**Will Docker or Puppet work?**

When it comes time to selecting a better method, will Docker or Puppet work?  
Let me rephrase the question, can containers and configuration management co-exist? Great article from Daniel and James.

Daniel Kovac's  
http://www.icesystems.com.au/docker-puppet-island-sacred-integration/   
  
and James Turnbull.  
https://puppetlabs.com/blog/can-containers-and-configuration-management-co-exist?   
  
Both articles bring insight and samples. But Daniel brings a bit more.   
Still, I highly recommend reading both.  
  
  
First Daniel's  
Docker and Puppet on the “Island of the Sacred Integration”  
by Daniel Kovacs | Nov 27, 2014 | blogs |  
Docker and Puppet  
  
Prolog  
  
Last month I was asked by Stephen Wallace to deliver a talk on Puppet and Docker.  
  
First, I’ve spent a little time on thinking through how many ways these technologies could be used together. We could use puppet to spin up docker containers. We could use puppet to build docker images. We could use docker to test puppet manifests. We could also ‘dockerize’ the puppet master itself.  
  
All of these scenarios are worth looking into. But due to time constraints I’ve decided to implement the first two: spinning up containers and creating images.  
  
Quick introduction of the technologies  
  
Let’s go through these two technologies very briefly focusing on the features we are going to use.  
  
Puppet is a system configuration tool. It enforces states of resources defined in a puppet manifest. A resource can be anything that has a state, including files, daemons, databases and custom services. The exact implementation of how to manipulate a resource and what are the possible states of a resource are defined in modules. It runs in a master-agent (server-client) architecture, where agents periodically check in at the master and ask for the manifest.  
  
Read more: http://puppetlabs.com/puppet/what-is-puppet  
  
Docker is a container technology. It is NOT virtualisation, all containers share the same kernel. Containers are based on images. Images are pre-configured states of a container. Consider them as a tar.gz of a Linux installation, including all libraries and packages. When a process is executed inside a container, it will be using libraries and configuration inside the container. Containers are single use: once the process terminates it can’t be re-executed in the same container. (Not after docker 1.3: it introduced docker exec. But the original philosophy was that containers are immutable.) Images are layered. Basically after executing something in the container, the end-state of the container can be saved to a new image. In order to save disk space, not the whole container will be saved, but only the difference between the original image and the end-state of the container. Docker uses AUFS to simulate a full filesystem based on these layers.  
  
Read more: https://www.docker.com/whatisdocker  
  
The Treasure  
  
  
  
Sacred Integration  
  
  
  
The two technologies have one overlapping area: provisioning. Docker has something called the Dockerfile, which is a set of instructions on how-to build an image. Puppet is a provisioning tool by itself, doing the same state: putting a system into a state.  
  
But while Puppet excels at precise orchestration, it’s also very slow to execute. Docker has limited configuration power compared to Puppet, but it’s extremely fast to spin up a new container based on an existing image.  
  
The idea is to use Puppet to provision a complete service into a Docker image, then use this image on the agents to start the services.  
  
There are a lot of benefits with this approach: – Time saver: The puppet manifest that provisions a service is only required to run once, during image creation. – The same image can be used in test, staging, production and development, lowering the diversity of environments. – Portability: the image can be used in any virtualisation, in the cloud or on bare metal. – Rollbacks are always possible and are always simple: you can always spin up a previous image.  
  
The time benefit can be huge, especially with a lot of agents and long puppet manifests.  
  
The Plan  
  
Start 3 virtual machines using Vagrant, one puppet master and 2 agents.  
  
Hostname Description  
master.dev Puppet Master based on Puppet Enterprise 3.3.2  
factory.dev Puppet Agent used as the image factory  
agent.dev Puppet Agent used as the deployment target  
  
  
  
Using puppet: Install Docker on all boxes.  
Using puppet: Spin-up a local docker registry (repository) with Docker on the master.  
Manually: Execute the puppet-based image building on the first agent – let’s call it factory, – and push the image to the registry on master. See detailed below.  
Using puppet: Spin-up a container on the second agent based on the image.  
  
To test upgrade and rollback:  
  
We’ll slightly modify the puppet manifest for the container node.  
Build a new version of the image.  
Update the puppet manifest for the second agent to use the newer image.  
Execute a puppet run.  
  
The rollback is the same, but instead of increasing, we’ll decrease the image version.  
  
Setting up  
  
Installation of Docker was very easy with the excellent puppet module provided by Gareth Rushgrove (get it from here: https://github.com/garethr/garethr-docker )  
  
Let’s spin up the local registry on master:  
  
To make the local registry available on the agent’s, we need to allow http access instead of https. To do this, we’ll need to customize the installation of docker.  
  
Creating an image  
The Plan  
  
The steps to create an image are the following:  
  
Start up a minimal Centos image.  
Install puppet agent into the container.  
Change the hostname in the container so it will be detected as a different puppet node.  
Execute a puppet run – this will provision the container.  
Remove the puppet agent from the container.  
Commit and tag an image from the container.  
Push the tag to the local registry.  
  
We’ll have two scripts: one that runs on the factory, and one that runs inside the container.  
  
As docker containers are single-use, we need to create a bash script that will do steps 1-4 inside the container. It’s called container-agnet.sh in my example.  
  
The puppet node  
  
Our puppet node for the container looks like this. We’ll just install Apache and a couple static files. We also need to create a wrapper script to start Apache.  
  
The rough equivalent of the above with Dockerfile would be:  
  
  
  
Invoking the build  
  
The installation script is invoked from the command line:  
  
After it runs, we’ll have a puppet-provisioned container. We can check out what we have using docker ps:  
  
  
  
Pushing the image  
  
We need to tag the image with the full url in order to make it pushable to our local registry.  
  
And now we push it to the registry:  
  
This is it, we now have our freshly built image in the local registry.  
  
  
  
happy  
  
  
  
Spinning up a container  
  
Spinning up a container is extremely easy. We just need to declare a docker::run resource in the agent’s node.  
  
We need to apply the puppet manifest to, an the service is stared. It takes less then 1 seconds to start up a docker container.  
  
To test the container, let’s open the url with a browser:  
  
  
  
browser-agent  
  
  
  
The docker module for puppet is very well designed. It’s actually creating a new service under /etc/init.d/, and names it after the resource. So in our case we can do this:  
  
  
  
Upgrading or rolling back:  
  
When releasing a new version of the image, it does not effect the existing containers. To upgrade or roll back, we just need to change the tag version of the image in the agent’s puppet manifest:  
  
  
  
Conclusions  
  
The implementation required a couple of nasty tricks to do, but otherwise the two tools played along nicely. Here are the main advantages and disadvantages I’ve encountered:  
  
Pros   
Very fast agent provisioning   
The two tools plays along nicely   
No config drift possible thanks to stateless containers   
  
Cons  
This is a hacky way to go in the current implementation  
Not all applications are ready to be ‘dockerized’  
Only provides time benefits when dealing with large instance count of the SAME service. (i.e. distributed apps)  
  
  
  
  
Now James article.   
  
Docker is an open source framework that automates the deployment of applications in lightweight and portable containers. The Docker framework is modelled on the concept of the standard shipping containers that are used to transport much of the world’s goods. Like shipping containers you can build, fill, open and transport Docker containers. These containers can then be run in a wide variety of places: on your laptop, in the Cloud, on a virtual machine or even on physical hardware.  
  
Docker has quickly become popular for the:  
  
Automation of application packaging and deployment  
Creation of lightweight, private PAAS environments  
Automated testing and continuous integration and deployment  
Deployment and scaling of web apps, databases and backend services  
  
Since Docker was announced, however, there have been a lot of discussions about where Docker fits with configuration management tools like Puppet.  
  
I’ve spent a bit of time thinking about scenarios, images, and management tooling, and talking to people about how they use Docker, either with or without configuration management tools. I didn't learn any startling insights but I did decide that, like most aspects of the domain, there is a lot of room for a lot of tools.  
Take the DevOps survey and get a chance to win some great prizes.  
  
Docker is first and foremost an image building and management solution. One of the largest objections to the "golden image" model is that you end up with image sprawl: large numbers of (deployed) complex images in varying states of versioning. You create randomness and exacerbate entropy in your environment as your image use grows. Images also tend to be heavy and unwieldy. This often forces manual change or layers of deviation and unmanaged configuration on top of images because the underlying images lack appropriate flexibility.  
  
Compared to traditional image models Docker is a lot more lightweight: Images are layered and you can quickly iterate on them. There is some legitimate argument to suggest that these attributes alleviate many of the management problems traditional images present. It is not immediately clear, though, that this alleviation represents the ability to totally replace or supplant configuration management tools.  
  
There is amazing power and control to be gained through the idempotence and introspection that configuration management tools can provide. And Docker itself still needs to be installed, managed and deployed on a host. That host also needs to be managed. In turn Docker containers may need to be orchestrated, managed and deployed, often in conjunction with external services and tools. Configuration management tools excel at providing these capabilities.  
  
It is also apparent that Docker represents (or perhaps more accurately encourages) some different behaviors for hosts, applications and services: short-lived, disposable, and focused on single services being provided in a container.  
  
These behaviors do not lend themselves or resonate strongly with the need for configuration management tools. With these behaviors you are rarely concerned with long-term management of state, entropy is less of a concern because containers rarely live long enough for it to be, and the recreation of state may often be cheaper than the remediation of state.  
  
The most commonly cited use case is testing. Docker containers are becoming a feature of fast, agile and disposable test environments that are wired into CI tools such as Jenkins. In these use cases, a Docker container is created by a Jenkins job, configured by Docker to run the required tests and then shut down. Here, the limited lifespan of the testing host does not lend itself to running a configuration management tool and indeed running that tool could well add overhead, complexity and time to a process where every second counts.  
  
But I don't believe all infrastructure can be represented with these behaviors. Much of it can, and perhaps in the future more of it will be, but it's not exclusive and will likely exist alongside more traditional infrastructure deployment. The long-lived host—perhaps also the host that needs to run on physical hardware—still has a role in many organizations. I'm also starting to see Cloud and virtual machine consumers, especially some of those on Amazon, with long-running instances whose uptime is measured closer to the traditional physical hosts they used to operate.  
  
As a result of these diverse management needs, and combined with the need to manage Docker itself, I think we'll see both Docker and configuration management tools being deployed in the majority of organizations. Indeed I can see the potential for some incredibly powerful deployments tools that combine containers, configuration management, continuous integration, continuous delivery and service orchestration.  
  
Disclosure: I was an employee of Puppet Labs and have a financial stake in the company. I currently work at Docker Inc and am also working on a book about Docker.  
James Turnbull  
  
About the author: A former IT executive in the banking industry and author of five technology books, James has been involved in IT Operations for 20 years and is an advocate of open source technology. He joined Puppet Labs in March 2010 as the VP of Operations, was VP of Engineering at Venmo and is currently VP of Services at Docker Inc. We highly recommend that you read his blog and follow him on Twitter.