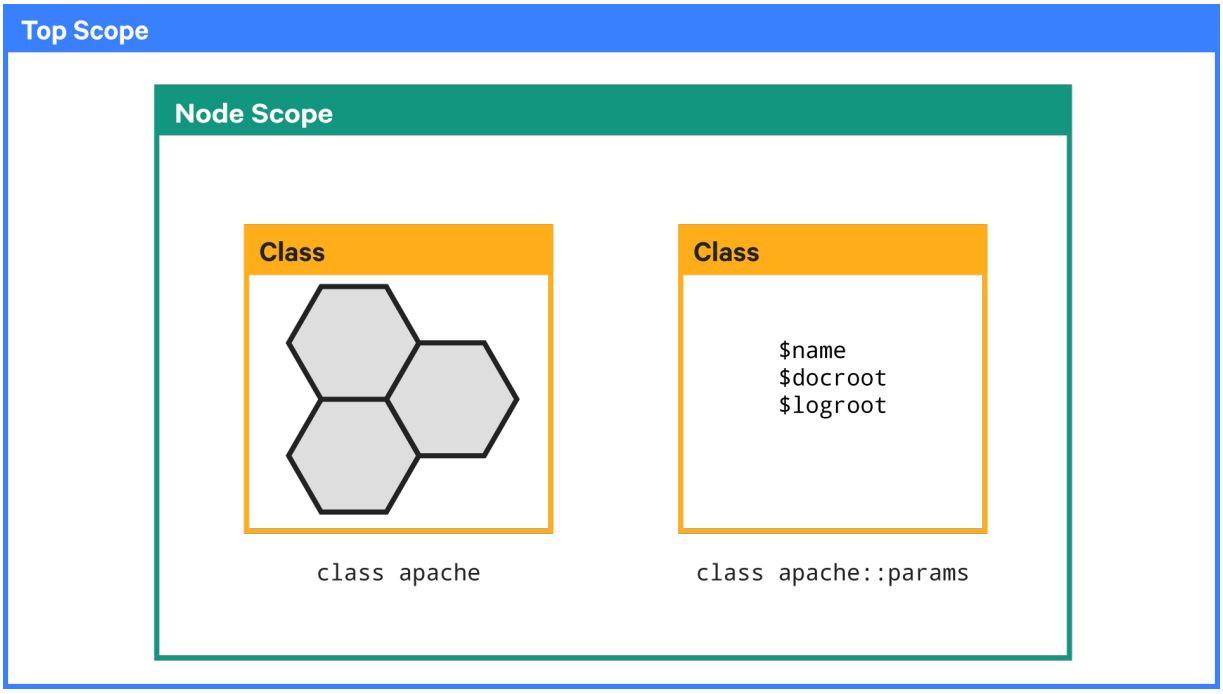
```
> My httpd_dir is /etc/httpd/conf.d
```

Variables in strings should be bracketed with `{}` for clarity.

Variables should be enclosed in curly braces when they are being interpolated, such as when they are part of a string inside double quotation marks. Curly braces should not be used outside of strings.

Scope

Partial isolation of areas of code.



Scope limits the reach of variables. Any given scope has access to its own contents, and also receives additional contents from the node and from top scope.

- Top scope is usually defined by `site.pp`, outside of any node definitions.
- Node scope is within the definition of the current node.
- Class scope is within the definition of the class.

Details on scope can be found in Puppet Documentation at https://puppet.com/docs/puppet/latest/lang_scope.html.

Lab 12.1: Expand initial roles and profiles

- Objectives:
  - Create security_baseline.pp profile.
  - Add windows_firewall and ssh wrapper profiles.
  - Include security_baseline.pp into the base profile.
  - Create roles and profiles in Puppet Enterprise.
  - Classify nodes and deploy the bastion role.

Checkpoint: Creating a Baseline

A baseline is an implementation of the roles and profiles model.

- True
- False

A baseline should be applied to which nodes?

- Each node individualy
- Problem nodes that need special remediation
- All nodes in a given location or environment
- Nodes that belong to the baseline group

A baseline implements custom business logic for all applications in your environment.

- True
- False

Save

Creating and Accepting Parameters in the Baseline

Lesson 13: Creating and Accepting Parameters in the Baseline

Objectives

At the end of this lesson, you will be able to:

- Create classes which accept parameters to customize behavior.
- Understand how to use Facter and Hiera as input for those parameters.
- Describe the different variable types and syntax within the Puppet DSL.
- Specify defaults for class parameters via values or functions.
- Supply parameters to a class from a manifest and the PE console.
- Explain Hiera use for maintaining data and for class parameters.
- Use `puppet lookup` to analyze data returned from Hiera in a given context.

Parameterized Classes

Customize behavior for different configurations.

```
class ssh (
  $server      = true,  # Enable the server
  $client      = true,  # Enable the client
  $allow_root  = true,  # permit root to log in
  $untrusted   = false, # permit untrusted hosts to log in
  $x11_forward = false, # forward X11 protocol; run remote graphical apps
) {
  include ssh::hostkeys  # set up keys for trusted hosts

  if $server {
    include ssh::server  # manage server
    file { '/etc/ssh/sshd_config':
      ensure  => file,
      content => epp('ssh/sshd_config.epp'),
    }
  }
  if $client {
    include ssh::client  # manage client
    file { '/etc/ssh/ssh_config':
      ensure  => file,
      content => epp('ssh/ssh_config.epp'),
    }
  }
}
```

Variable Type Boolean

Booleans are `true` or `false`.

If a non-boolean value is used where a boolean is required:

- The `undef` value is converted to boolean `false`.
- All other values are converted to boolean `true`.

Notably, this means the string values `""` (zero-length string) and `"false"` both resolve to true.

If you want to convert other values to booleans with more permissive rules (0 as false, "false" as false, etc.), the puppetlabs-stdlib module includes `str2bool` and `num2bool` functions.

Some modules will "munge" similar values to true/false values. Common examples are '1' & '0', 'yes' & 'no', and 'on' & 'off'. Check the documentation of the resource type you are using to see if it supports things other than true & false.

Variable Type Strings

Strings are unstructured text fragments of any length. They're probably the most common and useful data type.

- Bare word
- Single quoted
- Double quoted
- Heredocs

There are four ways to write literal strings in the Puppet language:

- Bare words
- Single-quoted strings
- Double-quoted strings
- Heredocs

Bare word example: `service { "ntp": ensure => running, # bare word string }`

Single-quote strings example: `if $autoupdate { notice('autoupdate parameter has been deprecated and replaced with package_ensure. Set this to latest for the same behavior as autoupdate => true.') }`

Double-quote string example: `"This is an example"`

Double-quoted strings can be interpolated, meaning that veriables and special characters will be expanded and interpreted. To prevent interpolation, use single-quotes.

When strings are surrounded by double quotes, "like this", line breaks within the string are interpreted as literal line breaks. You can also insert line breaks with `\n` (Unix-style) or `\r\n` (Windows-style).

Heredocs are more complex and won't be discussed. See the puppet documentation for more.

Variable Types Enums

Enum types are used when you want to limit input options for strings.

- Provide a list of valid options

`Enum['stopped', 'running']` — matches the strings `'stopped'` and `'running'`, and no other values.

Variable Types - Arrays

Arrays are a list.

```
$my_array = [1, 2, 3]
$my_animals = ['bird', 'cat', 'dog']
```

When assigning multiple variables from an array, there must be an equal number of variables and values. Nested arrays can also be used.

An array on the right hand side can be assigned to a single variable on the left hand side.

```
[$a, $b, $c] = [1,2,3]      # $a = 1, $b = 2, $c = 3
[$a, [$b, $c]] = [1,[2,3]]  # $a = 1, $b = 2, $c = 3
[$a, $b] = [1, [2]]         # $a = 1, $b = [2]
[$a, [$b]] = [1, [2]]       # $a = 1, $b = 2
```

If the number of variables and values do not match, the operation will fail.

Variable Types - Hashes

A hash is a group of key-value pairs.

```
$my_hash = { a => 10, b => 20 }

# or in a multi-line style
$my_hash = {
  a => 10,
  b => 20,
}
```

You can refer to individual hash items, if needed.

```
$one_element = $my_hash['a']
# sets $one_element to 10
```

Syntax for Datatypes

Strongly typed languages reduce confusion when someone supplies one type of data when another was expected.

- Specifying the data type makes it clear what is expected.
- Errors can be generated earlier, before Puppet tries to use the value.
- Data types are written as unquoted upper-case words, like String.
- Data types sometimes take parameters, which make them more specific.
  (For example, `String[8]` is the data type of a string with a minimum of eight characters.)

Each known data type defines how many parameters it accepts, what values those parameters take, and the order in which they must be given. Some of the abstract types require parameters, and most types have some optional parameters available.

The general form of a data type is: An upper-case word matching one of the known data types.

Sometimes, a set of parameters, which consists of:

- An opening square bracket (`[`) after the type's name. (There can't be any space between the name and the bracket.)
- A comma-separated list of values or expressions — arbitrary whitespace is allowed, but you can't have a trailing comma after the final value.
- A closing square bracket (`]`).

There is also a general resource data type, which all data types are more-specific subtypes of.

Like the Mytype-style data types, it matches no values that can be produced in the Puppet language. You can use parameters to restrict which values resource will match, but it will still match no values.

This is mostly useful if: - You need to interact with a resource type before you know its name. For example, you can do some clever business with the iteration functions to re-implement the create_resources function in the Puppet language, where your lambda will receive arguments telling it to create resources of some resource type at runtime.

- Someone has somehow created a resource type whose name is invalid in the Puppet language, possibly by conflicting with a reserved word — you can use a Resource value to refer to that resource type in resource declarations and resource default statements, and to create resource references.

Parameterized Classes

Specify parameter data types.

```
class ssh (
  Boolean $server      = true,  # Enable the server
  Boolean $client      = true,  # Enable the client
  Boolean $allow_root  = true,  # permit root to log in
  Boolean $untrusted   = false, # permit untrusted hosts to log in
  Boolean $x11_forward = false, # forward X11 protocol; run remote graphical apps
) {
  # [...]
}
```

- Describes the data type accepted for each parameter.
- Compilation will fail if the wrong type of data is passed in.

Declaring classes with data types is *optional*, but is recommended for Puppet 4+ codebases due to the built in sanity checking.

Parameterized classes can include default values. If every parameter has a default, then you can use the `include` function, like we have been up until this point.

Notice the use of the `Boolean` type in the class parameter list. This is an optional Puppet 4 language feature to enforce type checking on class parameters. If the class is declared with a non-boolean parameter value, the Puppet catalog compiler will exit with an error message. This is an easy way to ensure that data is validated as the catalog is compiled.

See https://docs.puppet.com/puppet/latest/reference/lang_data.html for more information on data types.

Parameterized Classes

Supply parameters in wrapper classes and profiles

```
class profile::ssh::workstation {
  class { 'ssh':
    x11_forward => true,
    server      => false,
  }
}
```

```
class profile::ssh::bastion {
  class { 'ssh':
    allow_root => false,
    untrusted  => true,
  }
}
```

```
node 'jumphost.example.com' {
  include profile::ssh::bastion
  # [...]
}

node 'web01.example.com' {
  include ssh # accept all default parameters to the ssh class
  # [...]
}
```

Node definitions become simply a list of classes to include. These support classes have been called `aspects`, `behaviours`, `roles`, etc. The key is that they separate the implementation of a configuration description from the assignment of that configuration. In other words, you can describe a node as a list of roles it should serve, rather than being forced to provide all the details for each bit of configuration each time you configure a node.

The `profile` module name is a common design pattern that we covered in an earlier lesson. Following this pattern helps to make more flexible and composable code that makes it easier to build trustworthy code and refactor with confidence.

Using a Data Model for Parameters

- Data are some of the most critical components of configuration management
- How you customize for your environment
- Worth the effort to create good data models

Single Source of Truth

Don't repeat yourself.

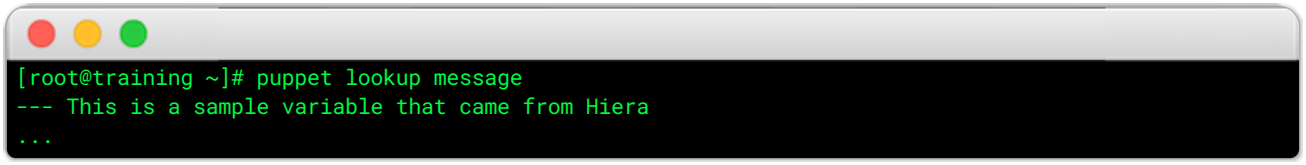
- Keep site-specific data out of your manifests.
- Puppet classes can request whatever data they need, *when they need it*.
- Benefits of retrieving configuration data from Hiera:
  - Easier to ensure that *all nodes* affected by changes in configuration data are updated in lockstep.
  - Infrastructure configurations can be managed without needing to edit Puppet code.
  - Easier to reuse or share modules.

Hiera

Flexible Data Lookup

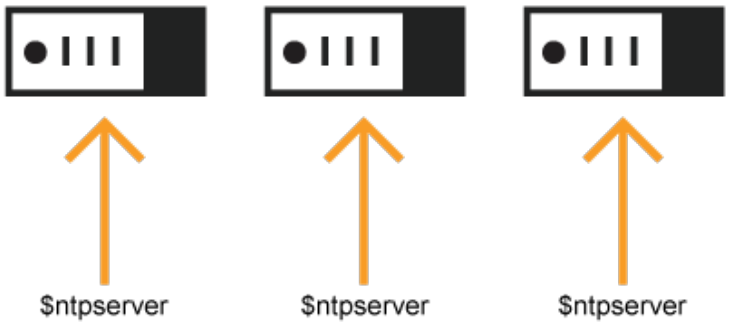
- External data lookup tool.
- Structured data storage.
- Configurable hierarchy for looking up data.
- Most specific match is returned.

```
# /etc/puppetlabs/code/environments/production/hieradata/common.yaml
---
message: "This is a sample variable that came from Hiera"
```



Note that Hiera is rarely configured on the agent, considering that functions are executed on the master. Configuring it locally allows you to experiment with Hiera usage during this class if you wish. Your local Hiera configuration will be available when running `puppet apply`, but not when requesting a catalog from the classroom master with `puppet agent -t`. This will use the Hiera data files existing on the master.

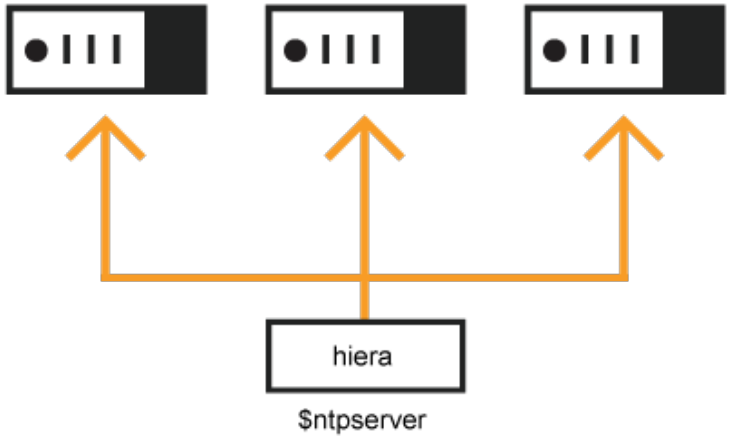
Configuration Data Without Hiera




```
class ntp {
  if ( $facts['fqdn'] == 'host4.example.com' ) {
    $ntpserver = '127.0.0.1'
  }
  elsif ( $::environment == 'test' or $facts['fqdn'] == 'test.example.com' ) {
    # Don't forget to update this to the new server on 8/17/2007
    $ntpserver = '192.168.2.1'
  } else {
    $ntpserver = 'us.pool.ntp.org'
  }

  class { 'ntp::client':
    server => $ntpserver,
  }
}
```

Consuming Hiera Data

Retrieve configuration data instead of hardcoding it.




```
class { 'ntp::client':
  server => lookup('ntpserver'),
}
```

The `lookup()` function is part of Hiera 5.

The `hiera()` function and its relatives were deprecated in Puppet 4.9 and will be removed in Puppet 6.

This provides a central location where all configuration data is kept separate from the implementation details. When updates need to be made, a single change will propagate across all the infrastructure reducing the chance of individual nodes being misconfigured. It makes the configuration specifics more clear, as well as reducing cut & paste configuration. It puts all site specific data in a single location, meaning that discoverability is greatly improved, and cuts down on required institutional knowledge. It also reduces the chances of unintended side effects, such as syntax errors breaking catalog compilations for other unrelated nodes.

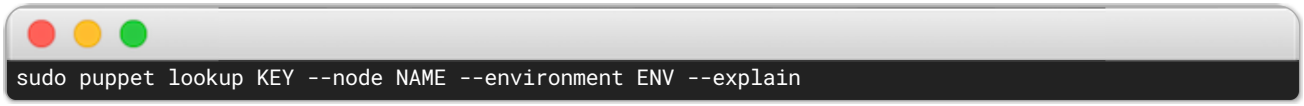
The `lookup()` function has more functionality than the `hiera()` function, as well. For example, this invocation will validate that the data type returned is a string and will return a default value if none is found.

```
class { 'ntp::client':
  server => lookup('ntpserver', String, first, 'us.pool.ntp.org'),
}
```

Puppet `lookup`

The `puppet lookup` command is a CLI for Puppet's '`lookup()`' function.

It returns a value for the requested lookup key, so you can test and explore your data.



Since this command needs access to your Hiera data, make sure to run it on a node that has a copy of that data. This usually means logging into a Puppet Server node and running `puppet lookup` with `sudo`.

The most common version of this command is:

When looking up a key, Hiera searches up to four different hierarchy layers of data, in the following order:

1. Global hierarchy.
2. The current environment's hierarchy.
3. The indicated module's hierarchy, if the key is of the form `<MODULE NAME>::<SOMETHING>`.
4. If not found and the module's hierarchy has a `default_hierarchy` entry in its `hiera.yaml` — the lookup is repeated if steps 1-3 did not produce a value.

   Note: Hiera checks the global layer before the environment layer. If no global `hiera.yaml` file has been configured, Hiera defaults are used. If you do not want it to use the defaults, you can create an empty `hiera.yaml` file in `/etc/puppetlabs/puppet/hiera.yaml`.

Lab 13.1: Class params

- Objectives:
  - Add parameters to the existing time profile.
  - Configure time servers via class parameters in the PE console.
  - Configure time servers via configuration data in the PE console.

Checkpoint: Creating and Accepting Parameters

Which is not a valid Puppet DSL data type?

- Enum
- String
- Long
- Boolean

Class parameters must be supplied from the Puppet Enterprise Console.

- True
- False

Hiera uses PuppetDB to store the values it looks up for use by classes.

- True
- False

Save

Define an Application Stack with Puppet

Lesson 14: Define an Application Stack with Puppet

Objectives

At the end of this lesson, you will be able to:

- Describe the package, file, service model.
- Describe how to manage the order of operations to deploy the necessary components of an application
- Use resource relationships to restart a service when its dependencies change.
- Describe the default ordering of resources and the implicit relationships between resources.
- Specify explicit relationships between resources.
- Describe the use of Pipelines to support a seamless workflow from building an application, through testing, to deployment via Puppet.

Dependency Management

How does Puppet prioritize the enforcement of resources?

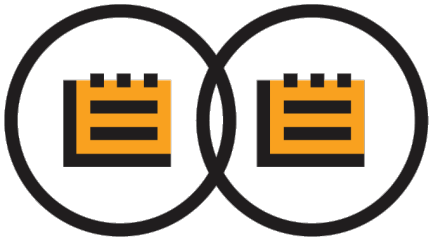
- Puppet does not necessarily enforce resources in simple source order.
- Instead, Puppet evaluates dependencies between resources.
- Enforcement is ordered to meet all relationship requirements.
    - Relationships may exist between any resources in the catalog.

> Manifests are parsed in source order when compiling, so you cannot use a variable before you define it, for example. The resource enforcement order, however, is driven by the dependency graph.

Note that Puppet Enterprise recently enabled the option of source based ordering. This does not replace understanding the dependency system, and will be covered toward the end of the section.

Relationships

Defined with metaparameters



- Explicitly define ordering relationships.
- Metaparameters work with all resource types.
- There are four metaparameters representing two different kinds of relationships between resources.

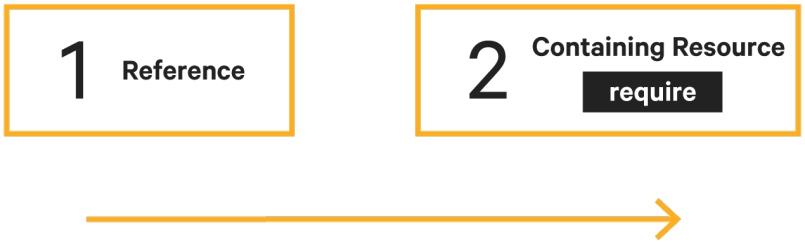
Four metaparameters that establish relationships between resources will be covered further:

- `require`
- `before`
- `subscribe`
- `notify`

Best practices are to *always* define the dependency relationships you need, and to *never* define the relationships that you don't.

`require`

`require` a referenced resource to be applied first.



A real world relationship:

> Early morning trainings *require* that I have coffee to stay awake.

Example: `require`

Ensure that `sshd` is started after `openssh` is installed.

```
package { 'openssh':
  ensure => present,
}

service { 'sshd':
  ensure  => running,
  enable  => true,
  require => Package['openssh'],
}
```

Referencing a Resource In a Relationship

Declaring a resource:

```
typename { 'title':
  attribute => value,
}
```

Referencing that resource:

```
service { 'myservice':
  subscribe => Typename['title'], # reference to a resource with a given type and title
}
```

This example uses an arbitrary name to demonstrate the capitalization pattern. A lowercase letter indicates a resource declaration and an uppercase letter with square brackets indicates a reference to that declaration, no matter what the type is.

- Puppet resources always get specified in pairs: type and title.
- When we make reference, we need both parts.
- When referencing existing resources from your catalog, make sure:
  - The first character of the type is capitalized.
  - The resource title goes into the brackets.
- The part in brackets *"indexes"* to the *title* of a resource.

See https://puppet.com/docs/puppet/latest/lang_data_resource_reference.html for more.

`before`

Request to be applied `before` a referenced resource.

Another real world relationship:

> I need a coffee *before* early morning trainings.

Even though this keyword sounds more like ordering, it's still only indicating a dependency relationship. To complete the coffee analogy, it doesn't actually matter which order coffee fits into the morning routine.

- You could get coffee at the hotel and then walk to the facility.
- You could walk from the hotel to a local coffee shop, and then to the facility.
- You could walk from the hotel to the training facility and then have coffee.

All of those are acceptable resolutions to having coffee before starting class.

Example: `before`

Also ensure that `sshd` is started after `openssh` is installed.

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

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

Notice that the `require` metaparameter has been moved from the `service` resource to a `before` metaparameter on the `package` resource.

This has exactly the same effect as the previous `require` statement did. `require` and `before` simply define either end of that same relationship. There's no functional difference between them; you can simply choose which one fits your current needs better.

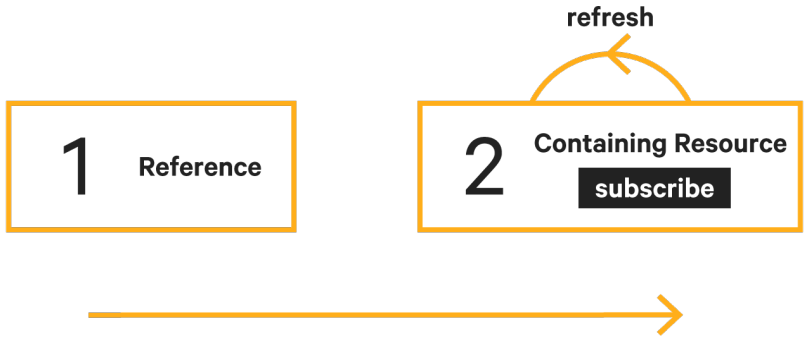
Refresh Events

Resource changes can refresh other resources.

- `subscribe` and `notify` metaparameters establish refresh relationships.
- Some resources that respond to refresh events are:
  - `service`
    - Restarts/reloads the service even if it's already running.
  - `mount`
    - Unmounts and remounts the volume.
  - `exec`
    - Runs the associated command again.
    - Can be set to `refreshonly => true`, which means the command only runs when it receives a refresh event.
  - `reboot`
    - Reboot a computer when required after updates.

Builtin types that explicitly respond to refresh events include `service`, `exec`, and `mount`. Third party modules from the Forge may respond to refresh events. The `reboot` type (from the puppetlabs-reboot Forge module) uses the refresh event to schedule a system reboot during or after a Puppet run.

`subscribe`



React when Puppet makes changes to another resource.

When *that resource* changes, let *this resource* know about it.

Refreshing Services

Restart `sshd` if Puppet changes `/etc/ssh/sshd_config`.

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  source => 'puppet:///modules/ssh/sshd_config',
}

service { 'sshd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ssh/sshd_config'],
}
```

- The `subscribe` metaparameter implies `require`.
- Enforces order as well as watching for changes.
- Only sends refresh events when Puppet makes changes.

`notify`

Inform another resource when Puppet makes changes.

When *this resource* changes, let *that resource* know about it.

Refreshing Services

Also restarts `sshd` if Puppet changes `/etc/ssh/sshd_config`.

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

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

- The metaparameter `notify` implies `before`.
- Enforces order as well as sending change notifications.
- Only sends refresh events when Puppet makes changes.

Package | File | Service

Effectively managing a service

- We commonly specify several resources together to model a complete configuration.
- A reasonable workflow when installing a service is to:
    1. Install a package.
    2. Configure one or more config files.
    3. Enable the service.
- To model this in Puppet, we use the *Package* | *File* | *Service* design pattern.

This is one of the most useful and common Puppet resource design patterns. Knowing it well will get you far in your understanding of Puppet.

Example:

```
package { 'ntp':
  ensure => latest,
}

file { '/etc/ntp.conf':
  mode    => "644",
  content => epp("ntp/client-ntp.conf.epp"),
}

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

First

Install a Package


```
package { 'ntp':
  ensure => present,
}
```

Second

Configure a File


```
package { 'ntp':
  ensure => present,
}

file { '/etc/ntp.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/ntp/ntp.conf',
  require => Package['ntp'],
}
```

Why does `/etc/ntp.conf` need to be configured after the package is installed?

If the config file was configured before the package was installed, it's possible that the package installation would overwrite it. To avoid that, we ensure that package installation happens first, then we overwrite any sample or default configuration with the expected configuration we'd like.

Third

Enable a Service

```
package { 'ntp':
  ensure => present,
}

file { '/etc/ntp.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/ntp/ntp.conf',
  require => Package['ntp'],
}

service { 'ntpd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ntp.conf'],
}
```

- The `ntpd` service resource is subscribing to the `/etc/ntp.conf` file resource.
- It will restart when Puppet modifies the config file.
- Why not also subscribe to the package resource?

In many cases, the package manager and/or package will schedule its associated services to stop prior to upgrade and restart afterwards. In that case, if the service is subscribed to the package resource, it would restart again.

If the service is not automatically restarted, then it would be useful to subscribe to the package resource.

Workflow recap:

1. Install the package.
2. Write out any configuration file(s).
3. Enable the service.
   - Restart service when Puppet updates config file(s).

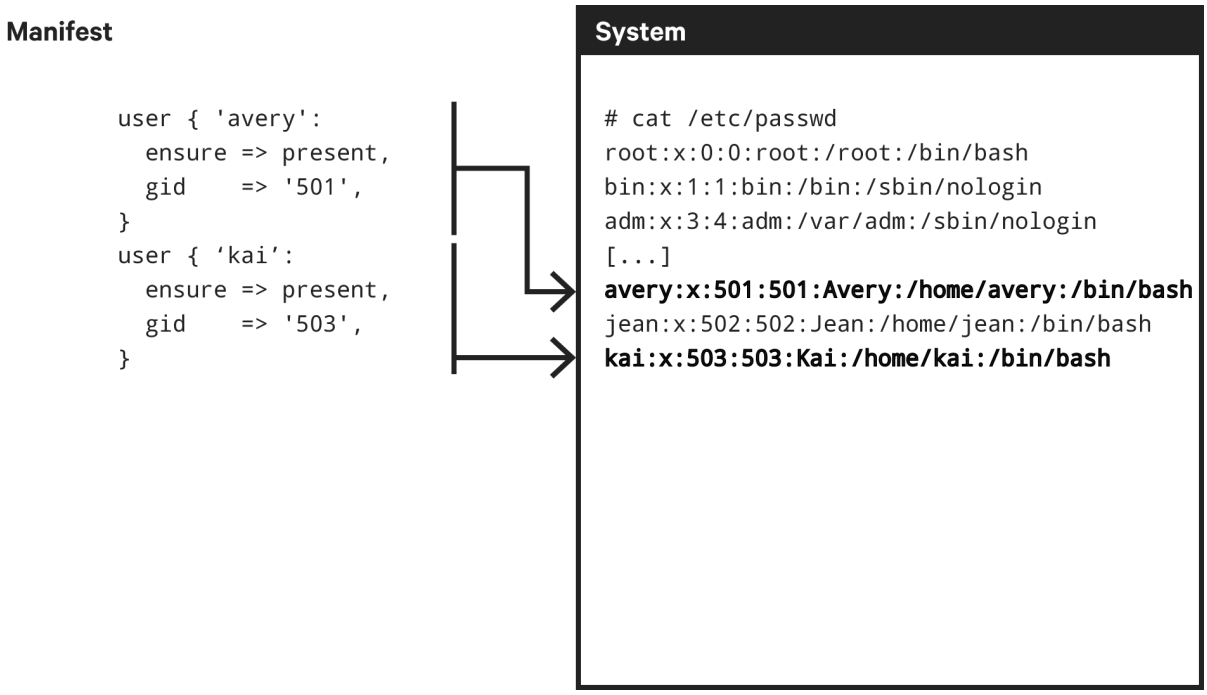
Package - File - Service

Common design pattern.


```
file { '/etc/ntp.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/ntp/ntp.conf',
  require => Package['ntp'],
}
package { 'ntp':
  ensure => present,
}
service { 'ntpd':
  ensure    => running,
  enable    => true,
  subscribe => File['/etc/ntp.conf'],
}
```

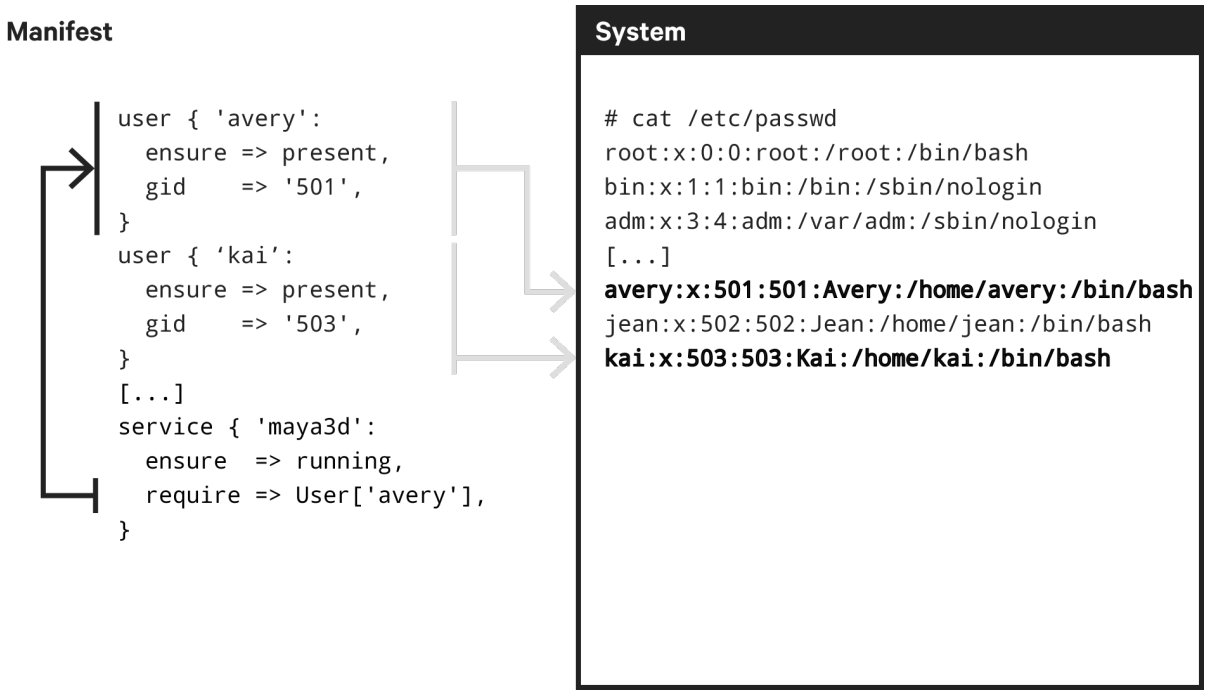
Reference Syntax Roundup

A resource in a manifest corresponds to a resource on the node it's applied to.



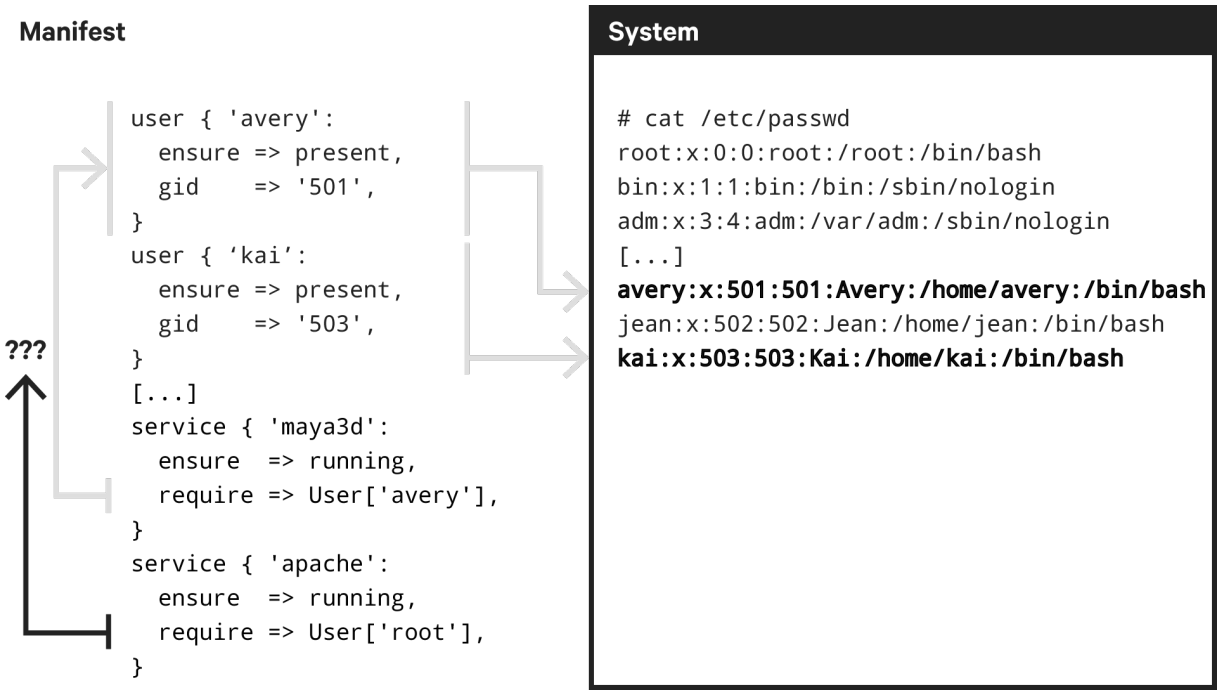
Reference Syntax Roundup

A reference in a manifest points to another resource in the catalog...



Reference Syntax Roundup

...which may or may not completely represent the state of the node.



Syntax Roundup

Using Resource Types

Declaring a resource:

```
typename { 'title':
  attribute => value,
}
```

Referencing that resource:

```
service { 'myservice':
  subscribe => Typename['title'], # reference to a resource with a given type and title
}
```

> This example uses an arbitrary name to reinforce the capitalization pattern. A lowercase letter indicates a resource declaration and an uppercase letter indicates a reference to that declaration, no matter what the type is.

https://docs.puppet.com/puppet/latest/lang_data_resource_reference.html

Syntax Roundup

Using Classes

Defining a class:

```
class ssh {
  package { 'openssh-server':              # declare a resource
    ensure => present,
  }
  file { '/etc/ssh/sshd_config':
    ensure  => file,
    require => Package['openssh-server'],  # express a dependency on a resource
  }
}
```

Declaring that class:

```
# ensure the ssh class is in the catalog somewhere
include ssh

# include the ssh class, but also enforce it before the current class
require ssh
```

Dependency Shortcuts

Specifying each and every dependency can be tedious.

- Implicit dependencies:
  - Certain resources always depend on one another.
  - Automatic soft dependencies for these related resources.
- Manifest ordering:
  - Defaults to source ordering in limited cases.
  - Only applies **within a single manifest file**.
  - Superceded by any other dependencies (anywhere in the codebase).
  - Provides no indication when it is overridden.
  - Does not trigger refresh events.

If no explicit dependencies are provided and no implicit dependencies exist, Puppet will default to enforcing resources from any given manifest in the order they appear in that manifest.

- All soft relationships are **superseded** by explicit relationship declarations.
- Manifest ordering has been an option since Puppet 3.3.0 and on by default since Puppet Enterprise 3.3 and Puppet 4.0.

Manifest ordering applies *only* to resources that aren't explicitly ordered and cannot be expected to work determinately across multiple files. It will also relationships anywhere else in the codebase, meaning that real errors can be masked and go unnoticed until their failure is critical.

Relying on this hidden ordering is a good way to create modules that break in unexpected fashions on machines that don't have it enabled or that include different classes or include them in a different order.

Best practices are to explicitly define all required resources, even when using manifest ordering.

Refresh events are only fired when we indicate interest by using the `subscribe` or `notify` metaparameters. This means that manifest ordering will never fire refresh events.

Users and Groups

Explicitly assigned dependency:

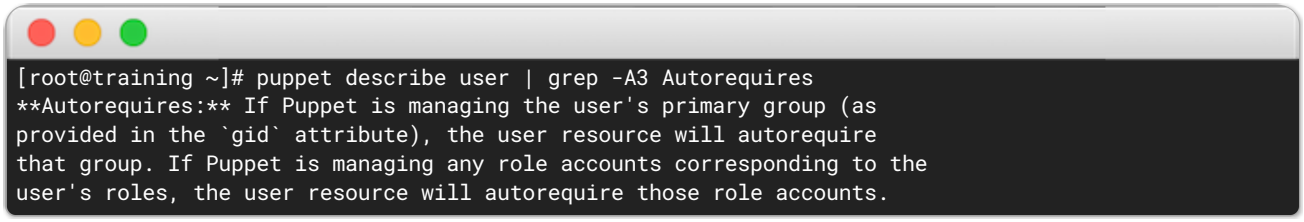
```
user { 'foo':
  ensure     => present,
  home       => '/home/foo',
  managehome => true,
  uid        => '5000',
  gid        => 'sysadmin',
  shell      => '/bin/bash',
  require    => Group['sysadmin'], # redundant!
}

group { 'sysadmin':
   ensure => present,
   gid    => '5000',
}
```

Users and Groups

Puppet implicitly orders users and groups:

```
user { 'foo':
  ensure     => present,
  home       => '/home/foo',
  managehome => true,
  uid        => '5000',
  gid        => 'sysadmin',
  shell      => '/bin/bash',
}

group { 'sysadmin':
   ensure => present,
   gid    => '5000',
}
```

Puppet documentation:



Files and Directories

The directory must exist before the file can be created:

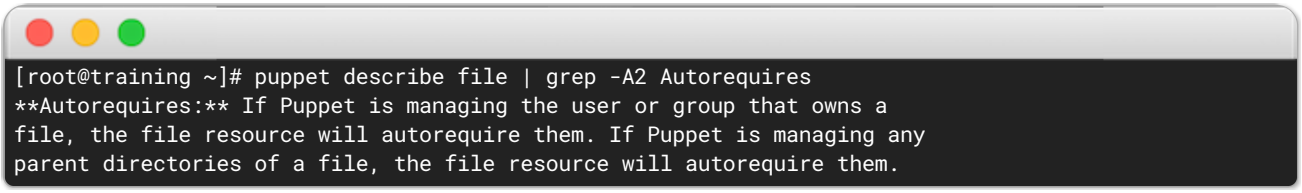
```
file { '/etc/httpd/conf.d':
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}

file { '/etc/httpd/conf.d/www.puppet.com.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => "# This file configures the puppet.com website\n",
  require => File['/etc/httpd/conf.d'],    # not required
}
```

Files and Directories

Puppet implicitly recognizes file hierarchy:

```
file { '/etc/httpd/conf.d':
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}

file { '/etc/httpd/conf.d/www.puppet.com.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => "# This file configures the puppet.com website\n",
}
```

Puppet documentation:



File Ownership

Implicit relationships between file ownership and user resources:

```
user { 'foo':
  ensure     => present,
  home       => '/home/foo',
  managehome => true,
  uid        => '5000',
  gid        => 'sysadmin',  # implicitly requires Group['sysadmin']
  shell      => '/bin/bash',
}

group { 'sysadmin':
   ensure => present,
   gid    => '5000',
}

file { '/etc/foo.conf':
  ensure  => file,
  owner   => 'foo',       # implicitly requires User['foo']
  group   => 'sysadmin',  # implicitly requires Group['sysadmin']
  mode    => '0644',
  content => 'debug = true;',
}
```

Tasks in Modules

- Some applications require their included installers to be used. Tasks are ideal for that use-case.
- By including install tasks with application modules, everything is included in one module distribution.
- Adding a task to a Puppet module is easy and can help simplify functionality by including one-time operations and long-term management in a single package.
- Add a task to your module with: `pdk new task <TASK_NAME>`

From the command line in your module's directory run `pdk new task <TASK_NAME>`. PDK creates a task file, with the naming convention `task_name.sh` and a task metadata file, `task_name.json` in the `./tasks` directory. Although the task template is in shell script, you can write tasks in any language the target nodes can run.

Puppet Pipelines

Pipelines allow a seamless workflow from building an application, through testing, to deployment via Puppet.

Using what we've learned throughout this course, you can automate an entire application stack with Puppet Pipelines.

Lab 14.1: Application

Define and Deploy an Application Stack

Objectives:

- Create the required role and profile structure for deploying a Java app.
- The role/profile will:
  - Set up required directory structure.
  - Install Java.
  - Copy application JAR into place.
  - Configure the JAR to be run as a system service.
  - Work for both Windows and Linux.

Course Summary

Over three days, you were able to:

- Identify use cases for adopting Puppet Enterprise.
- Differentiate between imperative and declarative techniques for configuration management.
- Locate, download, wrap, and deploy Puppet modules from the Puppet Forge.
- Gain visibility into the current state of your systems.
- Use Bolt & Puppet tasks to run scripts or initial automation tasks on multiple nodes.
- Model declarative state for ongoing platform management.
- Automate application delivery when the platform is stable.

Instructor-led Training: Practitioner

What's covered?

- Advanced Puppet language constructs
- Using custom facts to expose information about nodes
- Module design concepts and best practices
- Understanding the difference between Puppet tasks and Puppet modules
- Troubleshooting techniques and standard log files
- Contributing modules to the community via the Puppet Forge

If you were able to engage with the concepts in Getting Started with Puppet and successfully complete the labs, you may be ready for Puppet Practitioner. Learn how to design, build, and extend modules from the Forge and develop best practices for implementing roles and profiles. This course is offered both in-person and virtually.

In this three-day course, you will design, build, and extend modules from the Forge and learn best practices for implementing roles & profiles in your infrastructure. After completing this course, you will show a mastery of the Puppet language and problem solving techniques based on Puppet best practices.

Prerequisites

In this course, we expect attendees to have a solid understanding of configuration management strategies. Students should have either taken Getting Started with Puppet or have 6+ months of experience using Puppet.

If you feel like you need more hands-on practice before you move to module development and more advanced topics, start with the Puppet Learning VM. Work at your own pace on a series of quests that guide you through the core components of Puppet.

Resources & Next Steps

Community Involvement:

- Puppet Slack channel - http://slack.puppet.com
- Puppet Forge - https://forge.puppet.com/
- IRC channel - #puppet on Freenode

Self-Paced Learning:

- Download the Learning VM - http://puppet.com/download-learning-vm
- Puppet Training - https://learn.puppet.com
- Get Puppet Certified - http://puppet.com/certification
- Get Questions Answered - https://slack.puppet.com/

Working With Puppet:

- Download Puppet Enterprise - manage 10 nodes for free
  - http://puppet.com/download-puppet-enterprise
- Puppet Docs - https://puppet.com/docs

Resources & Next Steps

Community Involvement:

- Puppet Slack channel - http://slack.puppet.com
- Puppet Forge - https://forge.puppet.com/
- IRC channel - #puppet on Freenode

Self-Paced Learning:

- Download the Learning VM - http://puppet.com/download-learning-vm
- Puppet Training - https://learn.puppet.com
- Get Puppet Certified - http://puppet.com/certification
- Get Questions Answered - http://ask.puppet.com

Working With Puppet:

- Download Puppet Enterprise - manage 10 nodes for free
  - http://puppet.com/download-puppet-enterprise
- Puppet Docs - https://puppet.com/docs

Need more technical detail or product drill down? Schedule a follow-up call with a Puppet Professional Services Engineer.

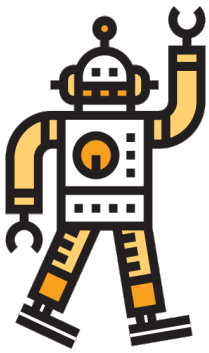
As always, don't forget to look for a community module on the Forge before attacking the problem yourself.

Class Feedback

At this time, we would appreciate any feedback you can share. Please complete our course survey.

Thank You!

Congratulations, you have finished!



Thanks for taking Getting Started with Puppet.

Thank You!

Appendix

This section contain additional resources, links, and documents to help you understand the course content. Send suggestions for additional resources to education@puppet.com.

Additional Information and Resources

Git

Puppet professional services engineers have compiled some resources they like for helping new folks get started with git.

- https://git-scm.com/videos - Videos on git basics
- https://try.github.io/ - Handbooks, tutorials, visuals, and classes for learning git
- http://www.wei-wang.com/ExplainGitWithD3 - Interactive tutorial
- https://learngitbranching.js.org/ - Interactive tutorial
- http://rogerdudler.github.io/git-guide/ - Tutorial
- http://think-like-a-git.net/ - "A GUIDE FOR THE PERPLEXED"
- https://github.com/k88hudson/git-flight-rules - A guide about what to do when things go wrong
- https://www.youtube.com/user/GitHubGuides - GitHub Training & Guides (for GitHub but most concepts are generic)
- https://www.atlassian.com/git/tutorials/ - Atlassian Git tutorials (for BitBucket but most concepts are generic)

Cheatsheets

- https://devpro.github.io/puppet/cheatsheet.html - Puppet cheatsheet
- https://gist.github.com/JamesMGreene/cdd0ac49f90c987e45ac - git cheatsheet

Self-paced courses

- Puppet Language Basics: get started writing Puppet code.
- Intro to Bolt and Puppet Tasks: two short modules on using orchestration.