Automation Tools

7.4.1

Introduction to Automation Tools

In this topic, you will learn about three of the most popular automation tools: Ansible, Puppet, and Chef.

You will also have the option to install one or all of them on your local workstation. If you want to try this, ensure that you have access to a Linux-based workstation, such as Ubuntu or macOS.

You should always refer to the tool's own installation documentation for your operating system.

7.4.2

What Do Automation Tools Do for Us?

What do automation tools do for us?

Automation tools like Ansible, Puppet, or Chef offer powerful capabilities compared to ad-hoc automation strategies using BASH, Python, or other programming languages.

Simplify and standardize

Automation tools "wrap" operating system utilities and API functions to simplify and standardize access. Often, they also establish intelligent defaults that speed code drafting and testing. They make tool-centric code less verbose and easier to understand than scripts.

You can still access deeper underlying functionality with built-in shell access that enables you to issue "raw" shell commands, inject shell and other scripts into remote systems to enable delicate configuration. You can reuse legacy configuration code, and add functionality to the tool itself by composing modules and plugins in languages like Python or Ruby.

Automation tool modules enable best practices that make code safer and idempotency easier to achieve. For example, many Ansible functions can back up configuration files on target systems or retrieve copies and store them locally on a deployer machine before making changes. This helps to enable recovery if a deployment breaks, or is interrupted.

Accelerate development with out-of-the-box features

Automation tools typically provide some very powerful functionality to accelerate development. For example, by default, Ansible 2.4+ provides functionality that lets you easily retrieve configuration snapshots from Cisco ACI network fabrics. It also has complementary functionality to help you enable rollback of ACI configurations to a prior snapshotted state.

Facilitate reusability, segregate concerns, promote security

Modern automation tools strive to be "data-driven" and enable the following:

- Compilation of variable definitions
- Server inventory as structured data and other details separate from generic code
- Orderly means to inject variable values into code, config file templates, and other destinations at runtime

For example, Ansible Vault supports sophisticated functionality for encrypting sensitive files and variables, securely editing encrypted file contents, and more.

Perform discovery and manage inventory

Automation tools typically gather information from target devices as a default in the normal course of operations. This information includes hardware configuration, BIOS settings, operating system, configuration of network and other peripheral cards and subsystems, installed applications, and other details.

Some tools, like Cisco ACI, can also gather configuration details from individual devices and higher-order virtualization frameworks. Others feature dynamic inventory systems that enable automated extraction, compilation, and realtime updating of data structures, describing all resources configured in a private or public cloud estate.

Handle scale

Most automation tools can work in a local mode, as well as a client/server or distributed agent mode. This lets the tool manage thousands or tens of thousands of nodes.

Engage community

Most popular tools are available in open source core versions, helping the community to accelerate development, and find and fix bugs. Users of these tools also share deployment playbooks, manifests, recipes, and more. These are designed for use with the tool, are may be distributed via GitHub and other public repositories, and on tool-provider-maintained repositories like Ansible Galaxy.

7.4.3

Critical Concepts

Idempotency: a review

Idempotent software produces the same desirable result each time that it is run. In deployment software, idempotency enables convergence and composability. Idempotent deployment components let you:

- More easily gather components in collections that build new kinds of infrastructure and perform new operations tasks
- Execute whole build/deploy collections to safely repair small problems with infrastructure, perform incremental upgrades, modify configuration, or manage scaling.

For example, suppose an operator misconfiguration causes a problem in a complex Kubernetes cluster. Idempotent lets you revert the most recent change in the deployment code and rebuild the cluster from scratch in five minutes, completely confident that you will have a working product.

This is the basic thinking behind infrastructure as code. Idempotency guarantees that your codebase achieves your desired goal by converging on a sequence of dependent goals. In this way, running your codebase against a base infrastructure target will have the idempotent result of producing a new or revised working infrastructure.

Procedure vs. declarative

Procedural code can achieve idempotency, but many infrastructure management, deployment, and orchestration tools have adopted another method, which is creating a declarative. A declarative is static model that represents the desired end product. This model is used by middleware that incorporates deployment-specific details, examines present circumstances, and brings real infrastructure into alignment with the model, via the least disruptive, and usually least time-consuming path.

These days, most popular automation tools are characterized as inherently procedural or declarative. Ansible and Puppet, for example, are often described as employing declarative Domain-Specific Languages (DSLs), whereas Chef is said to be more inherently procedural.

This is a somewhat artificial distinction, because all these platforms (as well as BASH, Python, etc.) are procedural at the lowest level; Ansible is based on Python, Puppet and Chef are built on Ruby. All can make use of both declarative and procedural techniques as needed, and many real-world automation tasks require both approaches.

```
- name: Install Apache webserver
  apt:
    name: apache2
    state: present
    update-cache: yes
```

Figure 1. Typical terse Ansible declaration, which installs the Apache web server on an Ubuntu host

In this example, `state: present` could be replaced with `state: absent` to remove the package if found. The `update-cache: yes` setting performs the equivalent of `apt-get update` before attempting the installation.

Provisioning vs. configuration vs. deployment vs. orchestration

Operations people tend to think in terms of a hierarchy of infrastructure layers and associated task-domains:

- **Provisioning** refers to obtaining compute, storage, and network infrastructure (real or virtual), enabling communications, putting it into service, and making it ready for use by operators and developers (e.g., by installing an operating system, machine-level metrics, ssh keys, and the lowest level of operations tooling).
- **Configuration** means installing base applications and services, and performing the operations, tasks, and tests required to prepare a low-level platform to deploy applications or a higher-level platform.
- **Deployment** involves building, arranging, integrating, and preparing multi-component applications (such as database clusters) or higher-level platforms (like Kubernetes clusters), often across multiple nodes.
- **Orchestration** may refer to several things. When meant concretely, it usually refers to user-built or platform-inherent automation aimed at managing workload lifecycles and reacting dynamically to changing conditions (e.g., by autoscaling or self-healing), particularly in container environments. When meant abstractly, it may refer to processes or workflows that link automation tasks to deliver business benefits, like self-service.

People who come to operations from software development tend to have a looser perspective on how these terms should be used. They tend to use to term deployment about anything that is not orchestration. They make the strongest distinction between "things you need to do to make a system ready for testing/use" and "adjustments the system needs to make automatically, or that you may be asked to make for it."

People also use the phrase "configuration management" when describing IT automation tools in general. This can mean one of two things:

- An expanding idea of "configuration" as installing applications and services.
- A distinction between configuration and coding: domains where static descriptions can neatly represent work-products vs. domains in which processes must be made explicit and possible side-effects anticipated. For more information, research the connection between declarative automation and functional/logical programming, and the complementary connection between procedural automation and imperative programming.

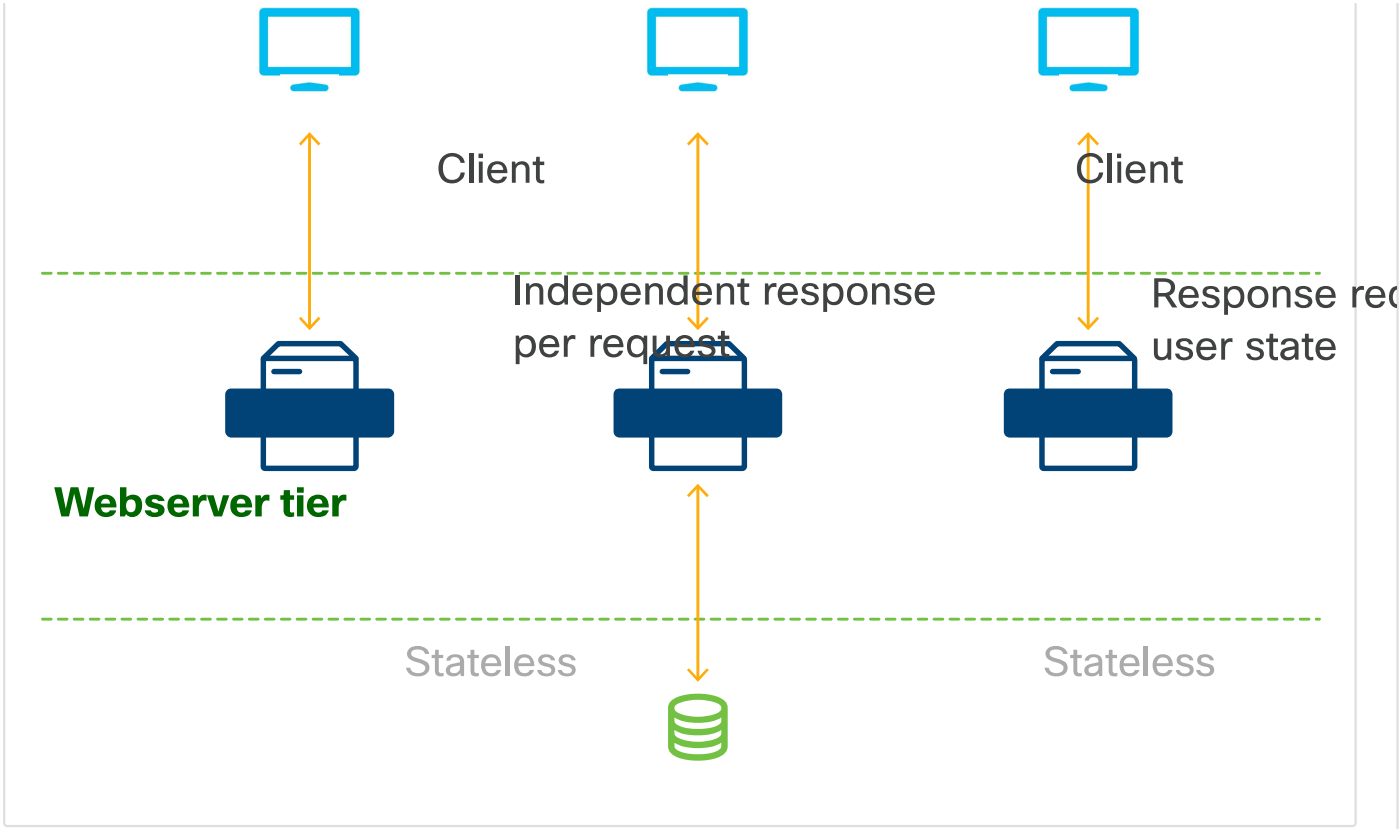
Statelessness

Automation works best when applications can be made stateless. This means that redeploying them in place does not destroy, or lose track of, data that users or operators need.

- **Not stateless** - An app that saves important information in files, or in a database on the local file system.
- **Stateless** - An app that persists its state to a separate database, or that provides service that requires no memory of state between invocations.

The discussion of full-stack (infrastructure + applications) automation in this topic assumes that the applications being discussed are stateless and/or that you, the developer, have figured out how to persist state in your application so that your automation can work non-destructively.

Examples of Statelessness and Statefulness



This diagram depicts three kinds of applications:

- **Stateless / No state to store -** This app requires only atomic/synchronous interactions between client and server. Each request from client to server returns a result wholly independent of prior and subsequent requests. An example of this application is a public web server that returns an HTML page, image, or other data on request from a browser. The application can be scaled by duplicating servers and data behind a simple load balancer.
- **Stateless / State stored in database -** User state is stored in a database accessible to any webserver in the middle tier. An example of this application is a web server that needs to be aware of the correspondence between a user ID and user cookie. New webservers and copies of the website can be added freely without disrupting user sessions in progress and without requiring that each request from a given user be routed to the specific server that maintains their session.
- **Stateful / State stored on server -** A record of user state must be maintained across a series of transactions. An example of this application is a website that requires authentication. The app is not allowed to serve pages to a user who is not logged in. User state is typically persisted by giving the client an identifying cookie that is returned to the server with each new request and used to match an ID stored there. This application cannot be scaled just by adding servers. If a logged-in user is routed to a server that has not stored an ID matching the user's cookie, that server will not recognize them as being logged in, and will refuse their request.

Apps that need to maintain state are inconvenient candidates for full-stack automation, because state will be destroyed by an ad hoc rebuild of their supporting infrastructure. They also cannot be efficiently migrated away from one pool of resources (for example, one set of application servers or hosts) to another.

7.4.4

Popular Automation Tools

The first modern automation tool was probably Puppet, introduced in 2005 as open source, and then commercialized as Puppet Enterprise by Puppet Labs in 2011.

Currently, the most popular tools are Ansible, Puppet, and Chef. They share the following characteristics:

- Relatively easy to learn
- Available in open source versions
- Plugins and adapters enable them to directly or indirectly control many types of resources: software-defined bare-metal infrastructure (Cisco UCS, Cisco devices), private cloud (VMware, OpenStack), and public cloud (Amazon Web Services, Microsoft Azure, Google Cloud Platform).

Many other solutions also exist. Private and public cloud providers often endorse their own tools for use on their platforms, for example, OpenStack's HEAT project, and AWS' CloudFormation. Other solutions, many aimed at the fast-growing market for container orchestration, pure infrastructure-as-code, and continuous delivery of infrastructure+applications, include SaltStack and Terraform.

7.4.5

Ansible

Ansible

Ansible is probably the most broadly popular of current automation solutions. It is available as open source, and in a version with added features from IBM/Red Hat that is called Ansible Tower. Its name comes from the novels of speculative fiction author Ursula K. LeGuin, in which an "ansible" is a future technology enabling instant communication at cosmic distances.

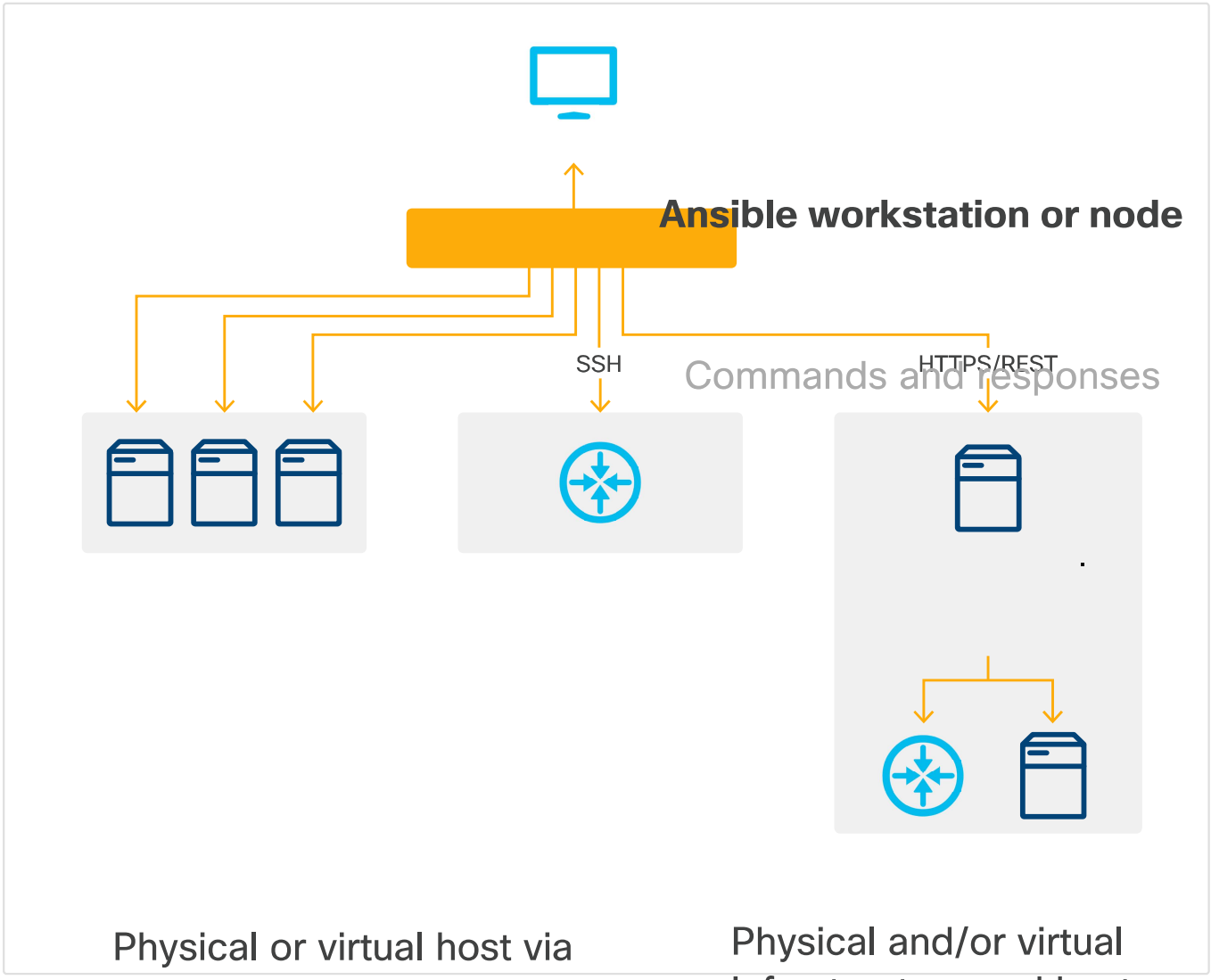
Ansible's basic architecture is very simple and lightweight.

- Ansible's control node runs on virtually any Linux machine running Python 2 or 3, including a laptop, a Linux VM residing on a laptop of any kind, or on a small virtual machine adjacent to cloud-resident resources under management. All system updates are performed on the control node.
- The control node connects to managed resources over SSH. Through this connection, Ansible can:
  - Run shell commands on a remote server, or transact with a remote router, or other network entity, via its REST interface.
  - Inject Python scripts into targets and remove them after they run.
  - Install Python on target machines if required.

- Plugins enable Ansible to gather facts from and perform operations on infrastructure that cannot run Python locally, such as cloud provider REST interfaces.

Ansible is substantially managed from the Bash command line, with automation code developed and maintained using any standard text editor. Atom is a good choice, because it permits easy remote work with code stored in nested systems of directories.

Ansible Architecture



Installing Ansible

The Ansible control node application is installed on a Linux machine (often a virtual machine) from its public package repository. To install Ansible on your workstation, refer to the installation documentation appropriate to your device.

Ansible code structure

In the Ansible code structure, work is separated into YAML (`.yml`) files that contain a sequence of tasks, executed in top-down order. A typical task names and parameterizes a module that performs work, similar to a function call with parameters.

Ansible has hundreds of pre-built Python modules that wrap operating-system-level functions and meta-functions. Some modules like `raw` only do one thing; they present a command in string form to the shell, capture a return code and any console output, and return it in accessible variables. The module `apt` can be used to install, remove, upgrade, and modify individual packages or lists of packages on a Linux web server running a Debian Linux variant. If you want to learn more, the `apt` documentation will give you a sense of the scope and power of Ansible modules.

Playbooks and roles

An Ansible playbook (or "series of plays") can be written as a monolithic document with a series of modular, named tasks. More often, developers will build a model of a complex DevOps task out of low-level playbook task sequences (called "roles"), then reference these in higher-level playbooks, sometimes adding additional tasks at the playbook level.

This segregation of concerns has many benefits.

- **Clarity -** Given a little context, almost anyone can interpret a higher-level playbook referencing clearly-named roles.
- **Reusability and shareability -** Roles are reusable and may be fairly closely bound to infrastructure specifics. Roles are also potentially shareable. The Ansible project maintains a repository for opensource role definitions, called Ansible Galaxy.

Example playbook

The following sample playbook builds a three-node Kubernetes cluster on a collection of servers:

- It installs Python 2 on all servers and performs common configuration steps on all nodes, via a role called `configure-nodes`, installing the Kubernetes software and Docker as a container engine, and configuring Docker to work with Kubernetes. The actual Ansible commands are not shown.
- It designates one node as master, installing the Weave container network (one of many network frameworks that work with Kubernetes), and performing wrapup tasks.
- It joins the `k8sworker` nodes to the `k8smaster` node.
- The statement `become: true` gives Ansible root privileges (via sudo) before attempting an operation.
- The line `gather_facts: false` in the first stanza prevents the automatic `system-facts` interrogator from executing on a target machine before Python is installed. When subsequent stanzas are executed, facts will be compiled automatically, by default.

```
---
- hosts: all
  become: true
  gather_facts: False
  tasks:
  - name: install python 2
    raw: test -e /usr/bin/python || (apt -y update && apt install -y python-
minimal)
- hosts: all
  become: true
  roles:
  - configure-nodes
- hosts: k8smaster
  become: true
  roles:
  - create-k8s-master
  - install-weave
  - wrapup
- hosts:
  - k8sworker1
```

```
  - k8sworker2
  become: true
  roles:
  - join-workers
...
```

Ansible project organization

Ansible projects are typically organized in a nested directory structure as shown below. The hierarchy is easily placed under version control and used for GitOps-style infrastructure as code. For an example, refer to “Directory Layout” in the Ansible documentation.

Ansible folder hierarchy elements

The Ansible folder/file hierarchy includes the following main elements:

- **Inventory files -** Also called hostfiles. These organize your inventory of resources (e.g., servers) under management. This enables you to aim deployments at a sequence of environments such as dev, test, staging, production. For more information about inventory files, refer to “How to build your inventory” in the Ansible documentation.
- **Variable files -** These files describe variable values that are pertinent to groups of hosts and individual hosts.
- **Library and utility files -** These optional files contain Python code for custom modules and the utilities they may require. You may wish to write custom modules and utilities yourself, or obtain them from Ansible Galaxy or other sources. For example, Ansible ships with a large number of modules already present for controlling main features of Cisco ACI, but also provides tutorials on how to compose additional custom modules for ACI features currently lacking coverage.
- **Main playbook files -** Written in YAML, these files may reference one another, or lower-level roles.
- **Role folders and files -** Each role folder tree aggregates resources that collectively enable a phase of detailed configuration. A role folder contains a `/tasks` folder with a `main.yml` tasks file. It also contains a folder of asynchronous `handler` task files. For more information about roles, refer to “Roles” in the Ansible documentation.

Ansible at scale

Ansible's control node is designed to sit close to the infrastructure that it manages. For example, it may reside on a VM, or in a container, running in the same subnet as managed resources. Enterprises and organizations with many hosts under management tend to operate many Ansible control nodes, distributing across infrastructure pools as required.

If you are not doing rapid-fire continuous delivery, Ansible nodes do not even need to be maintained between deployments. If your Ansible deployment code is stored in version control, control nodes can be launched or scratch-built as you need.

There are scaling challenges for large organizations, such as managing and controlling access to many Ansible nodes flexibly and securely. This also includes putting remote controllers seamlessly and safely under control of centralized enterprise automation. For this, there are two control-plane solutions:

- The commercial Red Hat Ansible Tower product provides a sophisticated web interface, REST API, and rich, role-based access control options.

- The open-source, feature-comparable alternative, AWX project, of which Ansible Tower is a value-added distribution. AWX, however, is said to represent a development branch that undergoes minimal testing, and is not made available via signed binaries. This may be a problem for many enterprises.

Continuous delivery around Ansible deployment can be performed with any general-purpose CI/CD automation tool such as Jenkins or Spinnaker. Larger, more complex projects often use the Zuul open source gating framework, originally developed by the OpenStack Project and spun off independently in 2018. The AWX Project, among many others, is a Zuul-gated project.

Larger-scale Ansible implementations will also benefit from Ansible Vault, a built-in feature that enables encryption of passwords and other sensitive information. It provides a straightforward and easily-administered alternative to storing sensitive information in playbooks, roles, or elsewhere as plaintext.

Cisco Ansible resources

Cisco and the Ansible community maintain extensive libraries of Ansible modules for automating Cisco compute and network hardware including:

- A very large set of built-in modules for configuring Cisco Application-Centric Infrastructure fabrics via the Application Policy Infrastructure Controller (APIC). These modules execute on the Ansible host (not the controller), communicating via the controller's REST interface.
- Remote control of Cisco network devices running IOS-XR, plus modules for sending commands and retrieving results from these devices via CLI, or via the standard NETCONF REST interface.
- Ansible modules for configuring Cisco UCS infrastructure via the Intersight REST interface.

7.4.6

Ansible Example

This exercise will let you view the structure of a simple Ansible playbook, which retrieves information about the container the demo environment resides in. Note that Ansible normally uses `ssh` to connect with remote hosts and execute commands. In this example, a line in the top-level `playbook.yml` file instructs Ansible to run this playbook locally, without requiring `ssh`.

```
connection: local
```

Normally, Ansible is used to perform deployment and configuration tasks. For example, you might use it to create a simple website on a remote host. Let's see how this might work.

Prerequisites

You can walk through this exercise yourself, or you can simply read along. If you want to complete this exercise, you will need:

- A target host running a compatible operating system (such as Ubuntu 18.04 server)

- SSH and keywise authentication configured on that host
- Ansible installed on your local workstation

This is typically how new virtual machines are delivered on private or public cloud frameworks.

For the target host, you can create one using a desktop virtualization tool like VirtualBox.

Building an Ansible project file tree

For the purposes of this exercise, the target machine's (DNS-resolvable) hostname is simply `target`. If you have your own target host set up and configured, substitute your host name when you create your files.

With your target machine SSH-accessible, begin building a base folder structure for your Ansible project.

```
mkdir myproject
cd myproject
```

At the top level in your project folder, you need:

- An inventory file, containing information about the machine(s) on which you want to deploy.
- A top level `site.yml` file, containing the most abstract level of instructions for carrying out your deployment.
- A role folder structure to contain your `webserver` role.

```
touch inventory
touch site.yml
mkdir roles
cd roles
ansible-galaxy init webservers
```

Creating your inventory file

Your inventory file for this project can be very simple. Make it the DNS-resolvable hostname of your target machine:

```
[webservers]
target # can also be IP address
```

You are defining a group called `webservers` and putting your target machine's hostname (or IP) in it. You could add new hostnames/IPs to this group block, or add additional group blocks, to assign hosts for more complex deployments. The name `webservers` is entirely arbitrary. For example, if you had six servers you wanted to configure in a common way, and you then configure three as webservers and three as database servers, your inventory might look like this:

```
[webservers]
target1
```

```
target2
target3
[dbservers]
target4
target5
target6
```

You don't actually need to create a `common` group, because Ansible provides means to apply a common configuration to all servers in an inventory, which you'll see in a moment.

Creating your top level playbook file

A top-level playbook typically describes the order, permissions, and other details under which lower-level configuration acts, defined in roles, are applied. For this example project, `site.yml` looks like this:

```
---
- hosts: webservers
  become: true
  roles:
  - webservers
```

`site.yml` identifies which hosts you want to perform an operation on, and which roles you want to apply to these hosts. The line `become: true` tells Ansible that you want to perform the roles as root, via sudo.

Note that instead of `hosts: webservers`, you could apply this role to all target hosts (which, in this case, would work fine, because you only have one target) by substituting the line:

```
- hosts: all
```

Creating your webservers role

Next step is to create the role that installs and configures your web server. You've already created the folder structure for the role using `ansible-galaxy`. Code for the role is conventionally contained in a file called `main.yml` in the role's `/tasks` directory. You can edit the `roles/webserver/tasks/main.yml` file directly, to look like this:

```
---
- name: Perform updates and install apache2
  apt:
    name: apache2
    state: present
    update_cache: yes
- name: Insert new homepage index.html
  copy:
    src: index.html
    dest: /var/www/html
```

```
    owner: myname
    mode: '0444'
```

The role has two tasks:

- Deploy Apache2.
- Copy a new `index.html` file into the Apache2 HTML root, replacing the default `index.html` page.

In the `apt:` stanza, you name the package, its required state, and instruct the apt module to update its cache. You are basically performing a `sudo apt update` before the installation happens.

In the second stanza, Ansible's copy routine moves a file from your local system to a directory on the target and also changes its owner and permissions. This is the equivalent of:

```
chown myname index.html
chmod 444 index.html
```

Creating your index.html file

Of course, you'll need to create a new `index.html` file as well. The Ansible copy command assumes that such files will be stored in the `/files` directory of the role calling them, unless otherwise specified. Navigate to that directory and create the following `index.html` file, saving your changes afterward:

```
<html>
<head>
<title>My Website</title>
</head>
<body>
<h1>Hello!</h1>
</body>
```


Now you're ready to run your deployment. From the top level directory of your project, you can do this with the statement:

```
ansible- -i inventory -u myname -K site.yml
```

- `-i`: names your inventory file.
- `-u` argument names your sudo user.
- `-K` tells Ansible to ask us for your sudo password, as it begins execution.
- `site.yml` is the file that governs your deployment.

If all is well, Ansible should ask us for your BECOME password (sudo password), then return results similar to the following:

```
BECOME password:
PLAY [webservers] ***************************************************************
TASK [Gathering Facts] **********************************************************
ok: [192.168.1.33]
TASK [webservers : Perform updates and install apache2]
***************************
changed: [192.168.1.33]
TASK [webservers : Insert new homepage index.html]
*******************************
changed: [192.168.1.33]
PLAY RECAP **********************************************************************
192.168.1.33               : ok=3    changed=2    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

And now, if you visit the IP address of your target machine in a browser, you should see your new homepage.

Note that Ansible gives back a full report on each execution, noting whether a step was actually performed, or whether Ansible determined that its desired goal was already reached (In that case, nothing happened, but the step was considered to have completed 'ok'.). This is an example of how Ansible maintains idempotency. You can typically run an Ansible deployment as many times as needed without putting a target system into an unknown state.

Ansible CI/CD walkthrough

Let's walk through the example as if they were part of a CI/CD pipeline.

A developer collaborating with you on GitHub commits a change to the website, such as in the `index.html` file.

Next, tests in the repository execute syntax and sanity checks as well as code review rules against each pull request, for example.

If tests **pass**: Accepts their commit and notifies CI/CD server to run tests.

If tests **fail**: Rejects their commit based on failed checks and asks them to resubmit.

Next, the CI/CD system, such as Jenkins, prepares an environment and runs predefined tests for any Ansible playbook. It should indicate the version expected each time and install it. Here's an example pipeline:

```
pip install ansible==2.9.2
ansible-version
ansible-playbook main.yml --syntax-check
ansible-playbook -i staging-servers.cfg main.yml --check
ansible-playbook -i staging-servers.cfg main.yml -vvvv
```

With this pipeline, you make sure:

- There are no syntax errors in `main.yml`.

- Code review rules are being followed before the code even gets merged into the repository.
- All modules are spelled correctly and available in the environment with the `-check` parameter.
- There's a common Ansible version, such as 2.9.2, because it's installed in the CI environment.
- The playbook can run in the staging environment on the staging servers.
- Other people can see what was run, using the highly verbose parameter, `-vvvv`.

After Jenkins is done running the job, you can get a notification that all is ready for staging and you can push these changes to production with another pipeline, this time for pushing to production. This is the power of code version control, Ansible, and multiple promotion through environments using CI/CD.

7.4.7

Lab – Use Ansible to BackUp and Configure a Device

In this lab, you will explore the fundamentals of how to use Ansible to automate some basic device management task. First, you will configure Ansible in your DEVASC VM. Next, you will use Ansible to connect to the CSR1000v and back up its configuration. Finally, you will configure the CSR1000v with IPv6 addressing.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM and CSR1000v VM
- Part 2: Configure Ansible
- Part 3: Use Ansible to Back Up a Configuration
- Part 4: Use Ansible to Configure a Device

Use Ansible to Back Up and Configure a Device

7.4.8

Lab – Use Ansible to Automate Installing a Web Server

In this lab, you will first configure Ansible so that it can communicate with a webserver application. You will then create a playbook that will automate the process of installing Apache on the webserver. You will also create a customized playbook that installs Apache with specific instructions.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Perform a Backup with Ansible
- Part 3: Configure IPv6 Addressing with Ansible
- Part 4: Use Ansible to install Apache on Web Servers

- Part 5: Add Options to Your Ansible Playbook for Apache Web Servers

Use Ansible to Automate Installing a Web Server

7.4.9

Puppet

Puppet was founded as open source in 2005 and commercialized as Puppet Enterprise by Puppet Labs in 2011.

Puppet's core architecture has the following characteristics:

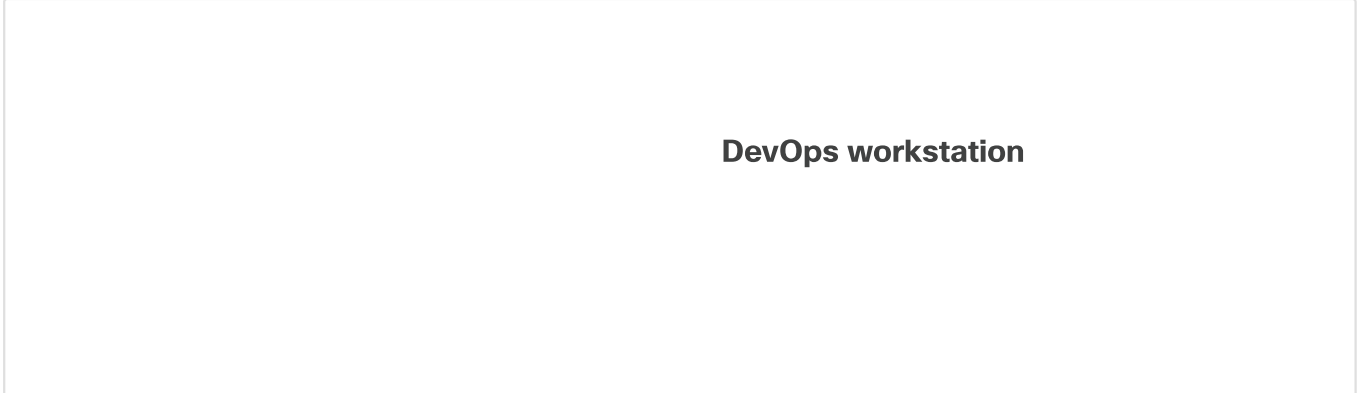
- A designated server to host main application components:
  - The Puppet Server (historically called "Puppet Master")
  - Facter, the fact-gathering service
  - PuppetDB, which can store facts, node catalogs, and recent configuration event history

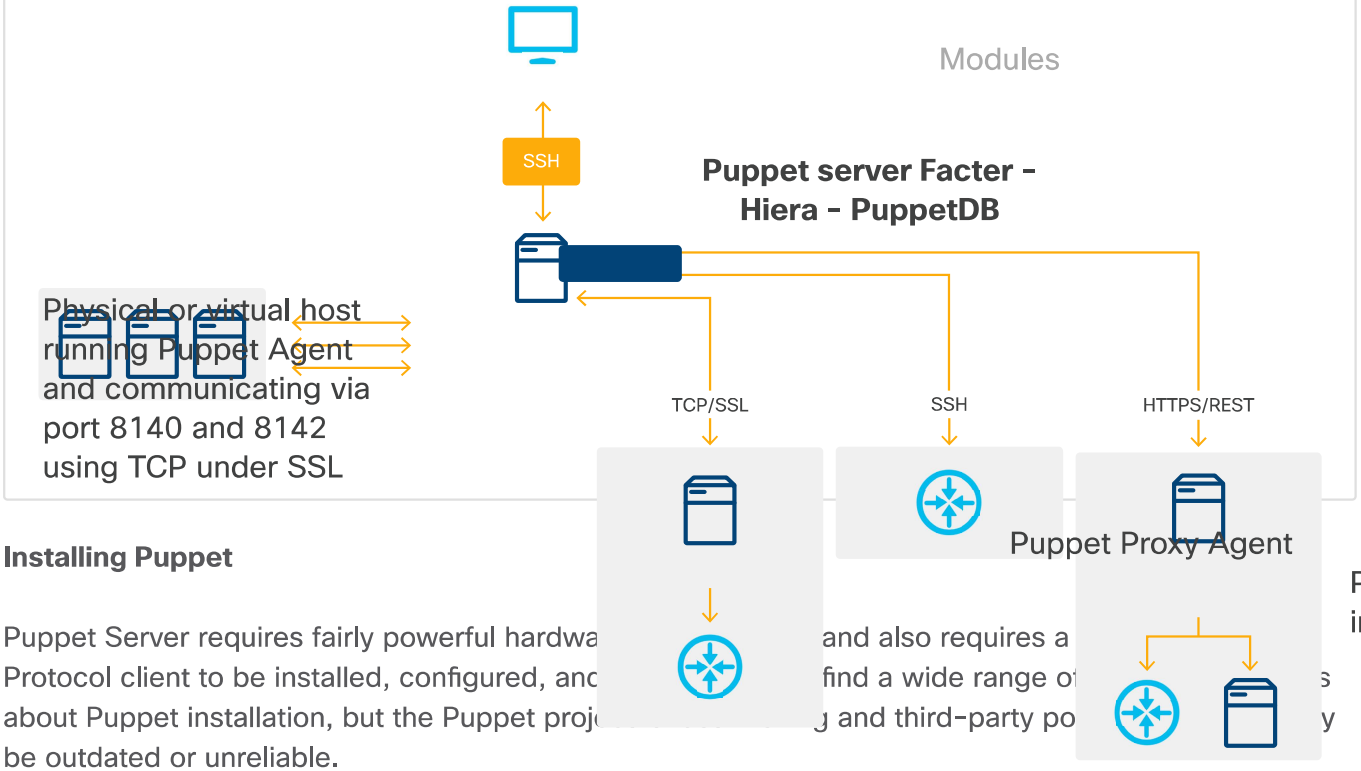
- A secure client, also known as a Puppet Agent, installed and configured on target machines. Clients and server are mutually authenticated with self-signed certificates, and SSL is used for transport. The agents gather facts (under control of the Facter service) and make configuration changes as directed by the Puppet Server.
- For cloud APIs and hardware that cannot run an agent, Puppet has modules available to enable these connections.
- In scaled-out implementations where many non-agent-capable devices are under management, Puppet enables a proxy agent to offload the work of directly connecting to device CLIs and exchanging information.

Operators communicate with the Puppet Server largely via SSH and the command line.

The Puppet Server can be a VM or even a Docker container for small self-teaching implementations, and Puppet provides a compact Docker install for this purpose, called PUPPERWARE. Standard packages are available for building a Puppet Server on Linux, which is currently the only option for a Server install. Puppet Agents (also called Clients) are available for Linux, Windows, and MacOS.

Puppet Architecture





Installing Puppet

Puppet Server requires fairly powerful hardwa and also requires a Protocol client to be installed, configured, anc find a wide range of about Puppet installation, but the Puppet proj and third-party po be outdated or unreliable.

When you have Puppet Server running, you can begin installing Puppet Agents on hosts you wish to manage. The agents will then need to have the `puppet.conf` file configured to communicate with a Puppet Server. After the client service is started, it will have its certificate signed by the server. The Server will now be able to gather facts from the client and update the client state with any configuration changes.

Puppet code structure

Like Ansible, Puppet lets you store components of a project or discrete configuration in a directory tree (`/etc/puppetlabs/code/environments`). Subsidiary folders are created according to the configuration in `puppet.conf`, or by the operator.

As an example, you may declare `environment = production` in `puppet.conf`. Puppet will create a directory for this default environment, which will contain a `modules` subdirectory in which you can store subsidiary projects and manifests for things you need to build and configure (`/etc/puppetlabs/code/environments/production/modules`).

To begin a small project, you might create a folder inside this directory, and then within that folder, create another called `manifests`, where you would store the manifest files declaring operational classes. These are units of code describing a configuration operation. Manifest files typically end in the extention `.pp`, and are written in Puppet's declarative language, which looks something like Ruby, and was inspired by the Nagios configuration file format.

Like Ansible and other configuration tools, Puppet provides a host of resources that can be invoked to define configuration actions to be performed on hosts and connected infrastructure. The basic idea is very similar to Ansible's practice, where a class that invokes a resource will be parameterized to function idempotently, and will be applied in context to produce the same desired result every time it runs.

Puppet comes with a set of basic resources (templates for performing configuration actions) built in. Many additional resources for performing all sorts of operations on all kinds of infrastructure can be

downloaded and installed from Puppet Forge using the puppet module command.

Puppet at scale

Puppet Server is somewhat monolithic, and a monolithic installation is recommended by the (open source) implementors, who say an appropriately configured Puppet Server can manage "up to 4000" hosts.

The first recommended step to accommodate more hosts is to create additional "compile masters", which compile catalogs for client agents and place these behind a load balancer to distribute work.

Puppet Enterprise customers can further expand capacity by replacing PuppetDB with a stand-alone, customized database called PE-PostgreSQL. The Puppet Enterprise product offers many other conveniences as well, including a web-based console giving access to reports, logs, and enabling certain kinds of point-and-click configuration.

Cisco Puppet resources

Cisco and the Puppet community maintain extensive libraries of modules for automating Cisco compute and network hardware including:

- Cisco IOS modules enabling management of IOS infrastructure.
- Cisco UCS modules enabling control of UCS via UCS Manager

7.4.10

Puppet Example

This example describes how to install Puppet and then use Puppet to install Apache 2 on a device. You can simply read along to better understand Puppet.

This approximates the normal workflow for Puppet operations in an automated client/server environment. Note that modules can be completely generic and free of site-specific information, then separately and re-usably invoked to configure any number of hosts or infrastructure components. Because modules and manifests are composed as text files, they can easily be stored in coordinated fashion in a version control repository, such as Git.

Installing Puppet Server

Puppet Server requires fairly powerful hardware (or a big VM), and also requires a Network Time Protocol client to be installed, configured, and tested. Instructions for installing the server can be found in Puppet's documentation.

Installing Puppet Client

When you have Puppet Server running, you can install Puppet Agents on a host. For example, on a Debian-type Linux system, you can install Puppet Agent using a single command:

```
sudo apt-get install puppet-agent
```

Modify

When installed, the Puppet Agent needs to be configured to seek a Puppet Server. Add the following lines to the file `/etc/puppet/puppet.conf`:

```
[main]
certname = puppetclient
server = puppetserver
environment = production
runinterval = 15m
```

This tells the Client the hostname of your server (resolved via `/etc/hosts`) and the name of the authentication certificate that you will generate in the next step.

Start the puppet service on the Client:

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

You should get a response similar to the following:

```
Notice: /Service[puppet]/ensure: ensure changed 'stopped' to 'running'
service { 'puppet':
  ensure => 'running',
  enable => 'true',
}
```

Note Puppet's declarative method of configuration. Puppet uses itself to configure and activate its service. Note the Ruby-like declarative syntax.

Certificate signing

Puppet Agents use certificates to authenticate with the server before retrieving their configurations. When the Client service starts for the first time, it sends a request to its assigned server to have its certificate signed, enabling communication.

On the Server, issuing the `ca list` command returns a list of pending certificates:

```
sudo /opt/puppetlabs/bin/puppetserver ca list
```

The response should be similar to the following:

```
Requested Certificates:
    puppetclient   (SHA256)
44:9B:9C:02:2E:B5:80:87:17:90:7E:DC:1A:01:FD:35:C7:DB:43:B6:34:6F:1F:CC:DC:C2:E9:DD:72
```

You can then sign the certificate, enabling management of the remote node:

```
sudo /opt/puppetlabs/bin/puppetserver ca sign --certname puppetclient
```

The response:

```
Successfully signed certificate request for puppetclient
```

The Server and Client are now securely bound and able to communicate. This will enable the Server to gather facts from the Client, and let you create configurations on the Server that are obtained by the client and used to converge its state (every 15 minutes).

Creating a configuration

Like Ansible, Puppet lets you store components of a project or discrete configuration in a directory tree:

```
/etc/puppetlabs/code/environments
```

Subsidiary folders are created according to the configuration in `puppet.conf` or by the operator. In this example, having declared `environment = production` in puppet.conf, Puppet has already created a directory for this default site, containing a `modules` subdirectory in which we can store subsidiary projects and manifests for things we need to build and configure.

```
/etc/puppetlabs/code/environments/production/modules
```

You will now install Apache2 on your managed client. Puppet operations are typically performed as root, so become root on the Server temporarily by entering:

```
sudo su -
```

Navigate to the `/modules` folder in the `/production` environment:

```
cd /etc/puppetlabs/code/environments/production/modules
```

Create a folder structure to contain the `install apache` module:

```
mkdir -p apache2/manifests
cd apache2/manifests
```

Inside this manifests folder, create a file called `init.pp`, which is a reserved filename for the initialization step in a module:

```
```

```
class apache2 {
  package { 'apache2':
    ensure => installed,
  }
  service { 'apache2':
    ensure  => true,
    enable  => true,
    require => Package['apache2'],
  }
}
```

The class definition orders the steps we want to perform:

Step 1. Invoke the `package` resource to install the named package. If you wanted to remove the package, you could change `ensure => installed` to read `ensure => absent`.

Step 2. Invoke the service resource to run if its requirement (in this case, that Apache2 is present) is met. Instruct it to ensure that the service is available, and then enable it to restart automatically when the server reboots.

Navigate to the associated `manifests` folder:

```
cd /etc/puppetlabs/code/environments/production/manifests
```

Create a `site.pp` file that invokes the module and applies it to the target machine:

```
node 'puppetclient' {
   include apache2
}
```

Deploying the configuration

You have two options to deploy the completed configuration:

- Restarting the Puppet Server will now cause the manifests to be compiled and made available to the Puppet Agent on the named device. The agent will retrieve and apply them, installing Apache2 with the next update cycle:

```
systemctl restart puppetserver.service
```

- For development and debugging, you can invoke Puppet Agent on a target machine (in this case our Puppet Client machine):

```
sudo puppet agent -t
```

- The agent will immediately interrogate the server, download its catalog (the configurations that reference it) and apply it. The results will be similar to the following:

```
root@target:/etc/puppetlabs/code/environments/production/manifests# puppet agent -
tInfo: Using configured environment 'production'Info: Retrieving pluginfactsInfo:
Retrieving pluginInfo: Retrieving localesInfo: Caching catalog for puppetagentInfo:
Applying configuration version '1575907251'Notice:
/Stage[main]/Apache2/Package[apache2]/ensure: createdNotice: Applied catalog in
129.88 seconds
```

After the application has been successfully deployed, enter the target machine's IP address in your browser. This should bring up the Apache2 default homepage.

7.4.11

Chef

Chef

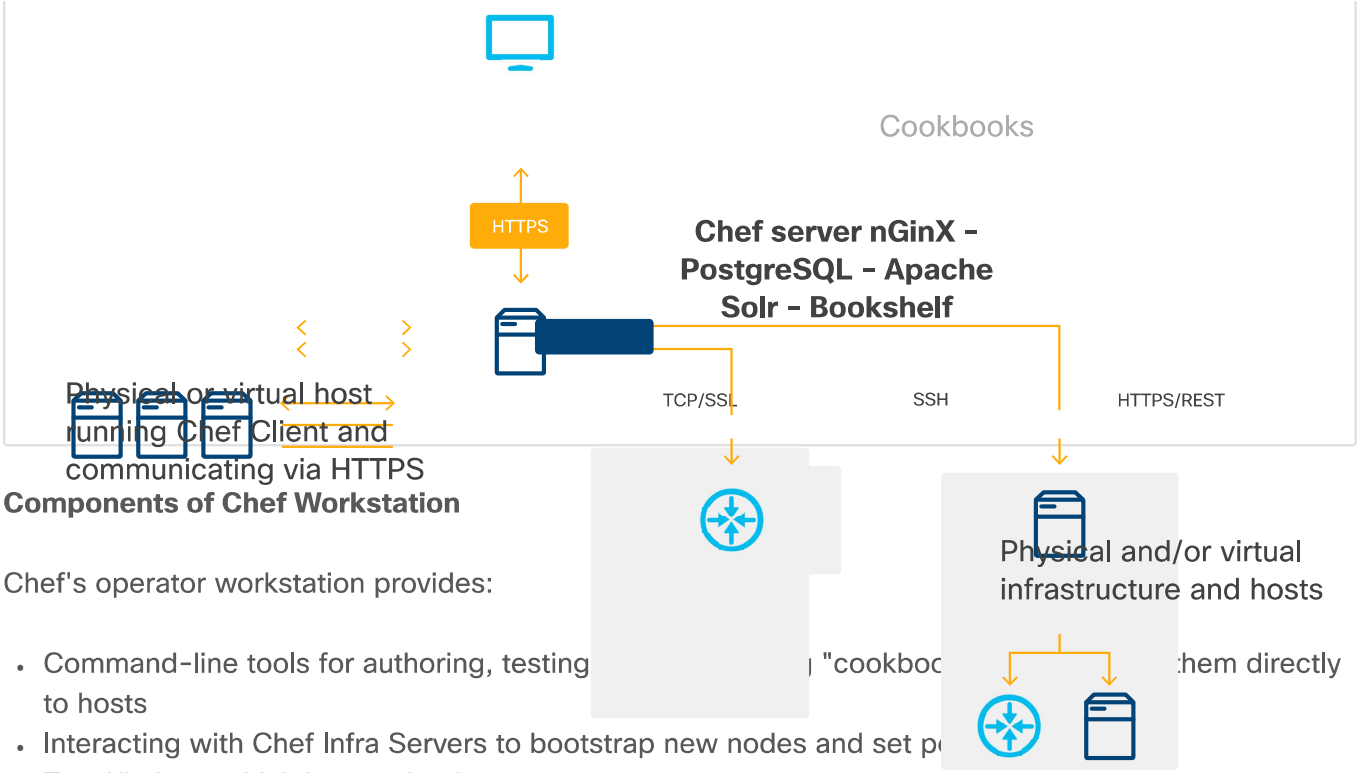
Chef provides a complete system for treating infrastructure as code. Chef products are partly licensed, but free for personal use (in Chef Infra Server's case, for fewer than 25 managed nodes).

Chef's products and solutions enable infra-as-code creation, testing, organization, repository storage, and execution on remote targets, either from a stand-alone Chef Workstation, or indirectly from a central Chef Infra Server. You should be aware of the main Chef components:

- **Chef Workstation** - A standalone operator workstation, which may be all that smaller operations need.
- **Chef Infra Client (the host agent)** - Chef Infra Clients run on hosts and retrieve configuration templates and implement required changes. Cookbooks (and proxy Clients) enable control of hardware and resources that cannot run a Chef Infra Client locally (such as network devices).
- **Chef Infra Server** - Replies to queries from Chef Infra Agents on validated hosts and responds with configuration updates, upon which the Agents then converge host configuration.

Most configuration tasks can also be carried out directly between Chef Workstation and managed nodes and devices.

Chef Architecture



Components of Chef Workstation

Chef's operator workstation provides:

- Command-line tools for authoring, testing "cookboo hem directly to hosts
- Interacting with Chef Infra Servers to bootstrap new nodes and set p
- Test Kitchen, which is a testing harness
- ChefSpec, which simulates the effects of code before implementing changes
- InSpec, a security/compliance auditing and testing framework

Chef provides hundreds of resources for performing common configuration tasks in idempotent fashion, as well as Chef Supermarket, a community-maintained sharing site for Cookbooks, custom resources, and other solutions.

Code is maintained in a local repository format called `chef-repo`, which can be synchronized with Git for enterprise-wide infra-as-code efforts. Within a repo, code is organized in "cookbook" folders, comprising "recipes" (actual linear configuration code, written in Chef's extended Ruby syntax), segregated attributes, custom resources, test code, and metadata.

Chef's domain-specific language (DSL) enables you to address configuration tasks by authoring a sequence of small, bracketed templates, each of which declares a resource and parameterizes it. Chef resources tend to be more abstract than Ansible's or Puppet's, which helps address cross-platform concerns. For example, the package resource can determine the kind of Linux, MacOS, or Windows environment that it is running on and complete a required installation in a platform-specific way.

Installing Chef Workstation

To begin using Chef, a good first step is to install Chef Workstation, which provides a complete operations environment. Workstation is available for Linux and Windows. Refer to the Chef Workstation downloads page for more information.

When Workstation is installed, you can use it immediately to start making configuration changes on accessible hosts. Some node preparation is helpful before trying to manage a target node with Chef. You should configure SSH keywise access to the host rather than using passwords. And it helps (if you are not running DNS) to include the IP address and hostname of the target machine in your Workstation machine's `/etc/hosts` file.

Running Chef at scale

Chef Infra Server was rewritten several years back in Erlang, to increase its capacity, enabling management of up to about 10,000 hosts. It can be configured for high availability by deploying its front-end services (including NGINX and stateless application logic) into an array of load-balanced proxies, which connect to a three-server active/active cluster supporting back-end services like elasticsearch, etcd, and PostgreSQL.

Chef also provides an array of products that together solve most of the problems enterprises face in dealing with increasingly-complex, large-scale, hybrid infrastructures. Its on-Workstation `chef-repo` structures harmonize with Git, enabling convenient version control and collaboration on DevOps code, and simplifying transitions to infra-as-code regimes. Its core philosophy of continuous configuration management dovetails well with the goal of continuous IT delivery.

Chef's built-in unit testing framework Test Kitchen, pre-deployment simulators, and companion auditing and security assessor InSpec provide the rest of a purpose-built DevOps test-driven development framework.

Cisco Chef Resources

Cisco has developed modified Chef Infra Agents that run in the guest shell of NX-OS switch equipment, enabling this hardware to work with Chef as if it were a managed host. Cisco has also developed, and maintains a Cisco Chef Cookbook for NX-OS infrastructure, available on Chef Supermarket.

A GitHub public repo of cookbook and recipe code is also maintained, to enable control of a wide range of Cisco products.

Cisco UCS infrastructure is easily managed with Chef, via a cookbook enabling integration with Integrated Management Controllers. Management via UCS Manager and Intersight is possible via Python and/or PowerShell SDKs.

7.4.12

Chef Example - Install and Use Chef

This example describes how to install Chef and use it to install Apache 2 on a device. You can simply read along to better understand Chef.

Installing Chef Workstation

Chef Workstation provides a complete operations environment. Workstation is available for Linux and Windows. The following example assumes you're installing on an Ubuntu 18.04 LTS virtual machine.

If your machine is set up with a standard desktop, you can browse to the Chef Workstation downloads page, find the download for Ubuntu 18.04, and install it automatically with the Debian package manager.

Alternatively, you can install from the command line by copying the URL of the `.deb` package and using the following steps:

```
wget https://packages.chef.io/files/stable/chef-
workstation/0.12.20/ubuntu/18.04/chef-workstation_0.12.20-1_amd64.deb
sudo dpkg -i chef-workstation_0.12.20-1_amd64.deb
```

Basic configuration management

After Workstation is installed, you can use it immediately to start making configuration changes on accessible hosts. You will use the `chef-run` command for this. It is a subsystem that takes care of bootstrapping a Chef Infra Agent onto the target host and then executes whatever commands you reference in files or provide in arguments.

The first time you use `chef-run` (or other Chef tools), you may be asked to accept licensing terms (type `yes`) for the utility you're using, or for subsystems it invokes.

For the first configuration exercise, you will provide the information Chef needs to install the `ntp` package. In the process, you will provide the remote username, their sudo (become root) password, the name of the remote host `target` (This is in your `/etc/hosts` file. Otherwise, you would use its IP address here.) and the name of the resource verb:

```
chef-run -U myname -sudo <password> target package ntp action=install
```

Chef connects to the node, initially via SSH, and bootstraps the Chef Infra Client onto it (if not already present). This can take a while, but `chef-run` helpfully shows you activity indicators. When the client is installed, the task is handed to it, and the process completes. You get back something that looks like this:

```
[✓] Packaging cookbook... done!
[✓] Generating local policyfile... exporting... done!
[✓] Applying package[ntp] from resource to target.
└── [✓] [target] Successfully converged package[ntp].
```

Note the vocabulary Chef uses to describe what it is doing. The configuration action that you request is treated as a policy that partially describes the target machine's desired state. Chef does what is required to make that policy happen, converging the machine to its desired state, where NTP is installed and its time-synchronization service is activated by default.

Installing Chef Infra Client

Chef Infra Client runs locally on conventional compute nodes. It authenticates to Chef Infra Server using public keypairs, which are generated and signed when a node is registered with a Chef Infra Server. This ensures that rogue nodes cannot request configuration information from the Server. Communications between authorized Clients and their Server are safely encrypted with TLS.

Chef Infra Client includes a discovery subsystem called Ohai, which collects system facts and uses them to determine whether (and how) the target system has drifted from its configuration, and needs to be converged.

Chef Workstation can bootstrap Infra Client onto target nodes. You can also preinstall Infra Client on nodes, for example, while creating new nodes on a public cloud. Below is an example script you might

run on a target host to do this. Note that user data scripts run as root, so `sudo` is not required here. However, if you log into a remote manually as a user (perhaps in the `sudoers` group) rather than as root, you would need to assume root privileges (using `sudo su -`) before creating and running this script locally.

The script uses a Chef-provided installer called Omnitruck to do this. The Omnitruck shell script figures out which kind of Linux distribution you are using and otherwise enables a safe, predictable installation of Chef software (you can also use it to install other Chef products). A Windows version of this script is also available that runs on PowerShell:

```
#!/bin/bash
apt-get update
apt-get install curl
curl -L https://omnitruck.chef.io/install.sh | bash -s once -c current -p chef
```

Note that the parameters shown above will install the latest version of the Chef client, and do not pin the version. This is dangerous for production work, because it permits updates to occur without warning, possibly introducing an incompatibility between the Client and Workstation or Server. The `-v` option lets you install a specific version of the Client, and pins it automatically. Bootstrapping a node with Chef installs the latest compatible version and pins it.

Chef Infra Server prerequisites

Chef Infra Server stores configuration and provides it to Clients automatically, when polled, enabling Clients to converge themselves to a desired state. Downloadable packages are listed and linked on this page. The server is free to use for fewer than 25 hosts.

Before installing Chef Infra Server, install openssh-server and enable keywise access. You would also need to install NTP for time synchronization. You can do this with Chef, or you can do it manually:

```
sudo apt-get install ntp ntpdate net-tools
```

On an Ubuntu system, turn off the default `timedatectl` synchronization service to prevent it from interfering with NTP synchronization:

```
sudo timedatectl set-ntp 0
```

After NTP is installed, ensure that it is synchronizing with a timeserver in its default pool. This may take a few minutes, so repeat the command until you see something like this:

```
ntpstat
synchronised to NTP server (198.60.22.240) at stratum 2
   time correct to within 108 ms
   polling server every 256 s
```

When this shows up, you can install Chef Infra Server.

Installing Chef Infra Server

To install Chef Infra Server on Ubuntu 18.04, you can perform steps similar to the manual Workstation install, above, after obtaining the URL of the .deb package. At time of writing, the current stable version was 13.1.13-1.

```
wget https://packages.chef.io/files/stable/chef-server/13.1.13/ubuntu/18.04/chef-server-core_13.1.13-1_amd64.deb
sudo dpkg -i chef-server-core_13.1.13-1_amd64.deb
```

After Chef Infra Server is installed, issue the following command to tell it to read its default configuration, initialize, and start all services. This is a long process, and is done by Chef itself, giving you an opportunity to see some of Chef's logic used to apply configuration details in converging a complex application to a desired state:

```
sudo chef-server-ctl reconfigure
```

The configuration process may initially fail on low-powered or otherwise compromised VMs, but because this is Chef (thus idempotent) it can be run more than once in an attempt to get it to complete. This sometimes works. If it does not work, it is a sign that you are trying to run on hardware (or virtualware) that is not powerful enough, and should be upgraded before trying again.

When `chef-server-ctl` begins the reconfigure process, on an initial run, you would be expected to accept several product licenses. Type `yes` at the prompt.

Create an Infra Server user:

```
sudo chef-server-ctl user-create <username> <firstname> <lastname> <email> '<password>' --filename <key_file_path_and_name.pem>
```

Provide your own preferred user information for the `<>`-bracketed terms (removing the `<>` brackets from your actual responses) and include a password. The argument to `--filename` provides a pre-existing and accessible path and filename for the `.pem` private key file that Chef will create for this user. This key file will need to be downloaded from the server and established on the Workstation to enable server management. It makes sense to store this key in a folder that is readily accessible from your OS user's (`myname`'s) home directory. **IMPORTANT**: Remember the key file path and filename!

Next, you create an organization, which is a structure Chef uses to isolate different bodies of configuration policy from one another. These can be actual organizations, or a concept more like 'sites'. Chef issues each organization an RSA key which is used to authenticate hosts to Server and organization, thus providing a multi-tenant infrastructure.

Provide a short name for the organization, a full descriptive name, the username you created in the last step to associate with the organization, and an accessible path to store the validation key. By convention (though this is optional) the key can be called `<ORGANIZATION>-validator.pem`:

```
sudo chef-server-ctl org-create <short_name> '<full_organization_name>' --association_user <username_you_just_created> --filename <key_file_path_and_name.pem>
```

It makes sense to store this key in the same directory that you used to store the user key generated in the prior step.

Install Chef-Manage

You can also install the web interface for Chef server. This can be done by entering:

```
sudo chef-server-ctl install chef-manage
```

When the process completes, restart the server and manage components. These are Chef operations, and may take a while, as before.

```
sudo chef-server-ctl reconfigure
(lots of output)
sudo chef-manage-ctl reconfigure --accept-license
(lots of output)
```

The argument `--accept-license` prevents `chef-manage-ctl` from stopping to ask you about the unique licenses for this product. When this process is complete, you can visit the console in a browser via `https://<IP_OF_CHEF_SERVER>`. Note that most browsers will return an error about the server's self-signed certificate, and you will need to give permission to connect. Use the username/password you created above with `chef-server-ctl user-create.`

Initially, there is not much to see, but this will change when you register a node.

Finish configuring Workstation

Before Chef Workstation can talk to your Infra Server, you need to do a little configuration. To begin, retrieve the keys generated during Server configuration, and store them in the folder `/home/myname/.chef`, created during Workstation installation:

```
cd /home/myname/.chef
scp myname@chefserver:./path/*.pem .
```

`/path/` is the path from your (`myname`) home directory on the Server to the directory in which the Server stored keys.

If you are not using keywise authentication to your Server, `scp` will ask for your password (your original user password, not your Chef Server user password). The `.` after `user@host:` refers to your original user's home directory, from which the path is figured. The wildcard expression finds files ending in `.pem` at that path. The closing dot means "copy to the current working directory" (which should be the `.chef` folder). Run the `ls` command from within the .`chef` folder to see if your keys made it.

7.4.13

Chef Example – Prepare to Use Knife

Simply read along with this example to better understand Chef.

Prepare to use Knife

Knife is a tool for managing cookbooks, recipes, nodes, and other assets, and for interacting with the Chef Infra Server. You need to give Knife a solid initial configuration.

Within the `.chef` folder, edit the (initially empty) file named `config.rb`, and include the following lines of code, adapting them to your environment:

```
chef_server_url 'https://chefserver/organizations/<short_name>'
client_key '/home/<myname>/.chef/<userkey.pem>'
cookbook_path [
  '/home/<myname>/cookbooks'
]
data_bag_encrypt_version 2
node_name '<username>'
validation_client_name '<short_name>-validator'
validation_key '/home/<myname>/.chef/<short_name>-validator.pem'
```

This configuration identifies the Chef Server and the organization you created, the path to your user key (created during Infra Server post-install configuration), the path you want to use for cookbook file trees, a desired encryption level for "databag" storage (for now, just set it to the recommended value), your server username, a name derived from the name of your validation key, and the local path to that key itself. Change the `<>` bracketed names to your own values.

Save the `config.rb` file, then create the directory `/home/myname/cookbooks`:

```
mkdir /home/myname/cookbooks
```

Finally, issue the command:

```
knife ssl fetch
```

If you have correctly set up the `config.rb`, Knife will consult with your server, retrieve its certificate, and store it in the directory:

```
Certificate stored in: /home/myname/.chef/trusted_certs
```

Chef will find this automatically when it is time to connect with the server, providing assurance that the server is authentic.

Bootstrap a target node with knife

Now that Knife is configured, you can bootstrap your target node. It is fine that you are doing this a second time (remember, earlier, you installed Chef Infra Client manually).

To bootstrap, issue the following command, replacing variable fields with your information. The command is set up to use keywise authentication to the target machine. The somewhat-redundant `--sudo` and `--use-sudo-password commands` tell Knife to use sudo to complete its work. The -P option provides your sudo password on the target machine. `<name_for_your_node>` is an arbitrary name. The `--ssh-verify-host-key never` flag and argument cause the command not to pause and ask your permission interactively if/when it finds that you've never logged into this server before:

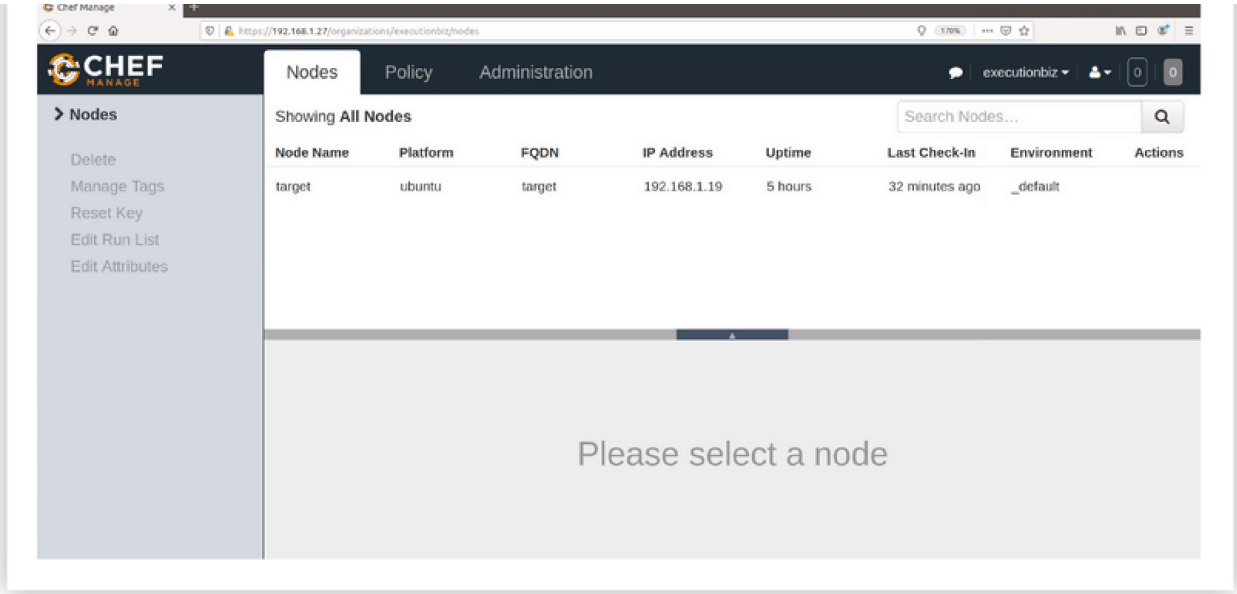
```
knife bootstrap target --connection-user <myname> -i ~/.ssh/id_ed25519 --sudo --
use-sudo-password -P <sudo_password> --node-name <name_for_your_node> --ssh-verify-
host-key never
```

If the command works correctly, you would get back something that looks like this. Note that Chef has detected the earlier installation and has not overwritten it:

```
[target] [sudo] password for myname:
[target] -----> Existing Chef Infra Client installation detected
[target] Starting the first Chef Infra Client Client run...
[target] +---------------------------------------------+
✓ 2 product licenses accepted.
+---------------------------------------------+
[target] Starting Chef Infra Client, version 15.5.17
[target]
[target] Creating a new client identity for target using the validator key.
[target]
[target] resolving cookbooks for run list: []
[target]
[target] Synchronizing Cookbooks:
[target]
[target] Installing Cookbook Gems:
[target]
[target] Compiling Cookbooks...
[target]
[target] [2019-12-10T15:16:56-05:00] WARN: Node target has an empty run list.
[target] Converging 0 resources
[target]
[target]
[target]
[target] Running handlers:
[target]
[target] Running handlers complete
[target] Chef Infra Client finished, 0/0 resources updated in 02 seconds
[target]
```

Now, if you check back in your browser and refresh Chef Manage, you should see that your target machine is now being managed by the server.

Chef Manage Displays Your Target Node



7.4.14

Chef Example – Putting it all Together

Simply read along with this example to better understand Chef.

Putting it all together

Now you will use everything together to create an actual recipe, push it to the server, and tell the target machine's client to requisition and converge on the new configuration. This is similar to the way Chef is used in production, but on a smaller scale.

To start, create a cookbook to build a simple website. Navigate to your cookbooks directory, create a new cookbook called `apache2`, and navigate into it:

```
cd /home/myname/cookbookschef generate cookbook apache2cd apache2
```

Take a look around the cookbook folder structure. There are folders already prepared for recipes and attributes. Add an optional directory and subdirectory for holding files your recipe needs:

```
mkdir -p files/default
cd files/default
```

Files in the /default subdirectory of a /files directory within a cookbook can be found by recipe name alone, no paths are required.

Now create a homepage for your website:

```
vi index.html
<html>
<head>
<title>Hello!</title>
</head>
<body>
<h1>HELLO!</h1>
</body>
</html>
```

Save the file and exit, then navigate to the recipes directory, where Chef has already created a `default.rb` file for us. The `default.rb` file will be executed by default when the recipe is run.

```
cd ../../recipes
```

Add some stanzas to the `default.rb` file. Again, edit the file:

```
vi default.rb
```

The header at the top is created for you.

```
#
# Cookbook:: apache2
# Recipe:: default
#
# Copyright:: 2019, The Authors, All Rights Reserved.
apt_update do
  action :update
end
package 'apache2' do
  action :install
end
cookbook_file "/var/www/html/index.html" do
  source "index.html"
  mode "0644"
end
```

Underneath, the recipe performs three actions. The first resource you are invoking, `apt_update`, handles the apt package manager on Debian. You would use this to force the equivalent of `sudo apt-get update` on your target server, before installing the Apache2 package. The `apt_update` resource's action parameter can take several other values, letting you perform updates only under controlled conditions, which you would specify elsewhere.

The package function is used to install the apache2 package from public repositories. Alternative actions include `:remove`, which would uninstall the package, if found.

Finally, you use the `cookbook_file` resource to copy the `index.html` file from `/files/default` into a directory on the target server (Apache's default web root directory). What actually happens is that the

cookbook, including this file, gets copied into a corresponding cookbook structure on the server, then served to the client, which executes the actions. The mode command performs the equivalent of `chmod 644` on the file which, when it reaches its destination, makes it universally readable and root-writable.

Save the `default.rb` file, then upload the cookbook to the server:

```
knife cookbook upload apache2
Uploading apache2      [0.1.0]
Uploaded 1 cookbook.
```

You can then confirm that the server is managing your target node:

```
knife node list
target
```

The Knife application can interoperate with your favorite editor. To enable this, perform the following export with your editor's name:

```
export EDITOR=vi
```

This lets the next command execute interactively, putting the node definition into vi to let you alter it manually.

```
knife node edit target{
  "name": "target",
  "chef_environment": "_default",
  "normal": {
    "tags": [
    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
  "recipe[apache2]"
]
}
```

As you can see, the expression `"recipe[apache2]"` has been added into the `run_list` array, which contains an ordered list of the recipes you want to apply to this node.

Save the file in the usual manner. Knife immediately pushes the change to the Infra Server, keeping everything in sync.

Finally, you can use the `knife ssh` command to identify the node, log into it non-interactively using SSH, and execute the `chef-client` application. This causes the node to immediately reload its state from the server (which has changed) and implement the new runlist on its host.

```
knife ssh 'name:target' 'sudo chef-client'
```

In this case, you would need to provide your sudo password for the target machine, when Knife asks for it. In a real production environment, you would automate this so you could update many nodes at once, storing secrets separately.

If all goes well, Knife gives you back a very long log that shows you exactly the content of the file that was overwritten (potentially enabling rollback), and confirms each step of the recipe as it executes.

```
target
target Starting Chef Infra Client, version 15.5.17
target resolving cookbooks for run list: ["apache2"]
target Synchronizing Cookbooks:
target   - apache2 (0.1.0)
target Installing Cookbook Gems:
target Compiling Cookbooks...
target Converging 3 resources
target Recipe: apache2::default
target   * apt_update[] action update
target     - force update new lists of packages
target     * directory[/var/lib/apt/periodic] action create (up to date)
target     * directory[/etc/apt/apt.conf.d] action create (up to date)
target     * file[/etc/apt/apt.conf.d/15update-stamp] action create_if_missing (up
to date)
target     * execute[apt-get -q update] action run
target       - execute ["apt-get", "-q", "update"]
target
target   * apt_package[apache2] action install
target     - install version 2.4.29-1ubuntu4.11 of package apache2
target   * cookbook_file[/var/www/html/index.html] action create
target     - update content in file /var/www/html/index.html from b66332 to 3137ae
target     --- /var/www/html/index.html 2019-12-10 16:48:41.039633762 -0500
target     +++ /var/www/html/.chef-index20191210-4245-1kusby3.html      2019-12-10
16:48:54.411858482 -0500
target     @@ -1,376 +1,10 @@
target     -
target     -<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
target     -<html xmlns="http://www.w3.org/1999/xhtml">
target     -  <!--
target     -    Modified from the Debian original for Ubuntu
target     -    Last updated: 2016-11-16
target     -    See: https://launchpad.net/bugs/1288690
target     -  -->
target     -  <head>
target     -    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
/>
target     -    <title>Apache2 Ubuntu Default Page: It works</title>
target     -    <style type="text/css" media="screen">
target     -  * {
target     -    margin: 0px 0px 0px 0px;
```

```
target     -    padding: 0px 0px 0px 0px;
target     -  }
target     -
target     -  body, html {
target     -    padding: 3px 3px 3px 3px;
target     -
target     -    background-color: #D8DBE2;
target     -
target     -    font-family: Verdana, sans-serif;
target     -    font-size: 11pt;
target     -    text-align: center;
target     -  }
target     -
target     -  div.main_page {
target     -    position: relative;
target     -    display: table;
target     -
target     -    width: 800px;
target     -
target     -    margin-bottom: 3px;
target     -    margin-left: auto;
target     -    margin-right: auto;
target     -    padding: 0px 0px 0px 0px;
target     -
target     -    border-width: 2px;
target     -    border-color: #212738;
target     -    border-style: solid;
target     -
target     -    background-color: #FFFFFF;
target     -
target     -    text-align: center;
target     -  }
target     -
target     -  div.page_header {
target     -    height: 99px;
target     -    width: 100%;
target     -
target     -    background-color: #F5F6F7;
target     -  }
target     -
target     -  div.page_header span {
target     -    margin: 15px 0px 0px 50px;
target     -
target     -    font-size: 180%;
target     -    font-weight: bold;
target     -  }
target     -
target     -  div.page_header img {
target     -    margin: 3px 0px 0px 40px;
target     -
target     -    border: 0px 0px 0px;
target     -  }
```

```
target     -
target     -  div.table_of_contents {
target     -    clear: left;
target     -
target     -    min-width: 200px;
target     -
target     -    margin: 3px 3px 3px 3px;
target     -
target     -    background-color: #FFFFFF;
target     -
target     -    text-align: left;
target     -  }
target     -
target     -  div.table_of_contents_item {
target     -    clear: left;
target     -
target     -    width: 100%;
target     -
target     -    margin: 4px 0px 0px 0px;
target     -
target     -    background-color: #FFFFFF;
target     -
target     -    color: #000000;
target     -    text-align: left;
target     -  }
target     -
target     -  div.table_of_contents_item a {
target     -    margin: 6px 0px 0px 6px;
target     -  }
target     -
target     -  div.content_section {
target     -    margin: 3px 3px 3px 3px;
target     -
target     -    background-color: #FFFFFF;
target     -
target     -    text-align: left;
target     -  }
target     -
target     -  div.content_section_text {
target     -    padding: 4px 8px 4px 8px;
target     -
target     -    color: #000000;
target     -    font-size: 100%;
target     -  }
target     -
target     -  div.content_section_text pre {
target     -    margin: 8px 0px 8px 0px;
target     -    padding: 8px 8px 8px 8px;
target     -
target     -    border-width: 1px;
target     -    border-style: dotted;
target     -    border-color: #000000;
```

```
target     -
target     -    background-color: #F5F6F7;
target     -
target     -    font-style: italic;
target     -  }
target     -
target     -  div.content_section_text p {
target     -    margin-bottom: 6px;
target     -  }
target     -
target     -  div.content_section_text ul, div.content_section_text li {
target     -    padding: 4px 8px 4px 16px;
target     -  }
target     -
target     -  div.section_header {
target     -    padding: 3px 6px 3px 6px;
target     -
target     -    background-color: #8E9CB2;
target     -
target     -    color: #FFFFFF;
target     -    font-weight: bold;
target     -    font-size: 112%;
target     -    text-align: center;
target     -  }
target     -
target     -  div.section_header_red {
target     -    background-color: #CD214F;
target     -  }
target     -
target     -  div.section_header_grey {
target     -    background-color: #9F9386;
target     -  }
target     -
target     -  .floating_element {
target     -    position: relative;
target     -    float: left;
target     -  }
target     -
target     -  div.table_of_contents_item a,
target     -  div.content_section_text a {
target     -    text-decoration: none;
target     -    font-weight: bold;
target     -  }
target     -
target     -  div.table_of_contents_item a:link,
target     -  div.table_of_contents_item a:visited,
target     -  div.table_of_contents_item a:active {
target     -    color: #000000;
target     -  }
target     -
target     -  div.table_of_contents_item a:hover {
target     -    background-color: #000000;
```

```
target     -
target     -    color: #FFFFFF;
target     -  }
target     -
target     -  div.content_section_text a:link,
target     -  div.content_section_text a:visited,
target     -   div.content_section_text a:active {
target     -    background-color: #DCDFE6;
target     -
target     -    color: #000000;
target     -  }
target     -
target     -  div.content_section_text a:hover {
target     -    background-color: #000000;
target     -
target     -    color: #DCDFE6;
target     -  }
target     -
target     -  div.validator {
target     -  }
target     -    </style>
target     -  </head>
target     -  <body>
target     -    <div class="main_page">
target     -      <div class="page_header floating_element">
target     -        <img src="/icons/ubuntu-logo.png" alt="Ubuntu Logo"
class="floating_element"/>
target     -        <span class="floating_element">
target     -          Apache2 Ubuntu Default Page
target     -        </span>
target     -      </div>
target     -<!--      <div class="table_of_contents floating_element">
target     -        <div class="section_header section_header_grey">
target     -          TABLE OF CONTENTS
target     -        </div>
target     -        <div class="table_of_contents_item floating_element">
target     -          <a href="#about">About</a>
target     -        </div>
target     -        <div class="table_of_contents_item floating_element">
target     -          <a href="#changes">Changes</a>
target     -        </div>
target     -        <div class="table_of_contents_item floating_element">
target     -          <a href="#scope">Scope</a>
target     -        </div>
target     -        <div class="table_of_contents_item floating_element">
target     -          <a href="#files">Config files</a>
target     -        </div>
target     -      </div>
target     --->
target     -      <div class="content_section floating_element">
target     -
target     -
```

```
target     -        <div class="section_header section_header_red">
target     -          <div id="about"></div>
target     -          It works!
target     -        </div>
target     -        <div class="content_section_text">
target     -          <p>
target     -                This is the default welcome page used to test the
correct
target     -                operation of the Apache2 server after installation on
Ubuntu systems.
target     -                It is based on the equivalent page on Debian, from
which the Ubuntu Apache
target     -                packaging is derived.
target     -                If you can read this page, it means that the Apache
HTTP server installed at
target     -                this site is working properly. You should <b>replace
this file</b> (located at
target     -                <tt>/var/www/html/index.html</tt>) before continuing to
operate your HTTP server.
target     -          </p>
target     -
target     -
target     -          <p>
target     -                If you are a normal user of this web site and don't
know what this page is
target     -                about, this probably means that the site is currently
unavailable due to
target     -                maintenance.
target     -                If the problem persists, please contact the site's
administrator.
target     -          </p>
target     -
target     -        </div>
target     -        <div class="section_header">
target     -          <div id="changes"></div>
target     -                Configuration Overview
target     -        </div>
target     -        <div class="content_section_text">
target     -          <p>
target     -                Ubuntu's Apache2 default configuration is different
from the
target     -                upstream default configuration, and split into several
files optimized for
target     -                interaction with Ubuntu tools. The configuration system
is
target     -                <b>fully documented in
target     -                /usr/share/doc/apache2/README.Debian.gz</b>. Refer to
this for the full
target     -                documentation. Documentation for the web server itself
can be
target     -                found by accessing the <a href="/manual">manual</a> if
the <tt>apache2-doc</tt>
```

```
target     -                package was installed on this server.
target     -
target     -          </p>
target     -          <p>
target     -                The configuration layout for an Apache2 web server
installation on Ubuntu systems is as follows:
target     -          </p>
target     -          <pre>
target     -/etc/apache2/
target     -|-- apache2.conf
target     -|       `--  ports.conf
target     -|-- mods-enabled
target     -|       |-- *.load
target     -|       `-- *.conf
target     -|-- conf-enabled
target     -|       `-- *.conf
target     -|-- sites-enabled
target     -|       `-- *.conf
target     -          </pre>
target     -          <ul>
target     -                          <li>
target     -                             <tt>apache2.conf</tt> is the main
configuration
target     -                             file. It puts the pieces together by
including all remaining configuration
target     -                             files when starting up the web server.
target     -                          </li>
target     -
target     -                          <li>
target     -                             <tt>ports.conf</tt> is always included from
the
target     -                             main configuration file. It is used to
determine the listening ports for
target     -                             incoming connections, and this file can be
customized anytime.
target     -                          </li>
target     -
target     -                          <li>
target     -                             Configuration files in the <tt>mods-
enabled/</tt>,
target     -                             <tt>conf-enabled/</tt> and <tt>sites-
enabled/</tt> directories contain
target     -                             particular configuration snippets which
manage modules, global configuration
target     -                             fragments, or virtual host configurations,
respectively.
target     -                          </li>
target     -
target     -                          <li>
target     -                             They are activated by symlinking available
target     -                             configuration files from their respective
target     -                             *-available/ counterparts. These should be
```

```
managed
target     -                           by using your helpers
target     -                           <tt>
target     -                                a2enmod,
target     -                                a2dismod,
target     -                           </tt>
target     -                           <tt>
target     -                                a2ensite,
target     -                                a2dissite,
target     -                            </tt>
target     -                                and
target     -                           <tt>
target     -                                a2enconf,
target     -                                a2disconf
target     -                           </tt>. See their respective man pages for
detailed information.
target     -                         </li>
target     -
target     -                         <li>
target     -                           The binary is called apache2. Due to the use
of
target     -                           environment variables, in the default
configuration, apache2 needs to be
target     -                           started/stopped with
<tt>/etc/init.d/apache2</tt> or <tt>apache2ctl</tt>.
target     -                           <b>Calling <tt>/usr/bin/apache2</tt>
directly will not work</b> with the
target     -                           default configuration.
target     -                         </li>
target     -          </ul>
target     -        </div>
target     -
target     -        <div class="section_header">
target     -            <div id="docroot"></div>
target     -                Document Roots
target     -        </div>
target     -
target     -        <div class="content_section_text">
target     -            <p>
target     -                By default, Ubuntu does not allow access through the
web browser to
target     -                <em>any</em> file apart of those located in
<tt>/var/www</tt>,
target     -                <a
href="http://httpd.apache.org/docs/2.4/mod/mod_userdir.html"
rel="nofollow">public_html</a>
target     -                directories (when enabled) and <tt>/usr/share</tt> (for
web
target     -                applications). If your site is using a web document
root
target     -                located elsewhere (such as in <tt>/srv</tt>) you may
need to whitelist your
```

```
target     -               document root directory in
<tt>/etc/apache2/apache2.conf</tt>.
target     -           </p>
target     -           <p>
target     -               The default Ubuntu document root is
<tt>/var/www/html</tt>. You
target     -               can make your own virtual hosts under /var/www. This is
different
target     -               to previous releases which provides better security out
of the box.
target     -           </p>
target     -       </div>
target     -
target     -       <div class="section_header">
target     -         <div id="bugs"></div>
target     -               Reporting Problems
target     -       </div>
target     -       <div class="content_section_text">
target     -         <p>
target     -               Please use the <tt>ubuntu-bug</tt> tool to report bugs
in the
target     -               Apache2 package with Ubuntu. However, check <a
target     -
href="https://bugs.launchpad.net/ubuntu/+source/apache2"
target     -               rel="nofollow">existing bug reports</a> before
reporting a new bug.
target     -         </p>
target     -         <p>
target     -               Please report bugs specific to modules (such as PHP and
others)
target     -               to respective packages, not to the web server itself.
target     -         </p>
target     -       </div>
target     -
target     -
target     -
target     -
target     -      </div>
target     -    </div>
target     -    <div class="validator">
target     -    </div>
target     -  </body>
target     +<html>
target     +<head>
target     +<title>Hello!</title>
target     +</head>
target     +<body>
target     +<h1>HELLO!</h1>
target     +</body>
target      </html>
target
target
```

```
target Running handlers:
target Running handlers complete
target Chef Infra Client finished, 4/7 resources updated in 02 minutes 31 seconds
```

At this point, you should be able to point a browser at the target machine's IP address, and see your new index page.

Chef will work to maintain this configuration, preventing drift in the configuration. If you were to log into the target server and make a change to the `index.html` file in `/var/www/html` (for example, changing the word "HELLO" to "GOODBYE"), Chef will fix the change pre-emptively the next time the agent runs (by default, within 30 minutes).

7.4.15

Summary of Automation Tools

Summary

This has been a high-level introduction to three modern DevOps toolkits. You should now be ready to:

- Deploy and integrate free versions of the major components of Ansible, Puppet, and/or Chef on a range of substrates, from desktop virtual machines (such as VirtualBox VMs) to cloud-based VMs on Azure, AWS, GCP or other IaaS platforms.
- Experience each platform's declarative language, style of infra-as-code building and organizing, and get a sense of the scope of its library of resources, plugins, and integrations.
- Get practice automating some of the common IT tasks you may do at work, or solve deployment and lifecycle management challenges you set yourself, in your home lab. Hands-on exercises and work will give you a complete sense of how each platform addresses configuration themes, and help you overcome everyday IT gotchas.

If you are building up your reputation with community, know that almost nothing impresses IT peers so much as a well-executed, insightfully automated deploy or manage codebase for a complex, head-scratching piece of software. Entire companies are built on the bedrock of knowing how to deploy complicated systems in robust configurations, such as specialized databases, container orchestration and cloud frameworks like Kubernetes and Openstack.

Be realistic, though. These are each extremely complex and sophisticated platforms that take users years to master, so do not get discouraged if you find them confusing! Reach out to the communities of the products you enjoy using (or that your workplace endorses) and you will learn more quickly.

7.3 Basic Automation Scripting

7.5 Infrastructure as Code