Sold to
alexclear@gmail.com

TASTE TEST

Puppet - Chef - SaltStack - Ansible



THIRD EDITION

Bonus Chapters
Communities
Security
Docker

MATT JAYNES

For my nephew, Austin

"Keep away from people who try to belittle your ambitions.
Small people always do that, but the really great make you feel that you, too, can become great."

- Mark Twain

Special thanks to everyone who took the time to review and help improve the book. You are good people.

Cover photo by Jamie McCaffrey

© 2015 Matt Jaynes

Table of Contents

Intro ..... 4.

Shell Script ..... 16.

Pre Tool Setup ..... 27.

Tool: Ansible ..... 32.

Tool: SaltStack ..... 51.

Tool: Chef ..... 70.

Tool: Puppet ..... 81.

Bonus: Where Docker Fits In ..... 93.

Bonus: CM Tool Security ..... 117.

Bonus: CM Tool Communities ..... 131.

Conclusion ..... 140.

© 2015 Matt Jaynes

Intro

You're reading this book because you'd like to get a quick taste of what it's like to work with the most popular Configuration Management (CM) tools like Puppet, Chef, Salt, and Ansible.

Using a CM tool is a huge win for your productivity, speed, and sanity. One client of mine saved over $2.7 million in the first year alone due to the productivity gains. It's generally the first thing I set up for new clients since it's such a huge win for them.

Exactly why is using a CM tool so powerful? Well, without one, your systems are destined for chaos. Chaos in your systems will leak slowness and misery throughout your company. It will start to feel like you've been cursed because everything becomes so difficult and bad luck seems to pervade everything. You'll start to wonder if you built your company on some ancient evil burial ground.

Real Life Example

I had a client a few years ago whose systems seemed like a bad horror movie.

No matter what, everything seemed to take several weeks.

Application deploy? Generally 3 weeks of arduous back and forth between developers, QA, and systems engineers.

Replace a broken server? 3 weeks.

Add a new server for scaling? 3 weeks.

It was the 3 week curse to do just about anything.

© 2015 Matt Jaynes

There were about 60 servers altogether and no two were exactly the same since they had all been set up by hand. Over time, the systems engineers would periodically update the servers, but they always seemed to miss some. Every server was a "beautiful snowflake" and unique from all the others.

The development environments were all slightly different too. So, an app release that worked on one developer's environment, wouldn't work on another developer's environment. And an app release that worked on all the developers' environments would have bugs in the staging environment. And an app release that worked in the staging environment would break in production. You get the idea.

Bugs in the production environment were especially hard to fix. Often the bug would occur on only 1 of the ~40 load-balanced app servers, so just trying to reproduce the bug was hugely time consuming.

They couldn't scale.

Customers were leaving.

Morale was low and the best people on the teams were jumping ship to other companies.

No one really understood the systems - even the systems engineers.

They were at risk of catastrophic systems failure that could kill the company.

Can you guess what saved the day?

That's right, a CM tool - Puppet in this case.

I spent about 6 weeks figuring out and scripting all the systems in Puppet. I worked closely with the developers and the QA team to ensure we got it right.

© 2015 Matt Jaynes

Once it was done, it was done.

The curse had been lifted.

New servers could be launched in under 5 minutes.

The app could be deployed in under 10 minutes.

What used to take weeks now took less time than a cup of coffee.

Morale, speed, stability, and everything good and wonderful seemed to bloom.

If you ever dreamed of a magic wand to miraculously make your company faster and more profitable, using a CM tool comes pretty close.

Goal

This isn't a deep exploration of these tools. Instead, I aim to give you a great head start by saving you the weeks of research you might have spent trying out the tools in order to choose one.

If you can quickly choose a CM tool, then you can get on with the business of making your systems more awesome.

Friendly wine and cheese event

Before we start, I'd just like to be clear about what this book *is* and what it *is not*.

This is not a bar brawl where we pit the CM tools against each other - it's more like a wine tasting.

© 2015 Matt Jaynes

The people that have built these tools I've found to be universally excellent, generous, kind people. They've helped create a wonderful DevOps community that is good-humored and generally free of the negativity you might see in other tech communities.

It's very rare to hear one of the leaders behind these tools disparage another tool's team. In fact, I can think of multiple instances where I've heard one of them take a stand and defend the other team.

That's not to say that they aren't true competitors. Each CM tool has a venture-backed company behind it. They absolutely are competing - but it's very much friends competing with friends rather than some kind of bitter war.

There's a lot of wisdom in this approach. What is their actual #1 competitor? **No tool at all**.

I'm continually surprised at the number of companies that don't use a CM tool. I don't think I'd be far off in estimating that over 80% of companies are still in the stone age of manually installing their servers or using some terrible cobbled together set of shell scripts. I see this *constantly* in the wild and it seems to affect every size of company from tiny startups to billion dollar corporations.

Keeping a great vibe in the DevOps community is essential to attracting people from the biggest competitor: **Using No CM Tool At All**.

Sample Project

In this book, I walk you through an identical sample project with each of the four CM tools. In writing the book, Ansible took the least time to set up the project (~2 hours). Salt has a higher learning curve and took a bit longer (~5 hours). Puppet had a few rough patches and took ~9 hours. Chef was the toughest and took ~12 hours.

© 2015 Matt Jaynes

Don't worry, it won't take you anywhere near that long now that you have this book. You should be able to set up each CM tool with the sample project in under 30 minutes each.

Why did Puppet and Chef take so long even though they were the two tools I had previous experience with? Well, I forced myself not to use any of my notes or past projects as reference - I only used the official documentation and whatever I could find via Google. But, ultimately it was outdated documentation, confusing flows, and inconsistencies that hindered both Puppet and Chef. In updating this book for the 3rd Edition, I noticed that both of them have improved their documentation, but it is still a rough experience trying to get started with them compared to Ansible and Salt.

A note on terminology

Each CM tool uses different terminology, so to avoid confusion I'm going to use a consistent terminology throughout the book. In the individual CM tool chapters, I'll mention the relevant unique terminology the tool uses.

Here are the terms I'll use:

"Directive"

The command a CM tool uses to tell a server to do something. For example, a directive might be `ensure user 'matt' exists`.

"Directives Script"

A script that includes multiple directives.

© 2015 Matt Jaynes

"Master Node"

The server that hosts the directive scripts used to define the children nodes. I'll also refer to it as the master server.

"Children Nodes"

The servers that get their directives from the master node. I'll also refer to these as children servers.

Fundamental Differences

There are a few differences that deserve covering before we get started.

Directive Ordering

Imagine if in programming, the lines in your code were run in random order instead of sequentially from top to bottom. Sounds crazy right? Well, in the past Puppet and Salt essentially did this and required you to explicitly declare the order and dependencies of your directives. Both tools argued that by doing this, it made things more "powerful", but in practice I never saw it be anything more than a big confusing headache. Fortunately, Puppet (version 3.3.0 and higher) and Salt (version 0.17 and higher) have seen the light and now run their directives in sequential order as you would expect. This has always been the case for Ansible and Chef. I only mention ordering here since you may come across older documentation and blog posts about Salt and Puppet that discuss their old non-sequential run ordering of directives.

© 2015 Matt Jaynes

Directives Language

Ansible and Salt use the standard YAML format and the simple Jinja2 templating language (which really is super simple, despite the ominous-sounding name). Both are easy to learn. They make Ansible and Salt highly accessible for developers of all languages.

Chef uses the Ruby programming language with an extended DSL (domain specific language). While this is very powerful and convenient for Rubyists, it makes things a little more challenging for non-Ruby developers. Fortunately, Ruby is a simple elegant language that is easy to learn.

Puppet uses its own custom configuration language. It's not difficult, but does add to the learning curve.

Master / Children nodes setup

Ansible

Ansible has the simplest setup and uses SSH to connect the children nodes. You only install Ansible on your master node (which can just be your laptop since Ansible just uses SSH to push the directive commands out to the children). There's no special client that needs to be installed on the children nodes. You usually already have SSH access to your servers, so Ansible piggybacks on that, which makes its setup super simple.

Salt

By default, Salt uses a Master / Children nodes setup. This requires installing a special service on the master node and also a special client on each child node. Each child node gets the directives from the master node via a high-speed communication bus and then the client runs the directive commands.

© 2015 Matt Jaynes

Salt also has an SSH push mode similar to Ansible's called `salt-ssh`, so you also have the option of running Salt without having to install a special client on the children nodes.

Chef

Chef uses a fairly standard Master / Children nodes setup, but also adds the concept of a workstation node which interacts with the Master node. The workstation node is generally your local machine like your laptop or desktop.

Chef has the most challenging setup of all the tools.

Chef Software, Inc. sells a hosted master server solution which is free for 5 children nodes or less. Then the price is $120 for 20 children, $300 for 50 children, and $700 for 100 children (as of March 2015). I fully support Chef Software, Inc. making money on this, but their documentation is nearly all geared to using their hosted solution, which makes it unnecessarily difficult to set up your own Chef master node. Due to security concerns, many companies will not want to use a hosted master node service, so having good documentation is essential but lacking.

The Chef master node also requires a good deal of RAM (4GB!) in order to be installed and run properly. When I tried to set it up on a server with less RAM, I got an error that mentioned nothing about memory issues, so it was very difficult to debug. When I increased the RAM on the server, the seemingly unrelated error went away.

As mentioned, Chef also requires a workstation node. Setting up the workstation is non-trivial and can be a pain especially if you have to set it up for multiple engineers. Since you're not running the commands from the master node directly, it adds to the number of steps you need to perform in your regular workflow to run directive scripts against your children nodes.

© 2015 Matt Jaynes

Because of these extra challenges in setting up Chef, many companies choose to bypass using a master and workstation node and just distribute their directive scripts to the children via some other means (capistrano, git, etc). They then run Chef in isolation on each child node. This is generally referred to as a "Chef Solo" setup (though more recently they've also added a similar "Chef Zero" setup).

Puppet

Puppet has a standard Master / Children nodes setup. Like chef, the master node requires a lot of RAM (4GB!). Puppet requires installing a special client on each child node. Each child node pulls the directives from the master node and then runs the directive commands.

Scalability

All of these CM tools can scale to over 10,000 nodes. Each tool needs to be configured a bit differently to handle extremely large scales. We won't be covering high scale scenarios in this book, but the scalability of these tools isn't much of a factor for most production systems.

Windows Support

Puppet, Chef, and Salt support Microsoft Windows. Ansible has recently added Windows support and is actively growing that functionality. Since managing Windows servers is rarer these days, we won't be discussing it in this book. I'll be reviewing these tools from the perspective that they'll only be used on Unix/Linux and similar operating systems.

© 2015 Matt Jaynes

Remote execution

Remote execution is the ability to run commands against your children nodes. For example, if you wanted to find out the date/time setting for each child node, you would want a way to execute a command like `date` on all of your children nodes and receive the output without logging into each of them individually. Tools for this include Fabric, Capistrano, and Func.

Ansible and **Salt** have robust, easy-to-use remote execution built-in and immediately available after installation.

Chef has a tool called 'knife' that is used for many purposes including remote execution, but it can be challenging to configure and feels clunky compared to Ansible/Salt.

Puppet doesn't have an included tool for this, but suggests using 'mcollective' which can be difficult to install, configure, and learn.

Up next

So, let's get to it. You want to see what these tools are like in action and I want to show you.

We'll go through a super simple multi-server project that allows me to demonstrate some of the key features of the tools.

Since we set up an identical system with each CM tool, it will give you a good taste of how each tool handles the job.

I'll take you step-by-step through the exact commands and directives to implement the project. That way you can follow along and get a sense of how each tool works.

© 2015 Matt Jaynes

This book is a quick short-cut to experiencing these tools. Rather than trudging through documentation and going down false paths, I've already done all that grunt work for you. I'll show you the easy way through and will warn you about the big rough spots so you can avoid them.

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Shell Script

Wait, why am I starting with a shell script?!

Well, before we dive into the CM tools it's important to show you how the example system would be set up manually. Then you'll have a clear idea of what parts the system has and what the tools do when they perform the setup for us.

The obvious limitations

Just having a shell script for server setups is more than a lot of systems have. Still, it has some big limitations.

Can't run multiple times safely

A shell script works fine for the initial setup, but is pretty useless for ongoing maintenance and system changes. It generally can only be run once safely, then it's a liability and only good for documenting what happened once-upon-a-time on the system.

Using a shell script to set up a server generally indicates that it will then be managed manually afterwards (which leads to sadness and despair!).

A huge advantage of using a CM tool is that they're "idempotent". Idempotency basically means that you can run the directives over and over again safely.

An idempotent command will verify that the system is how you defined it and will only make changes to bring the system back into alignment with what you defined. That means you can define your system in the language of the CM tool and use it not only for initial system setup, but also for monitoring, updating, and correcting a server's configuration over the life of the server.

© 2015 Matt Jaynes

The CM tool can ultimately act like a self-healing test suite for your systems - neat!

Disparate command interfaces

Each system command (`ls`, `cat`, `tail`, etc) has a different group of developers behind it. That leads to slightly different interfaces for using each command. There are conventions that are generally followed, but the commands still differ from each other in subtle ways and sometimes differ depending on which Unix/Linux/BSD distribution they're on (ex: usage for the `ps` command on OS X is different from `ps` on Red Hat).

Another advantage of CM tools is that they abstract away some of these differences and give you a more unified interface for interacting with your system. For example, rather than looking up whether your system uses `groupadd` or the `addgroup` command, you can just use the standardized CM tool command to create the group. This command/directive happens to simply be called `group` in all of the four CM tools we're covering.

Similarly, CM tools standardize the reporting output from the different commands it runs.

Scenarios

I want to show you how the CM tools work for some typical scenarios. In order to do that quickly, I've created a fairly arbitrary system that isn't very realistic, but will give you a good sense of how each tool presents some key features.

The main scenarios I'll be covering are:

- **master/children node setup**
- **installing packages**

© 2015 Matt Jaynes

- **user/group setup**
- **deploying a static file**
- **deploying a templated file**
- **running a service**

In order to show those, I'll be setting up two simple websites.

Why two?

Well, there are several basic features I want to highlight which require more than one.

Frivolous story

Need a story for why this system exists?

Well, in 1965 a secret underground cult dedicated to puppy worship started an invisible war with a secret kitten cult. That war has raged for decades now and many strongly-worded memos have been exchanged between the cults.

In the efforts of peace and more civil memos, you've devised a way to appease the cults. You've observed both groups and they both desperately want a browser home page that presents a simple idyllic picture of their favorite baby animal.

So, you've searched the Creative Commons images for suitable puppy and kitten photos and come up with these:

© 2015 Matt Jaynes





Now, the cults are very sensitive about where the electrons serving the photo come from, so the puppy and kitten images must never be served from the same server, lest their electrons be contaminated by the other baby animal photo!

They also insist that the user/group that owns the puppy/kitty image be named 'puppy' or 'kitty' respectively.

Yes, it makes no sense - but what cult was ever very reasonable? ;-)

Launch servers

First we'll launch a puppy and a kitty server on Digital Ocean and use Ubuntu 14.04 x64 as the OS for both of them.

Note:
You don't have to use Digital Ocean, but each server is less than $0.01 per hour, so for each demo of a CM tool, you'll spend less than 5¢. Just remember to destroy your servers (or "droplets" as Digital Ocean calls them) when you're done with them so you don't get charged while they're idle.

I've personally tested the walkthrough for the shell script and each of the CM tools with this

© 2015 Matt Jaynes

setup, so for the smoothest experience, I recommend following the exact setup I used.

If you are already an experienced Vagrant user, then it should be pretty straight-forward to set this up for the walkthroughs. If you don't have experience with Vagrant yet, I recommend finishing this book first with the recommended Digital Ocean setup, then tackling Vagrant as a separate learning project. There's a learning curve and several very large downloads involved, so you don't want to get distracted with that right now.

Set their hostnames as `puppy.dev` and `kitty.dev`, then when the server is created and you get their IP addresses, add them to your `/etc/hosts` file like this (replacing the IPs below with the actual IPs Digital Ocean creates):

```
999.999.999.2  puppy.dev
999.999.999.3  kitty.dev
```

Let's build this thing!

First, let's build the puppy server, then when we've finished `puppy.dev` we'll build `kitty.dev` with the respective commands.

The steps will be pretty simple:

- **install nginx**
- **add the photo**
- **create user/group**
- **change photo's ownership/permissions**
- **add the html page**
- **run nginx**

© 2015 Matt Jaynes

Install nginx

Since we're doing this on the Ubuntu linux distribution we'll use `apt-get` to install nginx.

First we'll update the package lists so we get the most up-to-date packages:

```
root@puppy:~# apt-get update
```

Then we'll install the nginx package:

```
root@puppy:~# apt-get install nginx --assume-yes
```

Great, that was easy. We use the `--assumeyes` (same as `-y`) to avoid having to answer the "Are you sure?" type of prompts.

The web root for nginx is now at `/usr/share/nginx/html`, so that's where we'll be putting the puppy photo.

Add the photo

I've already created a Github repository with the resources for this project, including the images, so let's just download the image to the web server document root.

```
root@puppy:~# wget https://raw.github.com/nanobeep/tt/master/puppy.jpg \
>                 --output-document=/usr/share/nginx/html/puppy.jpg
```

© 2015 Matt Jaynes

Create user/group

Now we'll create the user/group:

```
root@puppy:~# useradd --user-group puppy
```

The `--user-group` flag tells `useradd` to also create a 'puppy' group and add the newly created puppy user to it.

Change photo's ownership and permissions

```
root@puppy:~# chown puppy:puppy /usr/share/nginx/html/puppy.jpg
root@puppy:~# chmod 664 /usr/share/nginx/html/puppy.jpg
```

Add the html page

The html page is super simple and looks like this:

```
<html>
    <body bgcolor="gray">
        <center>
            <img src="/baby.jpg">
        </center>
    </body>
</html>
```

Let's download it to the right place:

```
root@puppy:~# wget https://raw.github.com/nanobeep/tt/master/index.html \
>                  --output-document=/usr/share/nginx/html/index.html
```

© 2015 Matt Jaynes

To have the right image source, we'll just replace 'baby' with puppy:

```
root@puppy:~# sed --in-place 's/baby/puppy/g' /usr/share/nginx/html/index.html
```

You can probably guess that this is the page we'll be demonstrating CM tool templating with later :)

Run nginx

Now all we have to do is run the web server and we should be done.

Ubuntu keeps its service management scripts in `/etc/init.d/`, so let's look there:

```
root@puppy:~# ls -l /etc/init.d/ | grep nginx
-rwxr-xr-x 1 root root 2235 Jul 12  2012 /etc/init.d/nginx
```

Great, it's there and ready, so let's start it:

```
root@puppy:~# /etc/init.d/nginx start
Starting nginx: nginx.
```

Verify

Now, we can verify everything works by checking in the browser:

http://puppy.dev/

© 2015 Matt Jaynes

Putting it all together

So, our shell script for setting up a puppy server looks like this:

```
apt-get update
apt-get install nginx --assume-yes
wget https://raw.github.com/nanobeep/tt/master/puppy.jpg \
     --output-document=/usr/share/nginx/html/puppy.jpg
useradd --user-group puppy
chmod 664 /usr/share/nginx/html/puppy.jpg
chown puppy:puppy /usr/share/nginx/html/puppy.jpg
wget https://raw.github.com/nanobeep/tt/master/index.html \
     --output-document=/usr/share/nginx/html/index.html
sed --in-place 's/baby/puppy/' /usr/share/nginx/html/index.html
/etc/init.d/nginx start
```

Kitty

So, now to set up the kitty server, we'll just make a few substitutions:

```
apt-get update
apt-get install nginx --assume-yes
wget https://raw.github.com/nanobeep/tt/master/kitty.jpg \
     --output-document=/usr/share/nginx/html/kitty.jpg
useradd --user-group kitty
chmod 664 /usr/share/nginx/html/kitty.jpg
chown kitty:kitty /usr/share/nginx/html/kitty.jpg
wget https://raw.github.com/nanobeep/tt/master/index.html \
     --output-document=/usr/share/nginx/html/index.html
sed --in-place 's/baby/kitty/' /usr/share/nginx/html/index.html
/etc/init.d/nginx start
```

Run that on the kitty server, then verify that it worked:

http://kitty.dev/

© 2015 Matt Jaynes

Up next

Now you're familiar with the sample system we'll be using. The CM tool setups you're about to see will make a lot more sense now that you've seen exactly how the systems would be created by hand.

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Pre Tool Setup

Before each CM tool setup, I'll be launching 3 fresh new servers on Digital Ocean and they will all use Ubuntu 14.04 x64:

- **master server**
- **puppy node**
- **kitten node**

Chef is an exception since it also requires a machine to interact with the master. We'll be setting up an additional server as a 'workstation' for that purpose with Chef.

Also, for most of the servers, you can use the 512MB RAM 'droplet'. However, for both the Puppet and Chef master servers, you'll need at least 4GB of RAM.

Note:
If you've never used Digital Ocean before, don't be intimidated. Setting up an account is extremely easy. The servers we're using cost about a US penny per hour and they accept PayPal and other standard forms of payment.

Just remember to destroy your servers when you're done with them so you aren't charged for them when they're idle.

Again, if you are already an experienced Vagrant user and and want to use Vagrant, you can, but if Vagrant is new to you, just use Digital Ocean for now.

© 2015 Matt Jaynes

Networking

Since we're just doing demos and want to keep the setup as minimal as possible, we'll just use `/etc/hosts` to set the DNS on each of the servers, like so:

```
999.999.999.1  master.dev
999.999.999.2  puppy.dev
999.999.999.3  kitty.dev
```

I suggest also putting the same entries in your local `/etc/hosts` for convenience. Of course, replace the example 999.999.999.* IPs in the example with your servers' actual IP addresses.

Then you'll be able to connect to your servers via:

```
> ssh root@master.dev
> ssh root@puppy.dev
> ssh root@kitty.dev
```

Note:
If you're not on Linux or Mac OSX, then your hosts file may be in a different location which you can find here: http://en.wikipedia.org/wiki/Hosts_(file)

Remember that if you will use the same hostnames like I am for the different CM tool server scenarios, then you'll want to delete the server entries from your `~/.ssh/known_hosts` file so you don't get warnings when trying to log into the servers.

© 2015 Matt Jaynes

Each CM tool requires certain ports to be open. I've already verified that all ports are open on the Digital Ocean Ubuntu servers with:

```
> iptables --list --numeric
```

Use the `--numeric` flag since you just want the IP addresses and don't want to do hostname resolution for the IP's.

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

That output shows that all the ports are open.

sudo / root

Because these are throw-away servers, we'll just be running everything as `root`. When we come across instructions in the CM tool docs that suggest using `sudo`, we'll just silently drop the `sudo` for the commands we run.

Naturally, in production you should use a more secure setup (like `sudo` with limited-privilege users).

© 2015 Matt Jaynes

Important reminder

Now we're ready to cover the CM tools!

Just remember that when you destroy and rebuild your servers in order to run the different CM tools that you'll need to:

1. **clear the relevant entries in your `~/.ssh/known_hosts` file**
2. **update your local `/etc/hosts` file with the new IP addresses**
3. **update the `/etc/hosts` file on each server with the new IP addresses**

If you only do a "rebuild" on your server rather than a hard destroy, then the IP address will be the same and you can skip steps #2 and #3.

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Ansible

Overview

It's simple, has a very low learning curve, and is just pleasant and intuitive to use.

You don't have to install a client or anything special on the children nodes (assuming they have Python 2.6 or greater installed - which most popular Linux distros do), so you can be up and running nearly instantly. This is particularly useful if you already have legacy systems in place - you don't have to mess with installing and managing a special child-client service on your existing servers.

Documentation

http://docs.ansible.com/

Directives Execution Order

Ansible executes the directives in the order you'd expect: sequentially as they're written in your directives script.

Directives Language

YAML and Jinja2. Both are very simple and easy to learn. This makes Ansible very accessible for developers of all languages.

© 2015 Matt Jaynes

Remote Execution / Orchestration

Remote execution / orchestration functionality is built-in and available immediately - again, without even having to install anything on the children nodes.

Terminology

Directives = Tasks

Directives Script = Playbook

Master Node = Control Machine

Children Nodes = Hosts

Setup

Make sure you first set up your servers according to the instructions in the Setup chapter.

Install Ansible on master node

Let's get started with the walk-through by setting up the master node:

```
root@master:~# apt-get update
root@master:~# apt-get install software-properties-common -y
root@master:~# add-apt-repository ppa:ansible/ansible -y
root@master:~# apt-get update
root@master:~# apt-get install ansible -y
```

We run the `apt-get update` twice since we need to get the updated package lists first in order to install `software-properties-common` and then again to update the package lists for the `ansible/ansible` repository we added.

© 2015 Matt Jaynes

The installation documentation is at:

http://docs.ansible.com/ansible/intro_installation.html

Tell the master node about the children

Ansible requires a hosts inventory file to be set. This file specifies the children nodes to connect with. Optionally, you can also specify connection options for the servers and group them logically if needed. Grouping the servers allows you to target specific groups of servers as you'll soon see. The hosts inventory file is in the INI format.

So, for this project let's do a 'puppy' group and a 'kitty' group to allow us to target them easily.

Create `/root/inventory.ini` with this content:

```
[puppy]
puppy.dev

[kitty]
kitty.dev
```

The inventory documentation is at:

http://docs.ansible.com/ansible/intro_inventory.html

Setup connectivity to the children

Kudos:
Usually you will already have access to your servers via a method like ssh keys, so you often won't even need this step. If you're working on legacy systems this is especially great since you can be up and running without having to install anything on the children nodes.

© 2015 Matt Jaynes

Now, before we start working with the children nodes, we want to be able to connect to them without having to enter in a password every time. So let's use ssh keys to connect to the children nodes.

Generate the ssh key on the master node:

```
root@master:~# ssh-keygen -t rsa -C "matt@nanobeep.com" -N "" -f /root/.ssh/id_rsa
```

Note: *Unfortunately,* `ssh-keygen` *doesn't have verbose flags (like* `--long-flag-name`*), so here's a key to the flags we used:*

`-t` *specifies the encryption type to use.*
`-C` *specifies the comment, which is typically your email address.*
`-N` *specifies the passphrase to use. Never use an empty passphrase for production keys.*
`-f` *specifies the location to save the generated keys. This also generates the public key* `id_rsa.pub` *in the same directory.*

Now let's view the content of our new public key (`/root/.ssh/id_rsa.pub`) so we can put it on the children nodes:

```
root@master:~# cat /root/.ssh/id_rsa.pub
```

Copy the contents of `id_rsa.pub` from the master server and then paste it into `/root/.ssh/authorized_keys` on both the puppy.dev and kitty.dev servers.

© 2015 Matt Jaynes

Now you can test the connectivity:

```
root@master:~# ansible all --module-name ping --inventory-file=/root/inventory.ini
kitty.dev | success >> {
    "changed": false,
    "ping": "pong"
}

puppy.dev | success >> {
    "changed": false,
    "ping": "pong"
}
```

Success!

Let's look at the parts of that command in more detail:

`ansible all` runs Ansible against "all" of the children nodes (as opposed to a subgroup of them).

`--module-name ping` runs the `ping` module on the children nodes for a quick ping/pong connectivity check. Often `--module-name` is abbreviated to just `-m`.

`--inventory-file=/root/inventory.ini` specifies the file where the list of children nodes is defined. Often `--inventory-file` is abbreviated to just `-i`.

Warning:
If you don't set up ssh keys, but still try to connect to the children, you'll probably get an error like:

```
root@master:~# ansible all --module-name ping --inventory-file=/root/inventory.ini
puppy.dev | FAILED => SSH encountered an unknown error during the connection. We reco
mmend you re-run the command using -vvvv, which will enable SSH debugging output to h
elp diagnose the issue
```

© 2015 Matt Jaynes

This error occurs since (without ssh keys) you need to install the `sshpass` *package and then add the* `--ask-pass` *flag to the ansible command:*

```
root@master:~# apt-get install sshpass -y
root@master:~# ansible all --module-name ping --inventory-file=/root/inventory.ini
--ask-pass
```

If you still have problems, then follow the error message's advice to add `-vvvv` *to the end of the command so you will get the verbose connection debugging output.*

Configuration

We don't want to have to specify the inventory file every time, so let's add that as a setting in Ansible's configuration.

Create `/root/ansible.cfg` (INI formatted) with the setting to define the location of the inventory file:

```
[defaults]
hostfile = /root/inventory.ini
```

Now we can save a little typing:

```
root@master:~# ansible all -m ping
kitty.dev | success >> {
    "changed": false,
    "ping": "pong"
}

puppy.dev | success >> {
    "changed": false,
    "ping": "pong"
}
```

© 2015 Matt Jaynes

The configuration documentation is at:

http://docs.ansible.com/ansible/intro_configuration.html

Note:
Ansible has default locations where it automatically looks for the inventory and configuration files.

Had we just put the inventory file in the default `/hosts` *location, then we never would have needed to specify* `--inventory-file=/root/inventory.ini`*. However, we set it in a custom location so I could show you how to set it in the configuration file.*

Ansible automatically picks up the configuration data in this order:
`ANSIBLE_CONFIG` *(an environment variable)*
`ansible.cfg` *(in the current directory)*
`.ansible.cfg` *(in the home directory)*
`/etc/ansible/ansible.cfg`

Remote execution

Ansible gives you remote execution capabilities right out of the box. Here's a quick example:

```
root@master:~# ansible all -m command -a "date"
kitty.dev | success | rc=0 >>
Wed Jul 29 23:13:19 EDT 2015

puppy.dev | success | rc=0 >>
Wed Jul 29 23:13:18 EDT 2015
```

© 2015 Matt Jaynes

If I wanted to run the command on just the 'puppy' group of servers, you'd replace `all` with the `puppy` group:

```
root@master:~# ansible puppy -m command -a "date"
puppy.dev | success | rc=0 >>
Wed Jul 29 23:14:06 EDT 2015
```

Note:
Targeting server groups comes in handy for real-life scenarios, since you'll often want to group and target your servers by their function (webserver, db, cache, etc):

```
[webservers]
web1.example.org
web2.example.org

[db]
db.example.org

[cache]
cache.example.org
```

Then you can target them by group:

```
root@master:~# ansible webservers -m command -a "date"
```

© 2015 Matt Jaynes

Another great feature is the documentation via the command line:

```
root@master:~# ansible-doc --list
accelerate           Enable accelerated mode on remote node
acl                  Sets and retrieves file ACL information.
add_host             add a host (and alternatively a group) to the ansible-playbo
airbrake_deployment  Notify airbrake about app deployments
apt                  Manages apt-packages
apt_key              Add or remove an apt key
...output truncated...
```

```
root@master:~# ansible-doc apt
> APT

  Manages `apt' packages (such as for Debian/Ubuntu).

Options (= is mandatory):

- cache_valid_time
        If `update_cache' is specified and the last run is less or
        equal than `cache_valid_time' seconds ago, the `update_cache'
        gets skipped.
...output truncated...
```

Setting up the directives

Now we're ready to set up the directives to install everything.

You'll notice that a directive is just a module that is passed the parameters you specify.

The documentation for Ansible directive modules is at:

http://docs.ansible.com/ansible/modules.html

© 2015 Matt Jaynes

nginx package

We know we'll need to put our image and html files in the nginx web root directory, so let's install nginx first.

First, we'll create the directives script called `taste.yml` in `/root` and add the nginx directive:

```
---
- hosts: all
  tasks:
  - name: ensure nginx is installed
    apt: pkg=nginx state=present update_cache=yes
```

We want nginx to be installed on all the children nodes so we set the `hosts` to `all`. (The earlier groupings we did in the inventory file, like 'puppy' and 'kitty' could be used here as well if we wanted to target a specific group of servers.)

The `tasks` section is where we put our directives for this set of hosts.

The `name` can be any text that is helpful for you to remember what the directive does.

The `apt` line is the actual directive (module + parameters) that will be run.

You can see that we're just using the 'aptitude' package manager module (`apt`) to ensure nginx is installed. We add the `update_cache=yes` parameter so that `apt-get update` is performed before nginx is installed.

© 2015 Matt Jaynes

First run

Let's run this against the children nodes now:

```
root@master:~# ansible-playbook taste.yml

PLAY [all] ********************************************************************

GATHERING FACTS ***************************************************************
ok: [kitty.dev]
ok: [puppy.dev]

TASK: [ensure nginx is installed] *********************************************
changed: [kitty.dev]
changed: [puppy.dev]

PLAY RECAP ********************************************************************
kitty.dev                  : ok=2    changed=1    unreachable=0    failed=0
puppy.dev                  : ok=2    changed=1    unreachable=0    failed=0
```

Nice - it installed smoothly.

Image files

Now, let's set up the image files.

Download the image files to the master node in `/root`

```
root@master:~# wget https://raw.github.com/nanobeep/tt/master/puppy.jpg
root@master:~# wget https://raw.github.com/nanobeep/tt/master/kitty.jpg
```

© 2015 Matt Jaynes

Now, update the `taste.yml` file to include the image directives:

```
---
- hosts: all
  tasks:
  - name: ensure nginx is installed
    apt: pkg=nginx state=present update_cache=yes


- hosts: puppy
  tasks:
  - name: ensure puppy.jpg is present
    copy: src=/root/puppy.jpg dest=/usr/share/nginx/html/puppy.jpg


- hosts: kitty
  tasks:
  - name: ensure kitty.jpg is present
    copy: src=/root/kitty.jpg dest=/usr/share/nginx/html/kitty.jpg
```

You can see now we're using `hosts` to target which servers the directives get run on. You'll recall we added a puppy and a kitty group in the `/root/inventory.ini` file earlier which allows us to do this.

Note:
Instead of downloading the images and using the `copy` *directive, we could have used the* `get_url` *directive.*

Let's run the new directives:

```
root@master:~# ansible-playbook taste.yml

PLAY [all] ********************************************************************

GATHERING FACTS ***************************************************************
ok: [kitty.dev]
ok: [puppy.dev]
```

```
TASK: [ensure nginx is installed] *********************************************
ok: [kitty.dev]
ok: [puppy.dev]

PLAY [puppy] ******************************************************************

TASK: [ensure puppy.jpg is present] *******************************************
changed: [puppy.dev]

PLAY [kitty] ******************************************************************

TASK: [ensure kitty.jpg is present] *******************************************
changed: [kitty.dev]

PLAY RECAP ********************************************************************
kitty.dev                  : ok=3    changed=1    unreachable=0    failed=0
puppy.dev                  : ok=3    changed=1    unreachable=0    failed=0
```

You can see from the output that Ansible put the images on the correct nodes.

User / Group and ownerships

Now, we need to create the puppy/kitty groups and users so we can update the image file ownerships.

Remember that Ansible runs the directives sequentially, so we'll have to put the directives in this order: Create group > Create user > Change file ownership.

```
---
- hosts: all
  tasks:
  - name: ensure nginx is installed
    apt: pkg=nginx state=present update_cache=yes

- hosts: puppy
  tasks:
  - name: ensure puppy group is present
```

```
    group: name=puppy state=present
  - name: ensure puppy user is present
    user: name=puppy state=present group=puppy
  - name: ensure puppy.jpg is present
    copy: src=/root/puppy.jpg dest=/usr/share/nginx/html/puppy.jpg
              owner=puppy group=puppy mode=664

- hosts: kitty
  tasks:
  - name: ensure kitty group is present
    group: name=kitty state=present
  - name: ensure kitty user is present
    user: name=kitty state=present group=kitty
  - name: ensure kitty.jpg is present
    copy: src=/root/kitty.jpg dest=/usr/share/nginx/html/kitty.jpg
              owner=kitty group=kitty mode=664
```

We can specify the file ownership with our existing `copy` directive, so we've just used that instead of using a separate module like `file`.

Now run the new directives:

```
root@master:~# ansible-playbook taste.yml
...output omitted...
```

HTML template

Now, we'll make the html template with the Jinja2 templating language.

Create the html template as `index.j2` in `/root` and add these contents:

```
<html>
  <body bgcolor="gray">
    <center>
      <img src="/{{baby}}.jpg">
    </center>
  </body>
</html>
```

© 2015 Matt Jaynes

You'll notice the 'baby' variable that I set in the Jinja2 syntax with double curly brackets. That variable will be parsed when the template is processed.

We'll declare that variable in our directives script like this:

```
- hosts: puppy
  vars:
    baby: puppy
  tasks:
      ...
```

Now let's add the directive for the template.

The full `taste.yml` now looks like:

```
---
- hosts: all
  tasks:
  - name: ensure nginx is installed
    apt: pkg=nginx state=present update_cache=yes

- hosts: puppy
  vars:
    baby: puppy
  tasks:
  - name: ensure puppy group is present
    group: name=puppy state=present
  - name: ensure puppy user is present
    user: name=puppy state=present group=puppy
  - name: ensure puppy.jpg is present
    copy: src=/root/puppy.jpg dest=/usr/share/nginx/html/puppy.jpg
              owner=puppy group=puppy mode=664
  - name: ensure index.html template is installed
    template: src=/root/index.j2
              dest=/usr/share/nginx/html/index.html

- hosts: kitty
  vars:
    baby: kitty
  tasks:
```

© 2015 Matt Jaynes

```
  - name: ensure kitty group is present
    group: name=kitty state=present
  - name: ensure kitty user is present
    user: name=kitty state=present group=kitty
  - name: ensure kitty.jpg is present
    copy: src=/root/kitty.jpg dest=/usr/share/nginx/html/kitty.jpg
              owner=kitty group=kitty mode=664
  - name: ensure index.html template is installed
    template: src=/root/index.j2 dest=/usr/share/nginx/html/index.html
```

Then run the new directives:

```
root@master:~# ansible-playbook taste.yml
...output omitted...
```

Run nginx

The last thing we need to do is ensure nginx is running so we can browse to our puppy/kitty sites.

Update this part of `taste.yml`:

```
- hosts: all
  tasks:
  - name: ensure nginx is installed
    apt: pkg=nginx state=present update_cache=yes
  - name: ensure nginx is running
    service: name=nginx state=started
```

Run the new directive:

```
root@master:~# ansible-playbook taste.yml
...output omitted...
```

© 2015 Matt Jaynes

Now we can browse to our puppy/kitty sites!

http://puppy.dev/

http://kitty.dev/

Conclusion

Ansible has the lowest learning curve of all the CM tools, so if you found this chapter at all challenging, you should use Ansible and not even consider the other tools.

For convenience, here's the full final `taste.yml` with some added whitespace and comments for clarity:

```
---

# Directives for all children nodes
- hosts: all

  tasks:
  - name: Ensure nginx is installed.
    apt: pkg=nginx state=present update_cache=yes

  - name: Ensure nginx is running.
    service: name=nginx state=started


# Directives for puppy node
- hosts: puppy

  vars:
    baby: puppy

  tasks:
  - name: Ensure puppy group is present.
    group: name=puppy state=present

  - name: Ensure puppy user is present.
    user: name=puppy state=present group=puppy

  - name: Ensure puppy.jpg is present.
```

© 2015 Matt Jaynes

```
    copy: src=/root/puppy.jpg dest=/usr/share/nginx/html/puppy.jpg
              owner=puppy group=puppy mode=664

  - name: Ensure index.html template is installed.
    template: src=/root/index.j2
              dest=/usr/share/nginx/html/index.html


# Directives for kitty node
- hosts: kitty

  vars:
    baby: kitty

  tasks:
  - name: Ensure kitty group is present.
    group: name=kitty state=present

  - name: Ensure kitty user is present.
    user: name=kitty state=present group=kitty

  - name: Ensure kitty.jpg is present.
    copy: src=/root/kitty.jpg dest=/usr/share/nginx/html/kitty.jpg
              owner=kitty group=kitty mode=664

  - name: Ensure index.html template is installed.
    template: src=/root/index.j2 dest=/usr/share/nginx/html/index.html
```

(Note that this chapter is one of the longest in the book not because Ansible is more complex, but because I decided to expand it to be a more extensive introduction to Ansible in this 3rd edition. I go into less depth with the other CM tools in order to keep this a book 'taste test', but Ansible is simple enough that I could give it a bit more coverage here and still keep the chapter pretty short. Just remember that the length of the chapter doesn't represent the complexity of the tool.)

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

SaltStack

Overview

Salt has good documentation and great remote execution functionality. If you don't mind a medium learning curve, then you might consider Salt.

While Ansible is generally used to "push" the directives from the master to the children, the other CM tools like Salt generally have the children nodes "pull" the directives from the master. Salt does this via its "scheduler" and can be set on the minions to pull and run the directives on whatever schedule you define (5 min, 60 min, etc). In our examples, we'll manually trigger the directive runs from the master so we don't have to set up a scheduler and wait for it to run. For more on the scheduler, see: http://docs.saltstack.com/en/latest/topics/jobs/schedule.html

> ***Note:***
> *Ansible can also be set up to similarly "pull" and run on a schedule. See http://docs.ansible.com/ansible/playbooks_intro.html#ansible-pull*

Salt also has `salt-ssh` which is similar to Ansible's push method, so you also have that as an option. It was in 'alpha' for quite a while, but fairly recently became a stable option for Salt. It's not commonly used yet, so we don't cover it here, but if you'd like to read more about it, you can do so here: https://docs.saltstack.com/en/develop/topics/ssh/index.html

© 2015 Matt Jaynes

Documentation

There's a lot of functionality that Salt has that's out of scope for us to cover here, so explore the docs to see its full capabilities.

http://docs.saltstack.com/

Directives Execution Order

Before version 0.17, Salt used its own internal ordering for directives. That required you to do the extra work of explicitly defining dependencies and then managing them vigilantly over the life of the project.

Fortunately, as of Salt version 0.17, that is no longer an issue since Salt now has a new default `state_auto_order` mode which will run your directives in the order you'd expect.

Because of this history, Salt allows you to also define explicit dependencies if you'd like.

The Salt ordering documentation is at:

http://docs.saltstack.com/en/latest/ref/states/ordering.html

Directives Language

YAML and Jinja2. Both are very simple and easy to learn. This makes Salt very accessible for developers of all languages.

Remote Execution / Orchestration

Remote execution / orchestration is standard and robust with Salt. Once you've set up Salt on your master and children nodes, it's available.

© 2015 Matt Jaynes

Terminology

Directives = States

Directives Script = SLS Formula (SLS stands for **S**a**L**t **S**tate)

Children Nodes = Minions

Node metadata = Grains

One of the downsides for beginners to Salt is the nonintuitive terminology.

You get used to it quickly, but you'll find yourself asking "What's a pillar again?" (for the curious it's the "interface used to generate arbitrary data for specific minions").

Setup

Make sure you first set up your servers according to the instructions in the Setup chapter.

Installation

SaltStack has done a great job making the installation quick and simple as you'll see below.

The installation documentation is at:

http://docs.saltstack.com/en/latest/topics/installation/ubuntu.html

© 2015 Matt Jaynes

Install on master node

```
root@master:~# add-apt-repository ppa:saltstack/salt -y
root@master:~# apt-get update
root@master:~# apt-get install salt-master -y
```

We run the `apt-get update` twice since we need to get the updated package lists first in order to install `software-properties-common` and then again to update the package lists for the `saltstack/salt` repository we added.

Install on children nodes

```
root@puppy:~# add-apt-repository ppa:saltstack/salt -y
root@puppy:~# apt-get update
root@puppy:~# apt-get install salt-minion -y
```

```
root@kitty:~# add-apt-repository ppa:saltstack/salt -y
root@kitty:~# apt-get update
root@kitty:~# apt-get install salt-minion -y
```

Set up connectivity between master and children nodes

Edit `/etc/salt/minion` on each child node and add this line:

```
master: master.dev
```

© 2015 Matt Jaynes

Then restart the Salt client on the children (this triggers the certificate requests from the children nodes to the master):

```
root@puppy:~# /etc/init.d/salt-minion restart
```

```
root@kitty:~# /etc/init.d/salt-minion restart
```

View the certificate requests on master:

```
root@master:~# salt-key --list-all
Accepted Keys:
Unaccepted Keys:
kitty.dev
puppy.dev
Rejected Keys:
```

Accept the certificate requests on master:

```
root@master:~# salt-key --accept-all
```

Test the connection with the children:

```
root@master:~# salt '*' test.ping
puppy.dev:
    True
kitty.dev:
    True
```

© 2015 Matt Jaynes

Bootstrapping Children, Certificates, and Maintenance:

You'll notice that we've just had to do some special additional steps (child node client install and certificate verification) that we didn't have to do for Ansible. For Salt (and Chef and Puppet), a client service is needed on the children servers. You also have to do certificate verification so they can communicate with the master node. That means for each new child server you add, you will need these special bootstrap steps to set up the CM tool (though, you could alternatively use `salt-ssh` *to avoid all of this).*

Along with that, you will also need to manage the CM tool client services running on the children nodes and maintain them (resource management, functionality updates, security updates, uptime, etc) for the life of the server. This is yet another maintenance task on top of whatever maintenance you already have for what the server is actually designed for (webserver, cached, db, etc).

Remote execution

Salt gives you remote execution capabilities right away:

```
root@master:~# salt '*' cmd.run 'date'
puppy.dev:
    Wed Jul 29 00:27:19 EDT 2015
kitty.dev:
    Wed Jul 29 00:27:19 EDT 2015
```

You can also target particular servers easily:

```
root@master:~# salt 'puppy*' cmd.run 'date'
puppy.dev:
    Wed Jul 29 00:27:45 EDT 2015
```

© 2015 Matt Jaynes

Another feature I love is that you can get the documentation for all the remote functions you can run by doing:

```
root@master:~# salt '*' sys.doc | less
```

Warning:
Salt uses its own cryptography for network security. That and other factors have led to versions with major security vulnerabilities. Be sure that if you use Salt, you use it on a private secured network if possible and use a version without known vulnerabilities.

For more info, see the Security chapter.

Setting up the directives

Now we're ready to set up the directives to install everything.

The documentation for Salt directives is at: http://docs.saltstack.com/en/latest/ref/states/all/index.html

First create the main directory for our directive files:

```
root@master:~# mkdir /srv/salt
```

Then create the directives file:

```
root@master:~# touch /srv/salt/taste.sls
```

nginx package

We know we'll need to put our image and html files in the nginx web root directory, so let's install nginx first.

© 2015 Matt Jaynes

Add this content to `/srv/salt/taste.sls`:

```
nginx:
  pkg:
    - installed
```

You'll notice that this is YAML, but since the file contains Salt "states" we use the `sls` extension.

First run

Let's run this against the children nodes now:

```
root@master:~# salt '*' state.sls taste
...some output truncated...
kitty.dev:
----------
          ID: nginx
    Function: pkg.installed
      Result: True
     Comment: The following packages were installed/updated: nginx.
     Changes:
              ----------
              nginx:
                  ----------
                  new:
                      1.1.19-1ubuntu0.6
                  old:
Summary
------------
Succeeded: 1
Failed:    0
------------
Total:     1
puppy.dev:
----------
          ID: nginx
    Function: pkg.installed
      Result: True
     Comment: The following packages were installed/updated: nginx.
     Changes:
```

© 2015 Matt Jaynes

```
              ----------
              nginx:
                  ----------
                  new:
                      1.1.19-1ubuntu0.6
                  old:
Summary
------------
Succeeded: 1
Failed:    0
------------
Total:     1
```

Great! It looks like it installed nginx without any trouble.

Oddity:
You'll notice that the command we ran was pretty odd. You would expect to the command to look like `salt '*' taste.sls` *right? Instead, we specify this other* `state.sls` *file that we've never seen and then specify our* `taste.sls` *file, except we leave off the extension and just put* `taste`.

Note:
For examples more complex than our trivial puppy/kitty servers, you'll want to use Salt's special `top.sls` *file. Despite its "sls" extension, this isn't a file that can contain Salt States. Instead, it's a special targeting and configuration file that allows you to define environments (production, staging, etc) and roles (webserver, db, etc) for servers as well as other options.*

When you run Salt with the default `top.sls` *setup, you use this command:*
`salt '*' state.highstate`

You'll notice that command is a bit more intuitive than our earlier command:
`salt '*' state.sls taste`

For more on using `top.sls`*, see http://docs.saltstack.com/en/latest/ref/states/top.html*

Image files (and grains and dependencies)

Now, let's set up the image files.

Before we can do that though, we need to be able to target which server gets which image file.

To do that, we'll use "grains" which is what Salt uses for metadata on the servers (like hostname, architecture, etc).

We'll use the "host" grain and a Jinja2 conditional to target the right children nodes.

© 2015 Matt Jaynes

Add this to `taste.sls`:

```
nginx:
  pkg:
    - installed

{% if grains['host'] == 'puppy' %}
/usr/share/nginx/html/puppy.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/puppy.jpg
    - source_hash: md5=8f3a3661eb7b34036781dac5b6cd9d32
{% endif %}

{% if grains['host'] == 'kitty' %}
/usr/share/nginx/html/kitty.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/kitty.jpg
    - source_hash: md5=f39b24938f200e59ac9cb823fb71cad4
{% endif %}
```

Conveniently, Salt lets us use the remote image files. We just needed to provide the `md5` hash to ensure we're getting the exact file we're expecting.

Warning:
You may be tempted to indent the lines within the Jinja2 conditional. Don't! It will break and you'll get an error like "Data failed to compile".

Note:
To get the md5 hash on OSX: `md5 kitty.jpg`
To get the md5 hash on most linux distros: `md5sum kitty.jpg`

© 2015 Matt Jaynes

Now run the new directives:

```
root@master:~# salt '*' state.sls taste
kitty.dev:
----------
          ID: nginx
    Function: pkg.installed
      Result: True
     Comment: Package nginx is already installed
     Changes:
----------
          ID: /usr/share/nginx/html/kitty.jpg
    Function: file.managed
      Result: True
     Comment: File /usr/share/nginx/html/kitty.jpg updated
     Changes:
              ----------
              diff:
                  New file
              mode:
                  0644

Summary
------------
Succeeded: 2
Failed:    0
------------
Total:     2
puppy.dev:
----------
          ID: nginx
    Function: pkg.installed
      Result: True
     Comment: Package nginx is already installed
     Changes:
----------
          ID: /usr/share/nginx/html/puppy.jpg
    Function: file.managed
      Result: True
     Comment: File /usr/share/nginx/html/puppy.jpg updated
     Changes:
              ----------
              diff:
                  New file
              mode:
                  0644
```

© 2015 Matt Jaynes

```
Summary
------------
Succeeded: 2
Failed:    0
------------
Total:     2
```

If you'd like to see all the grains data for your children nodes, run:

```
root@master:~# salt '*' grains.items
```

User / Group creation

Now, we need to create the puppy/kitty groups and users so we can update the image file ownerships.

```
nginx:
  pkg:
    - installed

{% if grains['host'] == 'puppy' %}
puppy:
  group:
    - present
  user:
    - present
    - groups:
      - puppy

/usr/share/nginx/html/puppy.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/puppy.jpg
    - source_hash: md5=8f3a3661eb7b34036781dac5b6cd9d32
    - user: puppy
    - group: puppy
    - mode: 664
{% endif %}
```

© 2015 Matt Jaynes

```
{% if grains['host'] == 'kitty' %}
kitty:
  group:
    - present
  user:
    - present
    - groups:
      - kitty

/usr/share/nginx/html/kitty.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/kitty.jpg
    - source_hash: md5=f39b24938f200e59ac9cb823fb71cad4
    - user: kitty
    - group: kitty
    - mode: 664
{% endif %}
```

And now run it:

```
root@master:~# salt '*' state.sls taste
puppy.dev:
...output truncated...
Summary
------------
Succeeded: 4
Failed:    0
------------
Total:     4

kitty.dev:
...output truncated...
Summary
------------
Succeeded: 4
Failed:    0
------------
Total:     4
```

Great, it ran smoothly.

© 2015 Matt Jaynes

HTML template

Now, we'll make the html template with the Jinja2 templating language.

Create the html template as `index.html` in `/srv/salt/index.html` and add these contents:

```
<html>
  <body bgcolor="gray">
    <center>
      <img src="/{{grains['host']}}.jpg">
    </center>
  </body>
</html>
```

Conveniently, our hostnames are the same as the base name for the image file. So we'll just simply utilize the grains data we used earlier and set the variable in the Jinja2 syntax with double curly brackets.

Here's the resulting directive for the template:

```
/usr/share/nginx/html/index.html:
  file:
    - managed
    - source: salt://index.html
    - template: jinja
```

You'll notice that Salt looks for its files from the base of its main directory - so for `/srv/salt/index.html` we use `salt://index.html`.

Now let's run it:

```
root@master:~# salt '*' state.sls taste
...output omitted...
```

© 2015 Matt Jaynes

Run nginx

The last thing we need to do is ensure nginx is running so we can browse to our puppy/kitty sites.

Update this part of `taste.sls`:

```
nginx:
  pkg:
    - installed
  service:
    - running
    - enable: True
```

The `enable: True` line tells the system to set up the service so that it will start automatically if the server is rebooted.

Now run it:

```
root@master:~# salt '*' state.sls taste
...output omitted...
```

Now we can browse to our puppy/kitty sites!

http://puppy.dev/

http://kitty.dev/

Conclusion

Salt has a higher learning curve, but has thorough documentation and remote execution capabilities.

The main issues I had with it were the higher learning curve, the terminology, and some nonintuitive commands.

© 2015 Matt Jaynes

For the official walkthrough with additional details, see:

http://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html

For convenience, our full final `taste.sls` is:

```
nginx:
  pkg:
    - installed
  service:
    - running
    - enable: True

/usr/share/nginx/html/index.html:
  file:
    - managed
    - source: salt://index.html
    - template: jinja

{% if grains['host'] == 'puppy' %}
puppy:
  group:
    - present
  user:
    - present
    - groups:
      - puppy

/usr/share/nginx/html/puppy.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/puppy.jpg
    - source_hash: md5=8f3a3661eb7b34036781dac5b6cd9d32
    - user: puppy
    - group: puppy
    - mode: 664
{% endif %}

{% if grains['host'] == 'kitty' %}
kitty:
  group:
    - present
  user:
    - present
    - groups:
      - kitty
```

© 2015 Matt Jaynes

```
/usr/share/nginx/html/kitty.jpg:
  file:
    - managed
    - source: https://raw.github.com/nanobeep/tt/master/kitty.jpg
    - source_hash: md5=f39b24938f200e59ac9cb823fb71cad4
    - user: kitty
    - group: kitty
    - mode: 664
{% endif %}
```

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Chef

Overview

Chef was the most difficult CM tool to get up and going. The onboarding process in the past was plagued with confusing documentation and an overly complex installation.

When updating this book for the 3rd Edition, I noticed that they have improved the documentation and installation process quite a bit, so it is less painful than before. However, it is still really confusing. Even for me writing the 3rd Edition of this book and having worked on several production projects in Chef, I still got lost from time to time and it took a lot of mental energy just to wrap my head around all the moving parts and oddities around Chef.

Rather than have a long arduous chapter defining all the oddities, I'm just showing you the "happy path" here.

If I had used Chef Software Inc's "Hosted Chef" master server product, then I probably could have avoided some of the pain. However, for this to be a fair comparison of the tools, I really needed to show how to set up the open source version.

Documentation

http://docs.chef.io/

Directives Execution Order

Chef executes the directives in the order you'd expect: sequentially as they're written in your directives script.

© 2015 Matt Jaynes

Directives Language

Ruby with an extended DSL (Domain Specific Language). While this is very powerful and convenient for Rubyists, it makes things a little more challenging for non-Ruby developers. Fortunately, Ruby is a simple elegant language that is easy to learn. Here's a guide for getting started with Ruby for Chef users:

https://docs.chef.io/just_enough_ruby_for_chef.html

Remote Execution / Orchestration

Chef includes the `knife` tool which has remote execution capabilities (among other things), but configuring it was unnecessarily difficult and it feels clunky to use. You can read more about it here:

https://docs.chef.io/knife.html

Terminology

Directives = Resources

Directives Script = Recipe

Group of recipes and supporting files = Cookbook

Ohai is the utility Chef uses for detecting node metadata (like architecture, OS distribution, RAM available, etc).

Setup

Make sure you first set up your servers according to the instructions in the Setup chapter.

© 2015 Matt Jaynes

Install Chef Server on the master node

Let's get started with the walk-through by setting up the master node:

> ***Caution:*** *The master.dev server must be set up with 4GB RAM, otherwise it will run out of memory and fail.*

Download the Chef Server package and install it

```
root@master:~# wget https://web-dl.packagecloud.io/chef/stable/packages/ubuntu/trusty/che
f-server-core_12.1.0-1_amd64.deb

root@master:~# dpkg -i chef-server-core_12.1.0-1_amd64.deb
```

Set up Chef Server and create the admin user and organization

```
root@master:~# chef-server-ctl reconfigure
...nearly 9000 lines of output from this command...

root@master:~# chef-server-ctl user-create admin Example Administrator \
> admin@example.com tempPassword --filename admin.pem

root@master:~# chef-server-ctl org-create example-org "Example Org" \
> --association_user admin --filename example-org.pem
```

Set up a workstation

Chef is a bit different from the other tools in that it requires installing a 'workstation' in order to interact with the master server.

© 2015 Matt Jaynes

Download and install the Chef Development Kit:

```
root@work:~# wget https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64/c
hefdk_0.6.2-1_amd64.deb

root@work:~# dpkg -i chefdk_0.6.2-1_amd64.deb
```

Add `ruby` to the `$PATH` environment variable:

```
root@work:~# echo 'eval "$(chef shell-init bash)"' >> ~/.bashrc
root@work:~# source ~/.bashrc
```

Install Git:

```
root@work:~# apt-get update
root@work:~# apt-get install git -y
```

Clone the `chef-repo`:

```
root@work:~# git clone git://github.com/opscode/chef-repo.git
```

Set up location for keys:

```
root@work:~# mkdir -p ~/chef-repo/.chef
```

Add master node to SSH `known_hosts` file:

```
root@work:~# ssh-keyscan master.dev > .ssh/known_hosts
```

© 2015 Matt Jaynes

Retrieve the keys from the master node:

```
root@work:~# scp root@master.dev:/root/admin.pem /root/chef-repo/.chef/admin.pem
root@work:~# scp root@master.dev:/root/example-org.pem /root/chef-repo/.chef/example-or
g.pem
```

Create the knife configuration file:

```
root@work:~# mkdir -p ~/.chef
root@work:~# touch ~/.chef/knife.rb
```

And add this to `~/.chef/knife.rb`:

```
log_level                :info
log_location             STDOUT
node_name                'admin'
client_key               '/root/chef-repo/.chef/admin.pem'
validation_client_name   'example-org-validator'
validation_key           '/root/chef-repo/.chef/example-org.pem'
chef_server_url          'https://master.dev:443/organizations/example-org'
syntax_check_cache_path  '/root/chef-repo/.chef/syntax_check_cache'
cookbook_path [ '/root/chef-repo/cookbooks' ]
```

The rest of these commands need to be done within the chef-repo directory:

```
root@work:~# cd chef-repo/
```

© 2015 Matt Jaynes

Get SSL certificates from master.dev:

```
root@work:~/chef-repo# knife ssl fetch
WARNING: Certificates from master.dev will be fetched and placed in your trusted_cert
directory (/root/chef-repo/.chef/trusted_certs).

Knife has no means to verify these are the correct certificates. You should
verify the authenticity of these certificates after downloading.

Adding certificate for master.dev in /root/chef-repo/.chef/trusted_certs/master_dev.crt
```

Verify that knife on the workstation can communicate with master.dev:

```
root@work:~/chef-repo# knife user list
admin
```

Install Chef on the children nodes from the workstation

This was the one bright spot with Chef - bootstrapping the children nodes was easy:

```
root@work:~/chef-repo# knife bootstrap puppy.dev -x root
...about 50 lines of output...

root@work:~/chef-repo# knife bootstrap kitty.dev -x root
...about 50 lines of output...
```

© 2015 Matt Jaynes

Set up the directives

Create a 'cookbook':

```
root@work:~/chef-repo# knife cookbook create taste
** Creating cookbook taste in /root/chef-repo/cookbooks
** Creating README for cookbook: taste
** Creating CHANGELOG for cookbook: taste
** Creating metadata for cookbook: taste
```

Then download the image files:

```
root@work:~/chef-repo# wget https://raw.github.com/nanobeep/tt/master/puppy.jpg \
>       --output-document=/root/chef-repo/cookbooks/taste/files/default/puppy.jpg

root@work:~/chef-repo# wget https://raw.github.com/nanobeep/tt/master/kitty.jpg \
>       --output-document=/root/chef-repo/cookbooks/taste/files/default/kitty.jpg
```

Then create the ERB (embedded ruby) template for `index.html`:

```
root@work:~/chef-repo# touch cookbooks/taste/templates/default/index.html.erb
```

And add this to `index.html.erb`:

```
<html>
  <body bgcolor="gray">
    <center>
      <img src="/<%= node['hostname'] %>.jpg">
    </center>
  </body>
</html>
```

© 2015 Matt Jaynes

Then edit `/root/chef-repo/cookbooks/taste/recipes/default.rb` and add this:

```
execute 'apt-get-update' do
  command 'apt-get update'
  ignore_failure true
end

apt_package "nginx" do
  action :install
end

service "nginx" do
  action [ :enable, :start ]
end

template "/usr/share/nginx/html/index.html" do
  source "index.html.erb"
  action :create
  mode "664"
end

if node['hostname'] == "puppy"
  group "puppy" do
    action :create
  end

  user "puppy" do
    action :create
    gid "puppy"
  end

  cookbook_file "/usr/share/nginx/html/puppy.jpg" do
    source "puppy.jpg"
    action :create
    owner "puppy"
    group "puppy"
    mode "664"
  end
end

if node['hostname'] == "kitty"
  group "kitty" do
    action :create
  end

  user "kitty" do
```

© 2015 Matt Jaynes

```
    action :create
    gid "kitty"
  end

  cookbook_file "/usr/share/nginx/html/kitty.jpg" do
    source "kitty.jpg"
    action :create
    owner "kitty"
    group "kitty"
    mode "664"
  end
end
```

Upload the directives and supporting files to the master

```
root@work:~/chef-repo# knife cookbook upload -a
Uploading taste        [0.1.0]
Uploaded all cookbooks.
```

Add the directives to the children nodes' "run list"

```
root@work:~/chef-repo# knife node run_list add puppy.dev 'taste'
puppy.dev:
  run_list: recipe[taste]

root@work:~/chef-repo# knife node run_list add kitty.dev 'taste'
kitty.dev:
  run_list: recipe[taste]
```

© 2015 Matt Jaynes

Run the directives on the children

It's not necessary to do this manually, but we do this since we don't want to wait the 30 minutes for it to run automatically.

```
root@puppy:~# chef-client
```

```
root@kitty:~# chef-client
```

View the sites:

http://puppy.dev/

http://kitty.dev/

Conclusion

Chef was known as a great alternative to Puppet for many years - particularly because of its sequential order of execution for directives. That is no longer an advantage though since all the CM tools have sequential order of execution now. Chef is overly complex, bloated, and many miles behind Ansible and Salt in usability.

A positive for Chef is its large established community. They've built a great group with a lot of amazing folks and have done a ton to advance the state of systems. Unfortunately, it is turning into a non-competitive legacy project that is suffering from its age, complexity, and over-engineering.

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Puppet

Overview

Puppet is mature, but like Chef, it suffers from bloat and over-complexity.

Like for Chef, this chapter was initially very long (38 pages) and detailed all the rough spots I ran into. Again like the Chef chapter, I've trimmed it down to just show the "happy path" which shows the basics of how to set up the project but isn't a full walk-through like I did for Ansible and Salt. For this *very* simple project it took 1 very long unpleasant day.

Documentation

http://docs.puppetlabs.com/

Directives Execution Order

Puppet used to use its own internal ordering for directives, so you used to have to consider that cost and be vigilant about explicitly defining dependencies. Fortunately, as of version 3.3.0 it now has the sequential execution of directives that you would expect.

Directives Language

Puppet uses its own custom configuration language. It's fairly simple, but does require learning. You can read more about it here:

https://docs.puppetlabs.com/puppet/latest/reference/lang_summary.html

© 2015 Matt Jaynes

Remote Execution / Orchestration

Puppet doesn't come with remote execution / orchestration functionality built-in. However, they recommend using `mcollective` which is another open source tool provided by Puppet Labs. It's fairly involved to set up and learn as you can see from the documentation: http://docs.puppetlabs.com/mcollective/. You'll be much better off using Ansible or Salt for remote execution and orchestration.

Terminology

Directives = Resources

Directives Script = Manifest

Facter is the utility Puppet uses for detecting node metadata: https://puppetlabs.com/facter

Setup

Make sure you first set up your servers according to the instructions in the Setup chapter.

Caution: *The master.dev server must be set up with 4GB RAM, otherwise it will run out of memory and fail.*

© 2015 Matt Jaynes

Install and start the Puppet server on master.dev

Download and install the server package:

```
root@master:~# wget http://apt.puppetlabs.com/puppetlabs-release-pc1-trusty.deb
root@master:~# dpkg --install puppetlabs-release-pc1-trusty.deb
root@master:~# apt-get update
root@master:~# apt-get install puppetserver -y
```

Start the Puppet master service:

```
root@master:~# service puppetserver start
```

Install Puppet client on the children nodes

Download and install the client package:

```
root@puppy:~# wget http://apt.puppetlabs.com/puppetlabs-release-pc1-trusty.deb
root@puppy:~# dpkg --install puppetlabs-release-pc1-trusty.deb
root@puppy:~# apt-get update
root@puppy:~# apt-get install puppet-agent -y
```

```
root@kitty:~# wget http://apt.puppetlabs.com/puppetlabs-release-pc1-trusty.deb
root@kitty:~# dpkg --install puppetlabs-release-pc1-trusty.deb
root@kitty:~# apt-get update
root@kitty:~# apt-get install puppet-agent -y
```

© 2015 Matt Jaynes

Configure and start the Puppet client on the children nodes

Now we'll tell the children nodes where their master is and what their certificate name should be.

Add these settings to the puppy node in `/etc/puppetlabs/puppet/puppet.conf`:

```
[main]
server = master.dev
certname = puppy.dev
```

Add these settings to the kitty node in `/etc/puppetlabs/puppet/puppet.conf`:

```
[main]
server = master.dev
certname = kitty.dev
```

Now we'll start and enable the Puppet clients (the 'enable' tells the system to automatically start the puppet service on reboots, etc):

```
root@puppy:~# /opt/puppetlabs/bin/puppet resource service puppet \
> ensure=running enable=true
Notice: /Service[puppet]/ensure: ensure changed 'stopped' to 'running'
service { 'puppet':
  ensure => 'running',
  enable => 'true',
}
```

© 2015 Matt Jaynes

```
root@kitty:~# /opt/puppetlabs/bin/puppet resource service puppet \
> ensure=running enable=true
Notice: /Service[puppet]/ensure: ensure changed 'stopped' to 'running'
service { 'puppet':
  ensure => 'running',
  enable => 'true',
}
```

Note: *You'll notice that we used the full path of* `/opt/puppetlabs/bin/puppet` *for the* `puppet` *command. That is because Puppet by default now installs its commands outside of the default PATH. If you would like to add the Puppet commands to your PATH environment variable, then add* `PATH=/opt/puppetlabs/bin:$PATH;export PATH` *to the server's* `.bashrc` *file and run* `source .bashrc`.

Have the master node sign the certificate requests of the children

View the certificate requests:

```
root@master:~# /opt/puppetlabs/bin/puppet cert list
"kitty.dev" (SHA256) CE:53:B6:AE:67:65:BD:76:8B:53:40:05:75:B6:A6:66:89:70:E5:20:85:B7:D
D:62:B0:8F:99...
"puppy.dev" (SHA256) B6:A8:6E:37:46:46:7C:F6:C9:E7:5D:C8:A7:2C:B4:65:36:4C:30:D9:D4:06:B
A:0B:7E:40:2E...
```

Note: *If you don't see the children node certificate requests, then run this on each child node:* `puppet agent --test`. *That will trigger the child to send the certificate request to the master node.*

© 2015 Matt Jaynes

Sign the certificate requests:

```
root@master:~# /opt/puppetlabs/bin/puppet cert sign --all
Notice: Signed certificate request for puppy.dev
Notice: Removing file Puppet::SSL::CertificateRequest puppy.dev at '/etc/puppetlabs/puppe
t/ssl/ca/requests/puppy.dev.pem'
Notice: Signed certificate request for kitty.dev
Notice: Removing file Puppet::SSL::CertificateRequest kitty.dev at '/etc/puppetlabs/puppe
t/ssl/ca/requests/kitty.dev.pem'
```

Set up the directives on master.dev

Add the supporting files:

```
root@master:~# mkdir -p \
> /etc/puppetlabs/code/environments/production/modules/taste/files

root@master:~# mkdir -p \
> /etc/puppetlabs/code/environments/production/modules/taste/templates

root@master:~# wget https://raw.github.com/nanobeep/tt/master/puppy.jpg \
> --output-document=/etc/puppetlabs/code/environments/production/modules/taste/files/pupp
y.jpg

root@master:~# wget https://raw.github.com/nanobeep/tt/master/kitty.jpg \
> --output-document=/etc/puppetlabs/code/environments/production/modules/taste/files/kitt
y.jpg
```

© 2015 Matt Jaynes

Then add `/etc/puppetlabs/code/environments/production/modules/taste/templates/index.erb` with this content:

```
<html>
  <body bgcolor="gray">
    <center>
      <img src="/<%= @hostname %>.jpg">
    </center>
  </body>
</html>
```

Then add `/etc/puppetlabs/code/environments/production/manifests/site.pp` with this content:

```
package { 'nginx':
  ensure => installed
}

service { "nginx":
  ensure => "running",
  require => Package["nginx"],
}

file { "/usr/share/nginx/html/index.html":
  content => template("taste/index.erb"),
  require => Package["nginx"],
}

if $hostname == "puppy" {
  group { "puppy":
    name => "puppy",
    ensure => "present",
  }
  user { "puppy":
    name => "puppy",
    groups => "puppy",
    require => Group["puppy"],
  }
  file { "/usr/share/nginx/html/puppy.jpg":
    owner => "puppy",
    group => "puppy",
```

© 2015 Matt Jaynes

```
    mode => "0664",
    source => "puppet:///modules/taste/puppy.jpg",
    require => [ User["puppy"], Package["nginx"] ],
  }
}

if $hostname == "kitty" {
  group { "kitty":
    name => "kitty",
    ensure => "present",
  }
  user { "kitty":
    name => "kitty",
    groups => "kitty",
    require => Group["kitty"],
  }
  file { "/usr/share/nginx/html/kitty.jpg":
    owner => "kitty",
    group => "kitty",
    mode => "0664",
    source => "puppet:///modules/taste/kitty.jpg",
    require => [ User["kitty"], Package["nginx"] ],
  }
}
```

Run the directives on the children nodes

On puppy.dev:

```
root@puppy:~# /opt/puppetlabs/bin/puppet agent --test
Info: Caching certificate for puppy.dev
Info: Caching certificate_revocation_list for ca
Info: Caching certificate for puppy.dev
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for puppy.dev
Info: Applying configuration version '1439234318'
Notice: /Stage[main]/Main/Package[nginx]/ensure: created
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]/content:
--- /usr/share/nginx/html/index.html    2014-03-04 06:46:45.000000000 -0500
+++ /tmp/puppet-file20150810-1693-jd7zm0    2015-08-10 15:18:51.514756000 -0400
@@ -1,25 +1,7 @@
-<!DOCTYPE html>
 <html>
```

```
-<head>
-<title>Welcome to nginx!</title>
-<style>
-    body {
-        width: 35em;
-        margin: 0 auto;
-        font-family: Tahoma, Verdana, Arial, sans-serif;
-    }
-</style>
-</head>
-<body>
-<h1>Welcome to nginx!</h1>
-<p>If you see this page, the nginx web server is successfully installed and
-working. Further configuration is required.</p>
-
-<p>For online documentation and support please refer to
-<a href="http://nginx.org/">nginx.org</a>.<br/>
-Commercial support is available at
-<a href="http://nginx.com/">nginx.com</a>.</p>
-
-<p><em>Thank you for using nginx.</em></p>
-</body>
+  <body bgcolor="gray">
+    <center>
+      <img src="/puppy.jpg">
+    </center>
+  </body>
 </html>

Info: Computing checksum on file /usr/share/nginx/html/index.html
Info: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]: Filebucketed /usr/share/n
ginx/html/index.html to puppet with sum e3eb0a1df437f3f97a64aca5952c8ea0
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]/content: content change
d '{md5}e3eb0a1df437f3f97a64aca5952c8ea0' to '{md5}50ef3f160dd9d76fa6765095bb100ebf'
Notice: /Stage[main]/Main/Group[puppy]/ensure: created
Notice: /Stage[main]/Main/User[puppy]/ensure: created
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/puppy.jpg]/ensure: defined content a
s '{md5}f39b24938f200e59ac9cb823fb71cad4'
Notice: Applied catalog in 10.43 seconds
```

On kitty.dev:

```
root@kitty:~# /opt/puppetlabs/bin/puppet agent --test
Info: Caching certificate for kitty.dev
```

```
Info: Caching certificate_revocation_list for ca
Info: Caching certificate for kitty.dev
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for kitty.dev
Info: Applying configuration version '1439234318'
Notice: /Stage[main]/Main/Package[nginx]/ensure: created
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]/content:
--- /usr/share/nginx/html/index.html    2014-03-04 06:46:45.000000000 -0500
+++ /tmp/puppet-file20150810-1693-jd7zm0    2015-08-10 15:18:51.514756000 -0400
@@ -1,25 +1,7 @@
-<!DOCTYPE html>
 <html>
-<head>
-<title>Welcome to nginx!</title>
-<style>
-    body {
-        width: 35em;
-        margin: 0 auto;
-        font-family: Tahoma, Verdana, Arial, sans-serif;
-    }
-</style>
-</head>
-<body>
-<h1>Welcome to nginx!</h1>
-<p>If you see this page, the nginx web server is successfully installed and
-working. Further configuration is required.</p>
-
-<p>For online documentation and support please refer to
-<a href="http://nginx.org/">nginx.org</a>.<br/>
-Commercial support is available at
-<a href="http://nginx.com/">nginx.com</a>.</p>
-
-<p><em>Thank you for using nginx.</em></p>
-</body>
+  <body bgcolor="gray">
+    <center>
+      <img src="/kitty.jpg">
+    </center>
+  </body>
 </html>

Info: Computing checksum on file /usr/share/nginx/html/index.html
Info: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]: Filebucketed /usr/share/n
ginx/html/index.html to puppet with sum e3eb0a1df437f3f97a64aca5952c8ea0
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/index.html]/content: content change
d '{md5}e3eb0a1df437f3f97a64aca5952c8ea0' to '{md5}50ef3f160dd9d76fa6765095bb100ebf'
Notice: /Stage[main]/Main/Group[kitty]/ensure: created
```

```
Notice: /Stage[main]/Main/User[kitty]/ensure: created
Notice: /Stage[main]/Main/File[/usr/share/nginx/html/kitty.jpg]/ensure: defined content a
s '{md5}f39b24938f200e59ac9cb823fb71cad4'
Notice: Applied catalog in 10.43 seconds
```

View the sites:

http://puppy.dev/

http://kitty.dev/

Conclusion

An advantage with Puppet is its large and mature community. Puppet was a great option for many years, however its user experience is now well behind that of Ansible and Salt. The learning curve is high and it feels heavy and over-engineered, but not quite as bad as Chef.

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Where Docker Fits In

Docker has hit the systems scene with great fanfare. While it's an exciting advancement for some systems scenarios, there are some large misunderstandings around it. In a nutshell, it can add a lot of complexity to your systems and that cost often doesn't justify the benefits (though sometimes it does). Often, there are much simpler tools that will solve the same problems Docker attempts to solve.

Narrowly focused advice!

This book focuses on tools that are primarily used for multi-host production server environments. So, my discussion of Docker is nearly entirely limited to multi-host setups of mission-critical systems. Please keep that in mind since my coverage and advice will probably not apply to the many other scenarios you can use Docker for.

Background on Docker

This chapter assumes a basic understanding of what Docker is and how it works generally.

It's beyond the scope of this book to give a full coverage of Docker, so if you're totally new to Docker, first go through these resources before continuing:

What is Docker?

Docker Basics

© 2015 Matt Jaynes

There's also an addendum at the end of this chapter that shows an example of how to set up the puppy.dev server with Docker. Remember though that it's a drastically oversimplified example that looks super easy and simple for that scenario, but in a real-life multi-host production setup it would be significantly more complex.

Brief Description of Docker

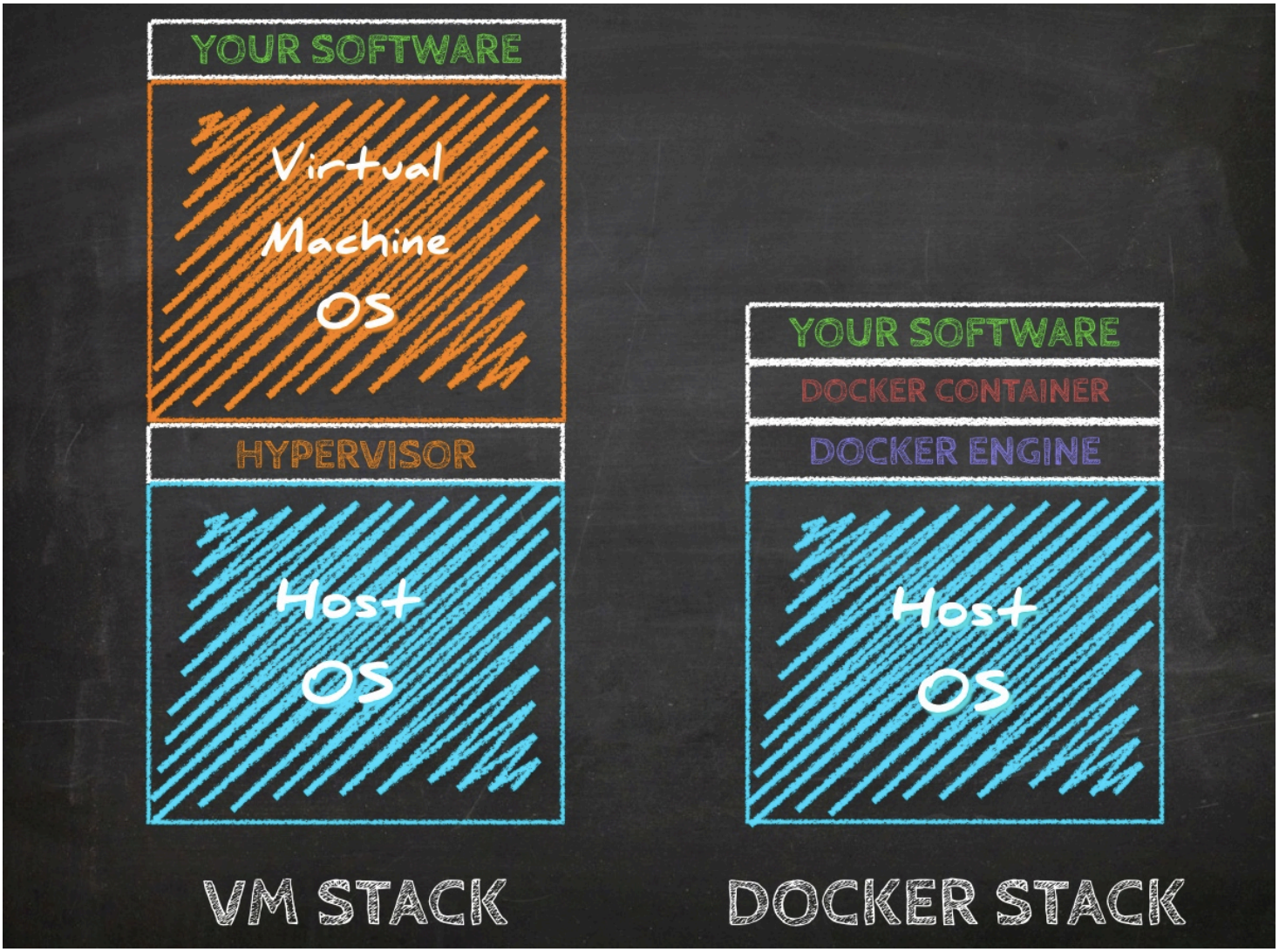
Docker is basically a really nice user-interface for version-controlled shareable Linux containers. I'm greatly simplifying Docker of course, and it's much more than that, but it's useful to think of Docker as a way to run extremely light-weight virtual machines (VMs) that are auditable and shareable due to their unique versioning system (similar to how you'd track and share code in git).

Performance over VMs

A full virtual machine is comparatively slow and resource intensive. That's because a VM is running its own full operating system on top of the host operating system. Docker instead runs as a 'container' which utilizes much of the underlying host operating system so that it can stay lean and fast.

'Containerization' is not new. Companies who manage very large pools of servers (many thousands) often use some type of containerized setup. However, the learning curve and added complexity of using containers is usually cost-prohibitive for most companies with system setups that don't span data-centers.

© 2015 Matt Jaynes



Version control and sharing

One of the nice features of Docker is that it allows you to define the containers in simple code (Dockerfiles) and then share the resulting container images.

© 2015 Matt Jaynes

Here's an example of a Docker file:

```
#
# Nginx Dockerfile
#
# https://github.com/dockerfile/nginx
#

# Pull base image.
FROM dockerfile/ubuntu

# Install Nginx.
RUN \
  add-apt-repository -y ppa:nginx/stable && \
  apt-get update && \
  apt-get install -y nginx && \
  echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
  chown -R www-data:www-data /var/lib/nginx

# Define mountable directories.
VOLUME ["/data", "/etc/nginx/sites-enabled", "/var/log/nginx"]

# Define working directory.
WORKDIR /etc/nginx

# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80
EXPOSE 443
```

You can also install all the software on the container manually (or via a CM tool) and Docker will keep track of all those changes and you can then save the container as a Docker image which can then also be shared with others.

This allows for simple auditing of system setups. Docker assigns a version to each iteration of an image so you can then track the changes that it's gone through over time. It's similar to being able to access the history of your codebase through a version control system like git. This also gives you the ability revert an image to a previous time in history.

If you're familiar with Amazon Machine Images (AMIs) that are used on Amazon's Elastic Compute Cloud (EC2), the Docker images are similar, but far more portable, much lighter, faster to use, and easier to audit and share.

Consistency across environments

Docker allows you to use the identical server setups across environments. You can use the same Docker image on your development machine that you use in production. This lets you avoid issues like the classic "It worked on my machine" problem where what worked in one environment doesn't work in another.

Golden image pattern (without the cruft)

Speaking of Amazon's AMIs, there are some companies that don't use configuration management tools at all. Instead they run a server, install and configure the software on it, then they save a snapshot (like an AMI) of that server so that they can create new instances of that server later.

This approach is called the "Golden Image" pattern.

This pattern works well until you need to make a small change to a group of servers. Let's say that you have 20 app servers that are all identical since you used a golden 'app' image to create them all. If a small change needs to be made to the app servers, the golden image pattern dictates that you create a new golden 'app' image, then replace all of your 20 app servers with new servers running the new golden 'app' image.

© 2015 Matt Jaynes

Going through this update process on Amazon AMIs would be tedious and time consuming. That is because you are dealing with heavy-weight VMs and replacing the entire host with a new host launched from a new image.

The Golden Image pattern works well for some setups, but usually only if those systems rarely change.

Docker now solves a lot of the problems with the Golden Image pattern. Because Docker is so fast and lightweight, it makes the golden image update flow relatively quick and painless.

For example, let's say that we have the same scenario as earlier with 20 app servers that we need to make a small update on. This time, we're using Docker to run the app image as a container on the app host servers. The host servers are bare bones except for having Docker installed on them. So, now if we want to make a change to the app servers, we don't need to replace (destroy and relaunch) the host servers. Instead, we just update the Docker containers to use the new 'app' server image. That's often a near instantaneous update rather than the arduous process it would be otherwise.

Why updates are so fast

Docker saves its images in cached layers. On a brand new host server, it would still have to download the full server image, however, for subsequent changes to the Docker image, it only downloads the new layers.

For example, on a particular server image, if we needed the 'curl' package installed, we would install curl then save the new image. Then when we instruct our existing Dockerized servers to update themselves, they would only download that one new layer of the image with the curl

© 2015 Matt Jaynes

update. So, rather than a server image update requiring something like a 1GB download, it might require only a 1MB download. That feature makes server updates via Docker very fast.

Here's an example where I download a Docker image to use:

```
root@server:~# docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx

902b87aaaec9: Pull complete
9a61b6b1315e: Pull complete
aface2a79f55: Pull complete
5dd2638d10a1: Pull complete
97df1ddba09e: Pull complete
e7e7a55e9264: Pull complete
72b67c8ad0ca: Pull complete
9108e25be489: Pull complete
6dda3f3a8c05: Pull complete
42d2189f6cbe: Pull complete
3cb7f49c6bc4: Pull complete
a486da044a3f: Already exists
library/nginx:latest: The image you are pulling has been verified. Important: image verif
ication is a tech preview feature and should not be relied on to provide security.

Digest: sha256:77e8d942886504b177cf6fa7e8199eaf3ba23ee54c7c56ce697e3060a66f02ec
Status: Downloaded newer image for nginx:latest
```

You'll notice that instead of one big download it downloaded multiple parts. Each part is an iteration of the image, similar to commits in version controlled code. Each part is essentially a layer on top of the previous layers and represents a time in history for this image. Once we've downloaded the base image, any updates to the image will only require downloading the new layer(s) instead of having to download the whole image.

© 2015 Matt Jaynes

Misconceptions

Docker is a great tool for some scenarios, but there are several misconceptions I see come up regularly about using Docker.

Misconception: If I learn Docker then I don't have to learn the other systems stuff!

Someday this may be true. However, currently, it's not the case. It's best to think of Docker as an advanced optimization for edge cases. Yes, it is cool and powerful, but it adds significantly to the complexity of your systems and should only be used in mission critical systems if you are an expert system administrator that understands all the essential points of how to use it safely in production.

At the moment, you need **more** systems expertise to use Docker, not less! Nearly every article you'll read on Docker will show you the extremely simple use-cases and will ignore the complexities of using Docker on multi-host production systems. This gives a false impression of what it takes to actually use Docker in production.

To run Docker in a safe robust way for a typical multi-host production environment requires very careful management of many variables:

- **secured private image repository (index)**
- **orchestrating container deploys with zero downtime**
- **orchestrating container deploy roll-backs**
- **networking between containers on multiple hosts**
- **managing container logs**
- **managing container data (db, etc)**
- **creating images that properly handle init, logs, etc**

© 2015 Matt Jaynes

- **much much more...**

This is not impossible and can all be done - several large companies are using Docker in production, but it's definitely non-trivial. This will likely change as the ecosystem around Docker matures, but currently if you're going to attempt using Docker seriously in production, you need to be **very** skilled at systems management and orchestration.

For a sense of what I mean, see these articles that get the closest to production reality that I've found so far (but still miss many critical elements you'd need):

Easily Deploy Redis Backed Web Apps With Docker

Integrating Docker with Jenkins for Continuous Deployment of a Ruby on Rails App

Using Docker with Github and Jenkins for Repeatable Deployments

Fixing the Docker Ubuntu image on Rackspace Cloud

If you don't want to have to learn how to manage servers, you should use a Platform-as-a-Service (PaaS) like Heroku. Docker isn't the solution!

Misconception: You should have only one process per Docker container!

It's important to understand that it is far simpler to manage Docker if you view it as role-based virtual machine rather than as deployable single-purpose processes. For example, you'd build an 'app' container that is very similar to an 'app' VM you'd create along with the init, cron, ssh, etc processes within it. Don't try to capture every process in its own container with a separate container for ssh, cron, app, web server, etc.

© 2015 Matt Jaynes

There are great theoretical arguments for having a process per container, but in practice, it's a bit of a nightmare to actually manage. There is significant overhead and maintenance involved in every container you manage, so just putting everything in it's own container significantly **increases** your management costs. Perhaps at extremely large scales that approach makes more sense, but for most systems, you'll want role-based containers (app, db, redis, etc).

If you're still not convinced on that point, read this post on microservices which points out many of the similar management problems: Microservices - Not A Free Lunch!

Misconception: If I use Docker then I don't need a CM tool!

This is partially true. You may not need the configuration management as much for your servers with Docker, but you absolutely need an orchestration tool in order to provision, deploy, and manage your servers with Docker running on them.

This is where Ansible really shines. Ansible is primarily an orchestration tool that also happens to be able to do configuration management. That means you can use Ansible for all the necessary steps to provision your host servers, deploy and manage Docker containers, and manage the networking, etc.

So, if you decide you want to use Docker in production, the prerequisite is to at least learn Ansible. There are many other orchestration tools (some even specifically for Docker), but none of them come close to Ansible's simplicity, low learning curve, and power. It's better to just learn one orchestration tool well than to pick a less powerful tool that won't do everything you need it to (then you'd end up having to learn more tools to cover the shortfalls).

© 2015 Matt Jaynes

Misconception: I should use Docker right now!

I see too many folks trying to use Docker prematurely. Your systems need to already be in fine working order before you even consider using Docker in production.

Your current systems should have:

- **secured least-privilege access (key based logins, firewalls, fail2ban, etc)**
- **restorable secure off-site database backups**
- **automated system setup (using Ansible, Puppet, etc)**
- **automated deploys**
- **automated provisioning**
- **monitoring of all critical services**
- **and more (documentation, etc)**

If you have critical holes in your infrastructure, you should not be considering Docker. It'd be like parking a Ferrari full of adorable puppies on the edge of an unstable cliff.

Docker is a great optimization - but it needs a firm foundation to live on.

Misconception: I have to use Docker in order to get these speed and consistency advantages!

Below I list some optimizations that you can use instead of Docker to get close to the same level of performance and consistency. In fact, most high-scale companies optimize their systems in at least some of these ways.

© 2015 Matt Jaynes

CM Tools

If your systems are automated, it allows you to easily create and manage them. Particularly in the cloud, it's cheap and easy to create and destroy server instances.

Cloud Images

Many cloud server providers have some capability to save a server configuration as an image. Creating a new server instance from an image is usually far faster than using a CM tool to configure it from scratch.

One approach is to use your CM tool to create base images for your server roles (app, db, cache, etc). Then when you bring up new servers from those images, you can verify and manage them with your CM tool.

When small changes are needed to your servers, you can just use your CM tool to manage those changes. Over time the images will diverge from your current server configurations, so periodically you would create new server images to keep them closer aligned.

This is a variant of the Golden Image pattern that allows you to have the speed of using images, but helps you avoid the tedious image re-creation problem for small changes.

Version Pinning

Most of the breakages that occur from environment to environment are due to software version differences. So, to gain close-to-the-same consistency advantages of Docker, explicitly define (pin) all the versions of all your key software. For example, in your CM tool, don't just install 'nginx' - install 'nginx version 1.4.6-1ubuntu3'.

© 2015 Matt Jaynes

If you're using Ansible, it's trivially easy to install your development environment in Vagrant using the same scripts that you use to install production. If you make sure you're also using the same OS version (like Ubuntu 14.04 x64, etc) across all your environments, then you will have highly consistent systems and breakages between environments will be very rare.

Version Control Deploys

If you use git (or a similar version control system), then you can use that to cache your application code on your servers and update it with very minimal downloads. This is similar to Docker's image layer caching. For example, if your codebase is 50MB and you want to deploy an update to your code which only involves a few changed lines in a couple of files, then if you just update the code on the server via git (or similar) it will only download those small changes in order to update the codebase. This can make for very fast deploys.

Note: You don't even have to use a version control system necessarily for these speed advantages. Tools like `rsync` would also allow you to essentially have most of your code cached on your servers and deploy code changes via delta updates which are very light and fast.

Packaged Deploys (of application code primarily)

If your software deploys require a time-consuming step like compiling/minifying CSS and Javascript assets, then consider pre-compiling and packaging the code to deploy. This can be as simple as creating a *.zip* <*spann class*="">> file of the code and deploying it that way. Another option would be to use an actual package manager like `dpkg` or `rpm` to manage the deploys.

If you're using git (or another version control system), then you could even have a repository (the same or separate from the code) just for the compiled assets and use that.

For greater speed, make sure that the package (in whatever form) is on the same network local to your servers. Being on the same network is sometimes only a minor speed-up, so only consider it if you have a bottleneck downloading resources outside the servers' network.

When to use Docker

For multi-host production use, I would recommend using the alternative optimization methods I mentioned above for as long as you can. If you reach a point in the scale of your servers where those methods aren't enough, then consider using Docker for the advanced optimizations it provides. Currently, you'll need to be at very large scale before the benefits of using Docker outweigh the costs of the added complexity it adds to your systems. This may change in the coming years as Docker and the tools and patterns around it mature.

Of course, this recommendation assumes that your systems are already robust and fully covered as far as being scripted, automated, secured, backed up, monitored, etc.

Conclusion

Docker is a great project and represents a step forward for some systems scenarios. It's powerful and has many use cases beyond what I've discussed here. My focus for evaluating Docker has been on server setups delivering web applications, however, there are other setups where my advice above won't be as relevant.

© 2015 Matt Jaynes

It's a very new set of optimizations that in the right situations, is a big step forward. But, remember that using and managing it becomes complex very quickly beyond the tiny examples that are shown in most articles promoting Docker.

Docker is progressing at a good pace, so some of my advice will eventually be out of date. I'd very much like to see the complexity go down and the usability go up for multi-host production Docker use. Until then, be sure to adequately weigh the cost of using Docker against the perceived benefits.

© 2015 Matt Jaynes

Addendum 1: Tips for sysadmins that want to use Docker in production

If you're an expert system administrator and your systems are already at the scale where Docker's cost/benefit trade-off makes sense, then consider these suggestions to help simplify getting started:

You don't need to Dockerize everything

Use Docker only for the server roles that will benefit from it. For example, perhaps you have thousands of app servers and you need Docker to optimize app deploys. In that case, only Dockerize the app servers and continue to manage other servers as they are.

Use role based Docker images

I mentioned this earlier in the chapter, but just to reiterate, it will be far easier to manage Docker if you use it for roles like app, db, cache, etc rather than individual processes (sshd, nginx, etc).

You will generally already have your servers scripted and managed by roles, so it will make the Dockerization process much simpler.

Also, if you are at scale, you will nearly always only have one role per server (an app server is only an app server, not also a database server) and that means only one Docker container per server. One container per server simplifies networking greatly (no worry of port conflicts, etc).

© 2015 Matt Jaynes

Be explicit (avoid magic) as long as possible

Docker will assign random ports to access services on your containers unless you specify them explicitly. There are certain scenarios where this is useful (avoiding port-conflicts with multiple containers on the same host), but it's far simpler and easier to manage if you stick with one role container (app, db, cache, etc) per host server. If you do that, then you can assign explicit port numbers and not have to mess with the complexity of trying to communicate random port numbers to other servers that need to access them.

There are tools like etcd, zookeeper, serf, etc that provide service discovery for your systems. Rather than hard-coding the location of your servers (ex: the database is at database.example.org), your application can query a service discovery app like these for the location of your various servers. Service discovery is very useful when you get to very large scales and are using auto-scaling. In those cases it becomes too costly and problematic to manage hard-coded service locations. However, service discovery apps introduce more complexity, magic, and point of failures, so don't use them unless you absolutely need to. Instead, explicitly define your servers in your configurations for as long as you can. This is trivial to do using something like the inventory variables in Ansible templates.

Don't store data in containers

Unless you really know what you're doing, don't store data in Docker containers. If you're not careful and stop a running container, that data may be lost forever. It's safer and easier to manage your data if you store it directly on the host with a shared directory.

For logs, you can either use a shared directory with the host or use a remote log collection service like logstash or papertrail.

© 2015 Matt Jaynes

For user uploads, use dedicated storage servers or a file storage service like Amazon's S3 or Google's Cloud Storage.

Yes, there are ways to store data in data-only containers that may not even be running, but unless you have a very high level of confidence, just store the data on the host server with a shared directory or somewhere off-server.

Use a paid private index provider

It's a chore to correctly set up a self-hosted secure private Docker index yourself. You can get going much quicker by using a hosted private Docker index provider instead.

Hosted Private Repositories

Docker does provide an image for hosting your own repositories, but it's yet another piece to manage and there are quite a few decisions that you'd need to make when setting up. You're probably better off starting with a hosted repository index unless your images contain very sensitive baked-in configurations (like database passwords, etc). Of course, you shouldn't have sensitive data baked into your app or your Docker images in the first place - instead use a more sane approach like having Ansible set those sensitive details as environment variables when you run the Docker containers.

Build on the expertise of others

Phusion (the company that makes the excellent Passenger web server) has built advanced Docker images that you can use as a base for your services. They have spent a lot of time solving many of the common problems that you'd experience when attempting to create role-based Docker images. Their documentation is excellent and can serve as a great starting point for your own images.

© 2015 Matt Jaynes

https://phusion.github.io/baseimage-docker/

https://github.com/phusion/baseimage-docker

https://github.com/phusion/passenger-docker

https://github.com/phusion/open-vagrant-boxes

Addendum 2: Tiny example with Docker

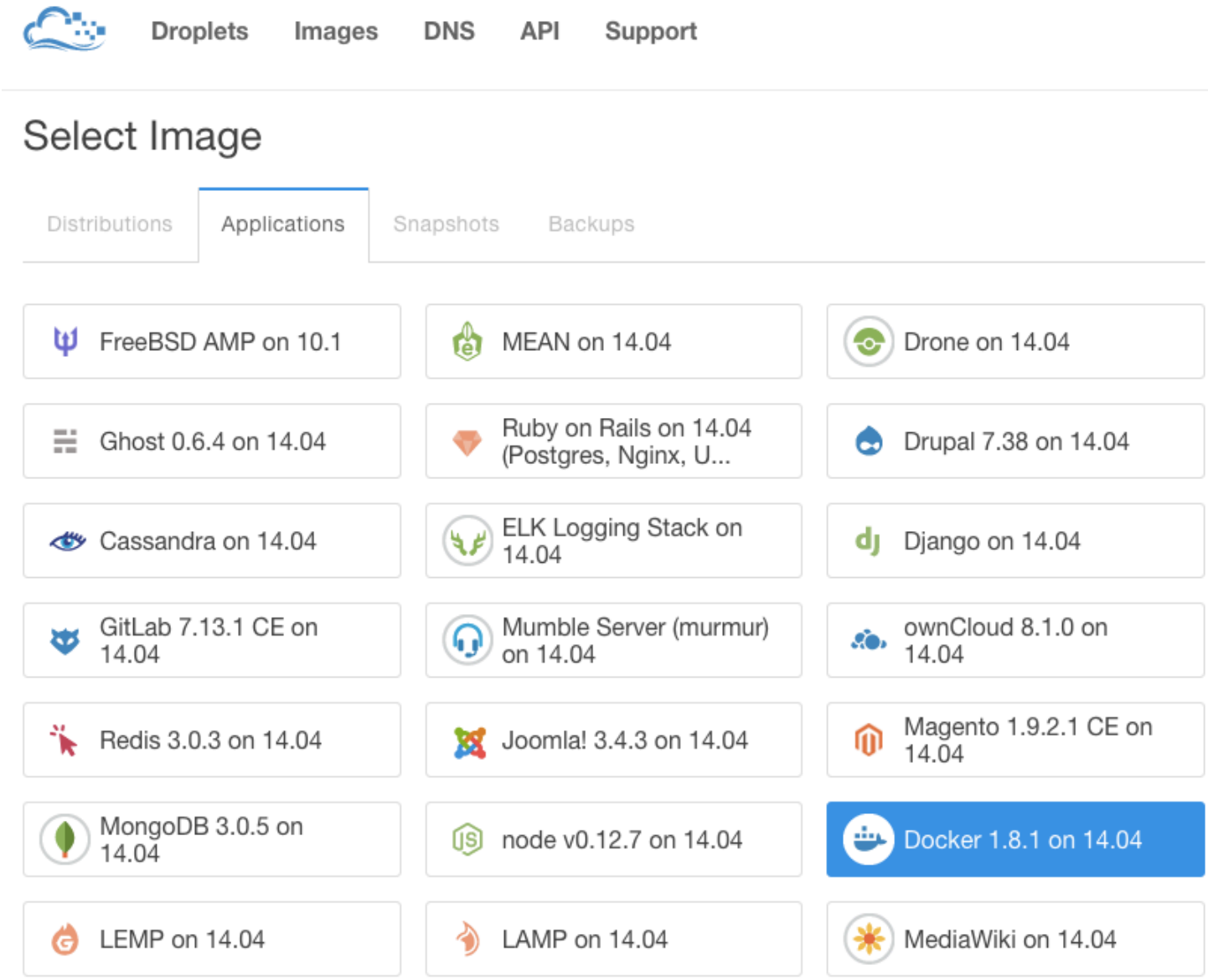
I'm including this example so that you can see how we would use Docker in a very minimalist way to set up a server like we did for the other CM tools. Remember though that this example (and many examples like this), are greatly misleading about the complexity involved for setting up Docker for a multi-host production environment. Examples like this skip all the hard parts and just show the easy parts of Docker.

It's out of the scope of this book to show a full production-level example of multi-host Docker. To do so would take a full book or course to do properly. However, I don't want to leave you without at least an example of Docker being used to set up the example project we've used throughout this book.

Set Up Server

To implement this example, first follow the instructions in the Setup chapter, but only set up the `puppy.dev` server. Instead of using 'Ubuntu 14.04 x64' for the server, DigitalOcean already provides a server image with Docker pre-installed. At the time of this writing, it's called 'Docker 1.8.1 on Ubuntu 14.04 x64', but the version numbers will likely be higher when you read this.

© 2015 Matt Jaynes



If you'd rather install Docker manually, follow the instructions here: How To Install and Use Docker

Trimmed Example

We're only going to set up the puppy.dev server below. Setting up the kitty.dev server is nearly identical.

© 2015 Matt Jaynes

I'm skipping some of the steps (like user creation and file ownership permissions) to keep this as short as possible, but it's easy to guess how you'd do the other steps since Docker just uses the same shell commands that you'd use for setting this up manually like in the Shell Script chapter.

Docker Index of Images

Docker, Inc (the company) provides a public image repository at https://hub.docker.com/ with shared public container images of common software setups.

For this tiny project, we'll be using the official nginx image.

Here is the Dockerfile that builds this public image:

```
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8FBC641079A6ABABF5
BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" >> /etc/apt/source
s.list

ENV NGINX_VERSION 1.9.4-1~jessie

RUN apt-get update && \
    apt-get install -y ca-certificates nginx=${NGINX_VERSION} && \
    rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

VOLUME ["/var/cache/nginx"]

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

© 2015 Matt Jaynes

You should have already launched your puppy.dev server, so go ahead and log into it and create these directories and files on the server. These will later be shared with the Docker container when we run it.

```
root@puppy:~# mkdir /root/data
root@puppy:~# touch /root/data/index.html
```

Add this to `index.html`:

```
<html>
  <body bgcolor="gray">
    <center>
      <img src="/puppy.jpg">
    </center>
  </body>
</html>
```

Then download the puppy image into the `/root/data` directory:

```
root@puppy:~# wget --directory-prefix=/root/data https://raw.github.com/nanobeep/tt/maste
r/puppy.jpg
```

Then start the Docker container and tell it what ports to use and how to map the shared directory:

```
docker run -d -p 80:80 -v /root/data:/usr/share/nginx/html nginx
```

You'll notice that we're mapping `/root/data/` with the default document root for nginx which is `/usr/share/nginx/html` so that nginx can serve the html and puppy image from within the Docker container.

Now if you go to http://puppy.dev you should see the puppy site.

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

CM Tool Security

Security is a tough topic to cover. Why? Because it's hard to get a good basis of comparison for each tool. Then when you finally get a grip on the security record and practices for each tool, it's ultimately a subjective decision on how to compare them.

There's no such thing as "perfect" security in the practical world. There's a range of security from zero security (like a bicycle left unattended on the street) to "pretty good" security (like a bicycle locked in an apartment).

Security is not free. It takes time and adds complexity overhead to your systems. How much you invest in security depends on what you are securing.

If you're a bank with billions of dollars at stake, you'll go to great lengths to secure access to your systems. But if you're a one-man business running a free to-do list app, then your security needs will be far less.

For systems management tools like Puppet, Chef, SaltStack, and Ansible, we need to set the bar pretty high. While they can be used for throw-away play projects, they are also used to support multi-billion dollar businesses as well.

If these tools are compromised, then the bad guys can access your systems and either silently surveil you and your customers, or wreak apocalyptic havoc on your business.

Now I'll walk through how I evaluated these CM tool's security and why I rated them the way I did. I'll also give you links to the security resources for each tool so that you can make your own decision if you disagree with my assessment.

© 2015 Matt Jaynes

The main things I'll be considering are:

- **Reporting Transparency**
- **Attack Surface**
- **Security Record (since 2012)**

Reporting Transparency

If there's a security issue with versions of a tool, is it easy to find out about it?

If you aren't readily informed about the security of a tool, then it's far less likely you'll take timely action to remedy the security issues that come up.

As of May 2014, only Puppet had a dedicated security page. As of August 2015, Ansible, SaltStack, and Chef have all added security pages.

The security pages for Puppet and Ansible inform you about their past security vulnerabilities so that you can easily see what patches or upgrades you will need to apply. Chef and SaltStack unfortunately don't publicly track their security vulnerabilities on their security page and instead ask the users to just follow the mailing list (Salt) or blog (Chef) in order to discover vulnerabilities.

The most vigilant reporters may look less secure, but might not be

If you look at Puppet's security page, you will see a very long list of security disclosures. If you try to find Chef's disclosures, you won't find many (unless you are very persistent), but that's largely due to their iffy

© 2015 Matt Jaynes

reporting practices. So if you compare the number of security notices from Chef, it may seem like Chef is more secure - but that may not be the case - it may just be poor reporting.

A good example of this problem is the discussion of the Heartbleed OpenSSL vulnerability. Both Puppet and Chef addressed the vulnerability for their web interface tools and provided fixes publicly. However, SaltStack and Ansible were silent (as far as I can find) about the vulnerability in their own web interface tools, (Ansible Tower and SaltStack Halite), yet they found the time to put out promotional articles about how their tool can fix the issue:

Some Salt for that Heartbleed

Fixing Heartbleed with Ansible

You might then assume that only Puppet and Chef had vulnerabilities to Heartbleed. However, you'd likely be wrong. Default installs of Ansible Tower and SaltStack Halite on many systems used (and still use as far as I can tell) OpenSSL for their SSL capabilities and so installations of these would probably have been vulnerable to Heartbleed. Perhaps Ansible and SaltStack contacted users privately, but an issue as serious as Heartbleed should have been addressed for these products publicly.

Security Reporting Ranking

1. **Puppet / Ansible (tie)**
2. **Chef / SaltStack (tie)**

Attack Surface

The attack surface or 'attack vector' is the figurative 'surface area' that is available for an attacker to attack. For example, it's much easier to hit a 10-foot bullseye with an arrow than a tiny 1-inch bullseye. Similarly, if a system is large and made up of many parts, it increases the opportunities for attack. The smaller and simpler a system is, the smaller its attack surface and so there are fewer opportunities of attack for a hacker.

For CM tools, the main attack surfaces I'm considering are:

- **Network Connectivity**
- **Software Dependencies**

Network Connectivity

When you look through the security record of the tools, the network connection is often a key attack vulnerability.

Except for Ansible, all of the tools use a master-child network setup by default with either a persistent or periodically established encrypted network connection. Ansible uses SSH and is run as needed from a control machine (which is generally the engineer's local machine, like their laptop).

Puppet and Chef use SSL for their network encryption. Some users of Puppet and Chef used OpenSSL for the SSL connection and were then vulnerable to Heartbleed.

Salt has implemented its own network encryption, which has led to major vulnerabilities in the past. However, this decision also saved it from exposure to the Heartbleed vulnerability for its core tool (not Halite).

© 2015 Matt Jaynes

Ansible has far less frequent network connections and uses SSH by default, which is not perfect (no tool is!), but is generally considered to be one of the most secure and extensively audited secure networking tools available.

In default configurations, Puppet, Chef, and SaltStack don't use SSH directly, however, the servers they manage nearly always have SSH running as well in order to support other management tasks. So they not only have their default network connections as an attack vector, but usually also have SSH as an attack vector as well. Ansible only has SSH as an attack vector and establishes its connections far less frequently generally than the other tools.

So for network attack surface, Ansible wins with the smallest attack surface.

Software Dependencies

Puppet and Chef are very very heavy applications that utilize heavy frameworks and many third-party dependencies that frequently have their own vulnerabilities (sometimes very severe) that must be patched.

Salt is very light regarding its dependencies. ZeroMQ is the main dependency and has a strong security track-record.

Ansible is the lightest of all and depends on very little other than SSH.

Ease of Fixing Vulnerabilities

An important note about larger attack surfaces is that they also require more maintenance and patches which can lead to unpatched systems if the admin isn't vigilant about keeping the software up-to-date.

© 2015 Matt Jaynes

A larger attack surface usually means higher security maintenance needs as well. For tools like Puppet and Chef that are very large applications with many dependencies, it's an extra challenge to also manage the timely patching of their multitude of various dependencies with any consistency. If these security patches aren't applied quickly, your systems will be vulnerable.

With higher security maintenance needs, you'll have the additional danger of those maintenance tasks being forgotten and your systems becoming less and less secure as vulnerabilities go unpatched.

Attack Surface Ranking

(less attack surface ranked higher)

1. **Ansible**
2. **Salt**
3. **Puppet**
4. **Chef**

Security Record (since January 2012)

For the security record, I'm focusing primarily on major recent vulnerabilities (past ~2 years). Also, I'm looking primarily at the open source versions of the main tool (and largely ignoring the secondary web interface products).

© 2015 Matt Jaynes

Puppet

Probably the most serious vulnerability was Heartbleed due to Puppet's reliance on OpenSSL in some instances. This vulnerability wasn't actually a part of Puppet itself, but was caused because the vulnerable OpenSSL libraries were used by Puppet and were therefore part of its attack surface.

The other more serious vulnerabilities were privilege-escalation issues, which would only affect you if you had untrusted users on your Puppet servers (unlikely for most systems).

As mentioned earlier, Puppet is the most vigilant in reporting their vulnerabilities in an easily discoverable way, so this assessment is easier to trust.

To track security, follow their Security page, but also follow their blog since sometimes major (like Heartbleed) vulnerabilities don't make it to their Security page. You'll also want to subscribe to their mailing list.

Chef

Chef's security reporting is now done through their blog with any post that is in their 'security' category, so it's a bit tricky to get a handle on. It's also hard to rely on since their posts are often not properly tagged and sometimes you'll end up missing critical security updates.

Chef's lackluster security response procedures were demonstrated by their fumbled response to Heartbleed:

April 8th: OpenSSL Heartbleed Security Update

April 10th: Update on Heartbleed and Chef Keys

April 11th: Postmortem: Chef Client Regressions and Heartbleed

© 2015 Matt Jaynes

Chef's most serious security vulnerability (that I could find) was due to Heartbleed.

Chef has also had a couple of significant data breach incidents on their sites:

Security Breach: User Information Compromised

Hosted Chef Data Leak

Again, it's difficult to assess Chef's security because of their odd security reporting practices, though it seems they've improved a little in the last year.

SaltStack

SaltStack had a few security issues with its alpha salt-ssh tool, but those were quickly fixed and probably didn't affect any production users.

The other major vulnerability they had was a privilege-escalation issue which would only have affected those who had untrusted users on their systems.

Salt version 0.15.1 deserves a special mention due to the number of extremely serious vulnerabilities that it solved, the most serious of which was a flaw in the way they were handling RSA key generation.

SaltStack's custom network encryption is a mixed bag. Generally it's a very bad idea to invent your own crypto, and the RSA public exponent issue was evidence of that. This vulnerability caused extremely weak keys to be generated with highly predictable values which made it trivial to compromise the encryption keys protecting Master and Minion (child server) communication. The vulnerability also made it easy to impersonate a Salt Master or Minion. Fortunately, the vulnerability was discovered in a SaltStack-commissioned audit, and their rapid fix and

disclosure protected most users from harm. However, as far as revealing private keys is concerned, the vulnerability in their handwoven AES implementation was nearly as serious as Heartbleed. However, by not using OpenSSL, they were not vulnerable to Heartbleed in their core product when that problem surfaced.

SaltStack has improved this year by adding a dedicated Security page, but rather than collect and report their security issues publicly, they ask users to just monitor their mailing list.

Ansible

No major vulnerabilities found. Security fixes released quickly and announced to the mailing list and made available on their Security page. Because Ansible does not run persistent daemons on the servers it manages, none of the vulnerabilities were remotely exploitable by an attacker who lacked access to the control machine.

Security Record Ranking

1. **Ansible**
2. **SaltStack**
3. **Puppet**
4. **Chef**

Note that I put Chef at the bottom not necessarily because it had more vulnerabilities than Puppet or SaltStack, but because their reporting is so hard to assess and it makes it very difficult to judge their security record, though from what I've seen in my research, it would still be at the bottom of the list.

© 2015 Matt Jaynes

In the 3rd edition of this book, I've put SaltStack ahead of Puppet (it was the other way around in the 2nd edition). Though SaltStack doesn't have the level of security reporting that Puppet does, its number of security incidents over the last year are almost non-existent and I'd argue that qualifies it as a more secure and reliable solution, especially since they've added much more robust security reporting procedures.

Conclusion

Overall Ansible is the clear winner for security. However, Puppet deserves praise for how seriously they take reporting and resolving security issues. SaltStack has also had a great year for avoiding security incidents and improving their security procedures.

Recommendations

Choose Ansible if you want low-maintenance and high-security.

Be extremely wary of using SaltStack, Puppet, or Chef in master-child setups unless you're on a secured private network. Naturally, Ansible would also benefit from being on a secured private network, but because it uses SSH, it is generally considered safe to use on public networks. SaltStack also is runnable on SSH, so that's also an option if you are on a non-private network.

If you want to use SaltStack, Puppet, or Chef outside of a secured private network, then I'd recommend against using their default master-child connectivity. Instead, consider using them only on the client servers and use Ansible to orchestrate configuration deploys and runs. For example, use chef-solo (or chef zero) on a server, but have Ansible deploy the Chef recipes and run them. This is how Rackspace and Atlassian currently manage their servers with Puppet - they use Ansible to deploy and then run the Puppet configs.

© 2015 Matt Jaynes

Never use a web-interface for any of these tools unless it is on a secured private network that is only accessible via VPN or another secure method, such as running the interface behind spiped or SSH tunnels so all connections to it are encrypted and authenticated. Web interfaces are nearly always dramatically less secure than the core product - most allow password logins and are exposed to the whole internet - the horror!

To reiterate, for low-cost maintainable security, use Ansible (via key-based SSH) for connecting to your servers unless you have a secured private network.

© 2015 Matt Jaynes

Appendix: Security

This is a somewhat subjective topic. Different people have different priorities for security and yours may be different than mine. If for any reason you found my assessment lacking, please feel free to research these issues for yourself. To give you a head start, here are some resources for each of the tools:

Puppet

Security Page: https://puppetlabs.com/security

CVE: http://www.cvedetails.com/vulnerability-list/vendor_id-11614/product_id-21397/Puppetlabs-Puppet.html

Blog: https://puppetlabs.com/blog

Issue Tracking: https://tickets.puppetlabs.com/browse/PUP/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel

Mailing Lists: https://puppetlabs.com/community/get-help#Mailing-lists

Chef

Security Page: https://www.chef.io/security/

CVE: http://www.cvedetails.com/vulnerability-list/vendor_id-12095/product_id-22765/Opscode-Chef.html (only 3 vulnerabilities showing here since 2012, despite there being far more than this and no other place to apparently find them other than hunting through their blog and bug tracker)

Blog: https://www.chef.io/blog/

© 2015 Matt Jaynes

Issue Tracking: https://tickets.opscode.com/browse/CHEF#selectedTab=com.atlassian.jira.plugin.system.project%3Aissues-panel

Mailing Lists: http://lists.opscode.com/sympa (all links broken when checked Aug 22, 2015)

Salt

Security Site: http://docs.saltstack.com/en/latest/security/index.html

CVE: http://www.cvedetails.com/vulnerability-list/vendor_id-12943/Saltstack.html

Blog: http://saltstack.com/blog/

Issue Tracking: https://github.com/saltstack/salt/issues

Mailing List: https://groups.google.com/forum/#!forum/salt-announce

Ansible

Security Site: http://www.ansible.com/security

CVE: http://www.cvedetails.com/vulnerability-list/vendor_id-12854/product_id-26114/Ansibleworks-Ansible.html

Blog: http://www.ansible.com/blog

Issue Tracking: https://github.com/ansible/ansible/issues

Mailing Lists: http://docs.ansible.com/ansible/community.html#mailing-list-information

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

CM Tool Communities

Community is another tough topic because it's very subjective and the community numbers can be interpreted different ways.

I've spent time examining all of the communities - on mailing lists, IRC, forums, etc. Generally you'll find each CM tool's community helpful and welcoming. Every community has its friendly folks that are happy to assist newcomers, and every community has its grumpy engineers that are kind of curmudgeons. The curmudgeons are fortunately in the minority, but don't be surprised when you encounter them. Generally they mean well, but just aren't suited for smooth interactions with other humans.

The only real trend I've been able to see is that the newer communities like Ansible and SaltStack have a closer-knit feel and seem more responsive. That's not too surprising since they are smaller communities and in an active growth stage. Puppet and Chef have larger more mature communities, so they have bigger events and a more enterprisey-feel (which makes sense since their main source of revenue is likely from enterprise support contracts).

Caveats

It's important to note that just because one community is friendlier doesn't mean that its tool is better - sometimes it's the opposite. A community that says "no" a lot can sometimes be much better at keeping the tool simple and focused. A friendlier community can end up saying "yes" too often and end up with a bloated less-secure tool.

© 2015 Matt Jaynes

Also, a less active community may mean that the tool is more mature and has already solved the core use-cases. So, just because a community is more active doesn't necessarily mean it's better, it can also mean that it is less mature and still in a growth state.

Comparing

I'll give my interpretation so you have something to start with, but ultimately what matters is that the community produces a usable, secure tool for you to use.

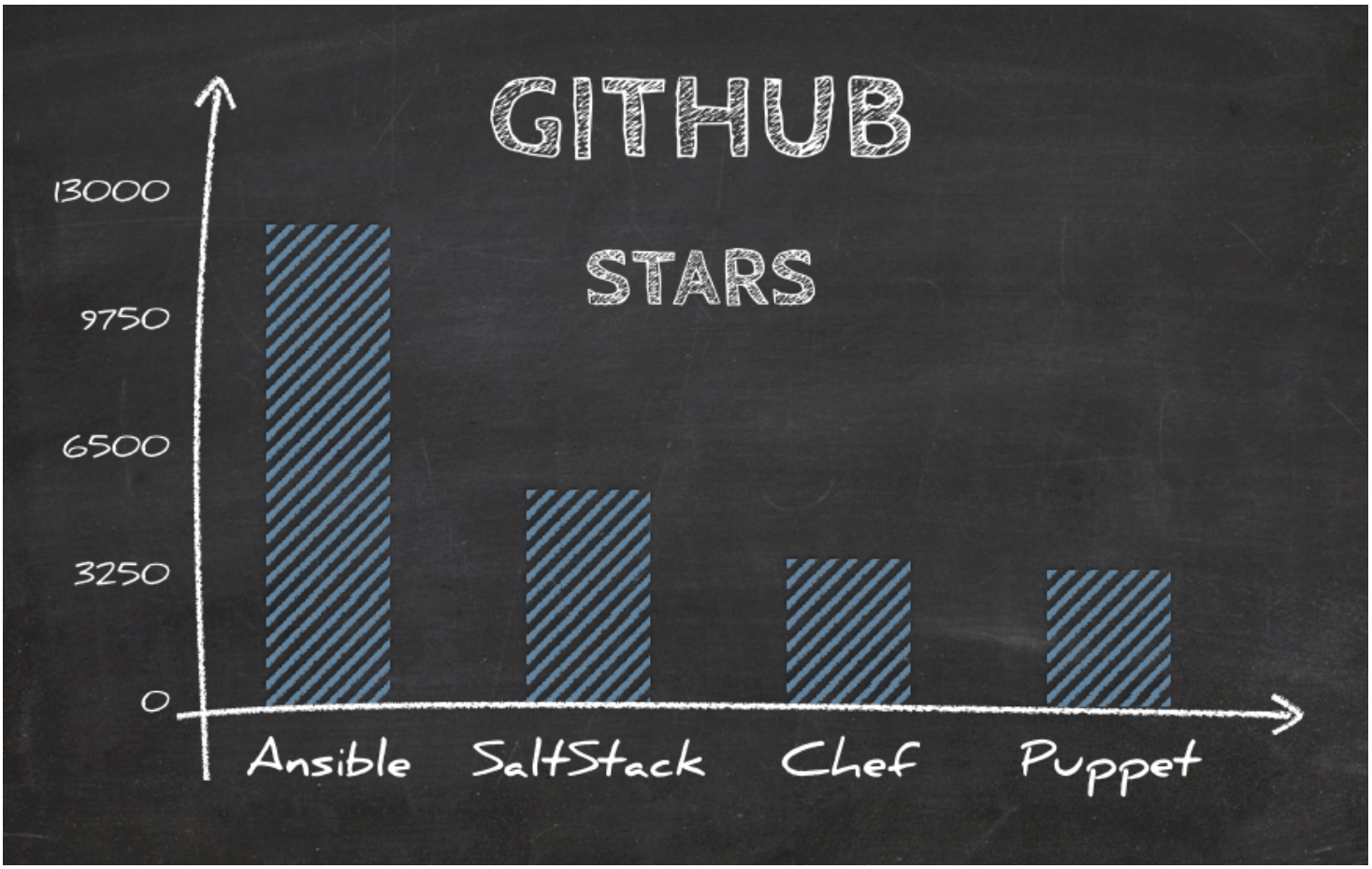
All of these CM tools have active communities. Naturally, Puppet and Chef which have been around the longest have the largest communities, but SaltStack and Ansible in particular are rapidly gaining ground in community size.

There were quite a few metrics I could have explored, but most of them gave poor data for at least one of the CM tools and so didn't work well for a comparison. Complicating matters is the fact that terms like "chef", "puppet", and "salt" are used in many other contexts (puppet shows, chef's cooking, salting passwords, etc) and so it's hard to get good metrics on how the tools trend in "mentions" on discussion sites, etc.

I ended up choosing just a few metrics that seemed the most reliable for getting a sense of the activity of the community. I realize these aren't great metrics to use, but these are the most reliable I could find. Other metrics like downloads, installs, term mentions, etc were just so unreliable that to include them would be misleading at best. Hopefully this will give you a tiny sense of the scale and activity of these communities. *(Data gathered on August 22, 2015)...*

© 2015 Matt Jaynes

Github Stars (for their primary repository)

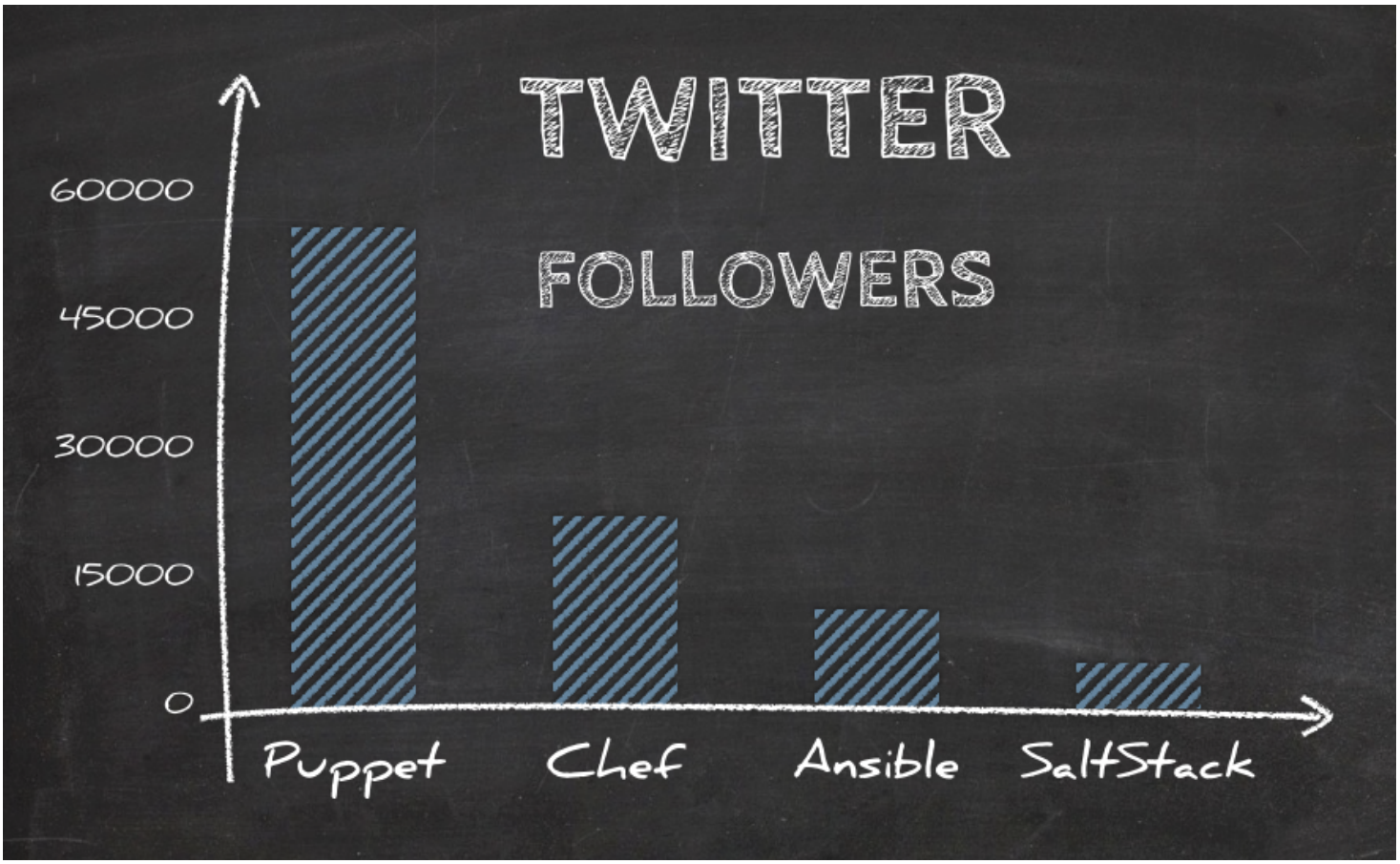


Difference in stars since last year are in brackets…

- **Ansible: 12,310 [+6,370]**
- **SaltStack: 5,550 [+2,130]**
- **Chef: 3,785 [+1,097]**
- **Puppet: 3,498 [+1,326]**

© 2015 Matt Jaynes

Twitter



Difference in followers since laste year are in brackets…

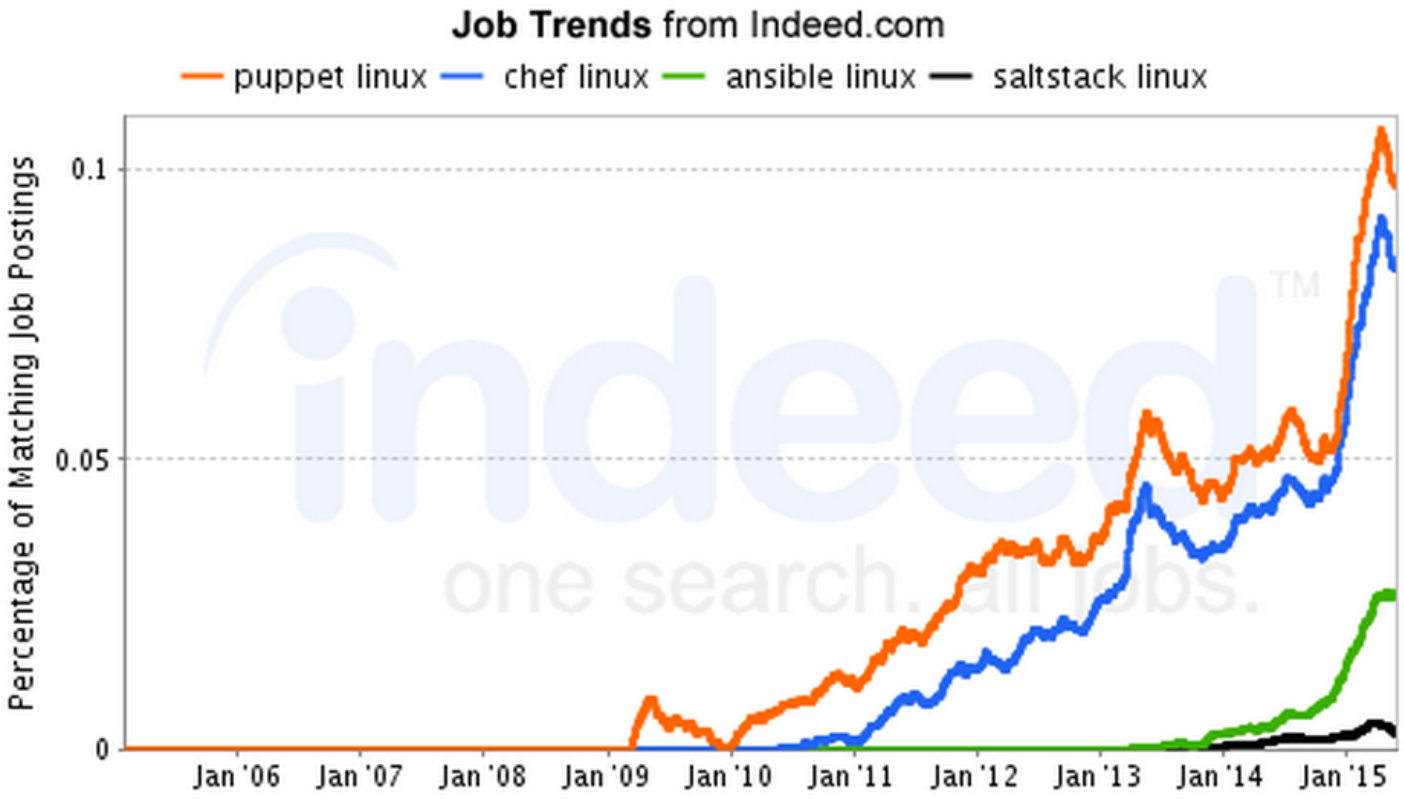
- **Puppet: ~56,500 [+6,700]**
- **Chef: ~22,700 [+11,200]**
- **Ansible: 11,800 [+8,240]**
- **SaltStack: 5,455 [+3,180]**

© 2015 Matt Jaynes

Indeed Job Trends

Note: *Since some of the terms would match other non-systems jobs ("chef" cough cough), I've added the term 'linux' to each of the terms to ensure that we only get results for actual systems jobs mentioning these tools.*

Job Listings

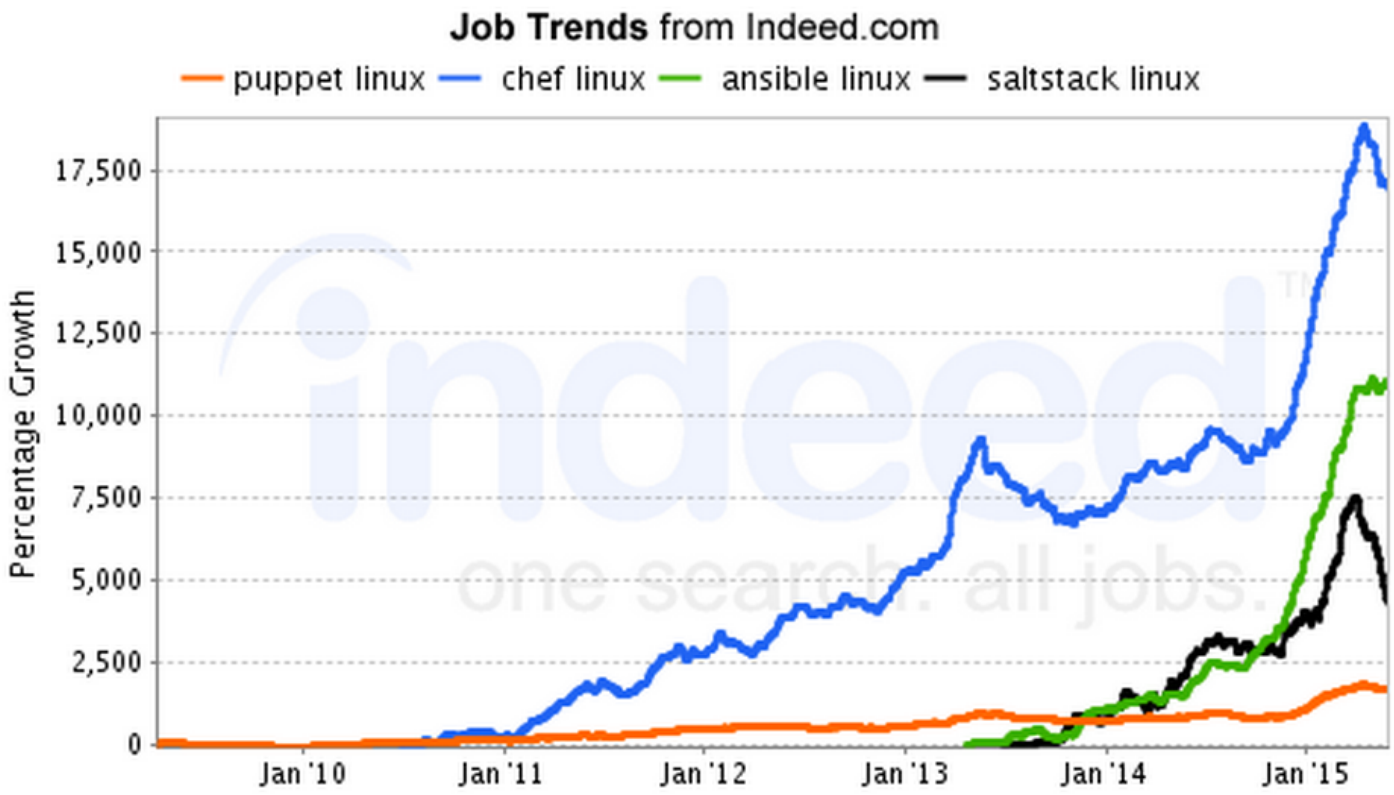


You can see that there was recently a huge surge in jobs for most of the tools except SaltStack.

© 2015 Matt Jaynes

Job Growth

Now, let's look at the growth rates:



This graph may be more informative. We see strong growth of Ansible, SaltStack, and Chef. However, SaltStack and Chef seem to have their growth slowing recently.

Conclusion

You can see that the younger tools Ansible and SaltStack are more popular (in Stars) on Github, but they're still catching up a bit in terms of social followers and jobs.

When I wrote the first edition of this book in the summer of 2013, Ansible and SaltStack seemed to be about equal in most of these metrics, but Ansible seems to have picked up quite a bit of steam since then.

© 2015 Matt Jaynes

Puppet (2005) and Chef (2008) have been around for many more years and so have larger followings. You can see though that their popularity may be showing some slowing according to some metrics.

I've participated in all of the communities at one time or another. My general impression is that they're full of excellent people and the communities are all pretty friendly and ready to help newcomers. I've noticed that it's generally easier to get feedback from the Ansible and SaltStack communities now than the Puppet and Chef communities. My guess is that that is due to Ansible and SaltStack still actively trying to grow, while Puppet and Chef are more mature and primarily focused on their paying enterprise customers now. Of course, your experience and observations may differ from mine, so it's fair to be very skeptical of my assessments here!

© 2015 Matt Jaynes

Appendix: Community

The stats I mention in this chapter will probably be a bit out-of-date when you read this. To see current stats on the communities, see these links:

Indeed Job Trends

Job Listings / Job Growth

Puppet

Github / Twitter

Chef

Github / Twitter

SaltStack

Github / Twitter

Ansible

Github / Twitter

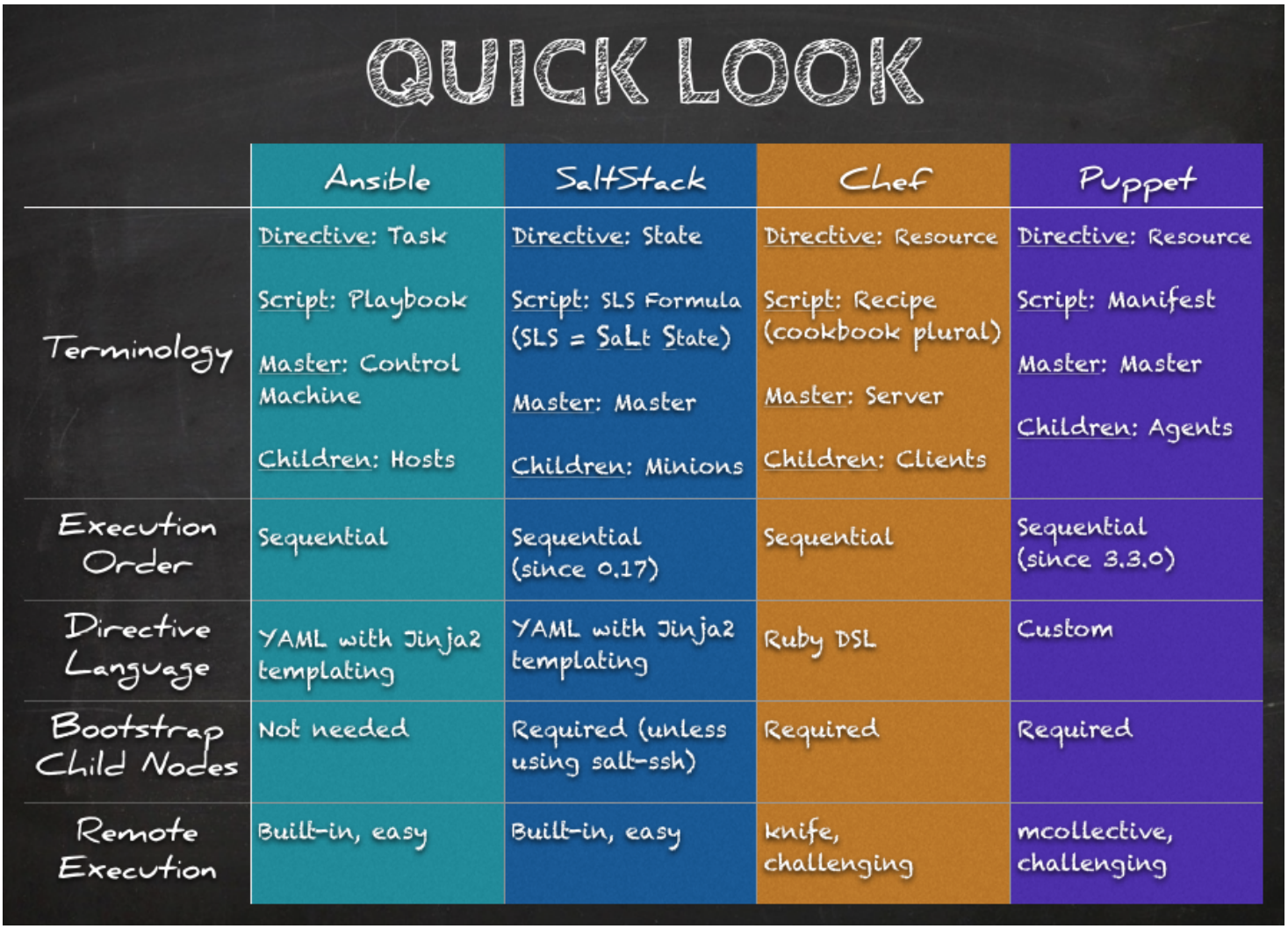
© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes

Conclusion



New Generation

I recommended Puppet and Chef for many years, but Ansible is just so simple and powerful that I have to continue recommending it now - especially to anyone just getting started with configuration management.

Salt is another good contender you may also consider, but it has a higher learning curve and a seemingly smaller community so you'll want to consider whether its feature-set is worth those trade-offs.

© 2015 Matt Jaynes

It's important to remember that Ansible and Salt owe their success in large part to the previous work done by earlier CM tools like Puppet and Chef. The creators of these new tools had a great advantage in seeing how previous CM tools did things and they've taken that knowledge and have built superior options.

That's a big win for you.

Choosing

Hopefully now you have a good idea of which CM tool you want to use.

If you're still undecided, you've at least been able to narrow the field. Now you can go and explore your finalist tools in more depth.

Remember too, that if you like a tool, but are concerned about its master-child networking security, then you can use Ansible to distribute and manage the 'solo' versions of those tools as discussed in the Security chapter.

I'd caution against taking more than a couple of days to decide. The benefits from using a CM tool are tremendous and if you choose Ansible or Salt, they are simple enough that it won't be too big of a deal if you change your mind and want to switch later. The main thing is to just get started making your systems more excellent.

Good luck!

© 2015 Matt Jaynes

Quick Nav:

- Intro
- Shell Script
- Pre Tool Setup
- Tool: Ansible
- Tool: SaltStack
- Tool: Chef
- Tool: Puppet
- Bonus: Where Docker Fits In
- Bonus: CM Tool Security
- Bonus: CM Tool Communities
- Conclusion

© 2015 Matt Jaynes