An Automation Development Stack

One Implementation of an automation and configuration stack that allows users to leverage a complex stack via a high level of abstraction in order to reduce complexity.

Christopher Steel, December 14th, 2018

Obtaining a High Level of Abstraction

In this case obtaining a High Level of Abstraction:

- Makes is "easy" to leverages the magical powers of some complex technology.
- Allows for the creation of provider agnostic automation scripts which in turn
- Leads to standardized automation roles.
- Likely to increase development velocity, perhaps significantly.

Not exactly magic but...

"technology sufficiently advanced is indistinguishable from magic"

Arthur C. Clarke'a Third Law

- So, no rabbits, just a software stack,
- a few Virtual machines and containers.
- and perhaps a COW or two.

Development Stack Overview

- Ansible - automation software
- LXC / LXD - container system and hypervisor
- Molecule - Ansible role development aid
- OpenZFS - Powerful file system and logical volume manager
- VirtualBox - Opensource Virtual Machine application

Ansible

An opensource automation engine

- Easy to grok
- Provisioning and Configuration management
- Application deployment and Intra-service orchestration.
- Modules in abundance made using any language.
- SSH / No agents required

LXC and LXD

Linux System Containers

- As fast as bare metal (2% slower).
- Ultralight hypervisor
- Well considered CLI / REST API.

Molecule

- Allows for consistently developed Ansible roles
- supports multiple:
  - instances
  - operating systems
  - distributions
  - virtualization providers
  - test frameworks
  - testing scenarios.
  - target nodes (bare-metal, VM's, cloud(s) and/or containers)

OpenZFS

OpenZFS acts as both the volume manager and the file system and has some magical properties.

- Copy-on-write (COW) transactional object model.
- Continuous integrity checking and automatic repair.
- lots of other interesting stuff, RAID-Z, Native NFSv4 ACLs...

VirtualBox

- Opensource
- Well understood
- Mature
- Runs on many OS, able to host many OS...

Virtualization Overview

Before we get started with the details of this particular development stack lets take some time and talk a bit about virtualization and how to go about choosing the right type of virtualization for your particular project or experiment because one size does not fit all when it comes to virtualization.

Reasons you want (need) to virtualize?

- Isolation / security
- Density / effective use of resources
- Efficiency and/or Speed
- Repeatability and/or portability
- Required in order to...
- Other reasons

What is a Hypervisor?

Generally speaking a Hypervisor:

- Allows a physical host to operate multiple VM's or containers as guests.
- Is a process that separate a systems OS and apps from the physical hardware.
- Is software, but could be embedded software on a chip.

Major types of Virtualization (Hypervisors)

- Type 1 - Native (Bare Metal)
- Type 2 - Hosted ("on top of" OS)
- Type C - OS level virtualization / Containerization / Other

Type 1 - Native ("Bare Metal")

```
Hardware <--> Hypervisor <--> Hosted OS
```

- Runs directly on the hosts hardware
- Controls the hardware
- Manages the guest operating system(s)

Type 1 (Bare Metal) Hypervisor Examples:

- VMware ESXi
- Xen (open source - loads it's own paravirtualized host operating system)
- Xbox One system software

Some special properties of Type 1 Virtualization (Hypervisors)

- Speed and Flexibility?
  - Windows hosts (Xen / VMware)
  - FreeBSD (Xen)
  - NetBSD (Xen)
- Security implications
  - No host OS(!)
  - Probably a full OS on your VM's

Type 2 - Hosted Hypervisors

```
Hardware <--> Host OS <--> Hypervisor <--> Hosted OS
```

- Generally an application that runs on a conventional (host) operating system OS.
- The guest operating system runs as a process on the host OS.
- Abstracts the guest OS from the host OS

Examples of Type 2 (Hosted) Hypervisors

- Docker (Probably belongs here now, used to be Type C.)
- KVM
- VirtualBox
- VMware Workstation Pro

Some properties of Type 2 Virtualization

- Isolation of instances
- Support for non-linux OS
- Well understood, familiar to more users.

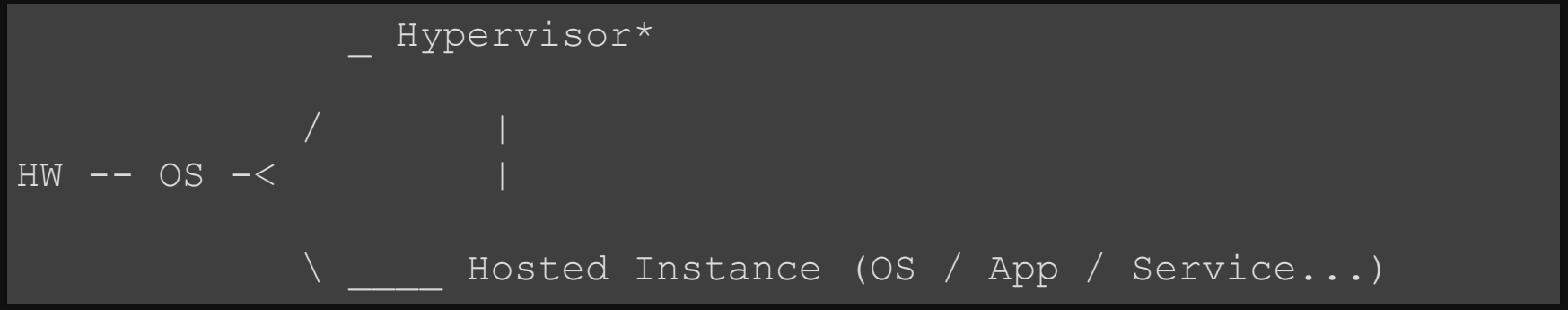
Type C Virtualzation

Catch all including: linux system containers, jails and chroot jails, AIX Workload Partitions (WPARs) and virtual environments.

- Diverse, but shared history
- May make use of a hypervisor
- May allow for the existence of multiple isolated user-space instances.
- May be recursively virtualizable

Containers, Type C architectures that employ a hypervisors

Instances (launched containers) look like real computers from the point of view of the programs running in them. Type C virtualization that makes use of a Hypervisor tends to look like this:



A hypervisor is computer software, firmware or hardware that creates and runs virtual machines. Not all Type C virtualization takes place using a hypervisor according to this definition.

Examples of Type C Virtualization using Hypervisors

- Docker - Used to be an alternative Hypervisor for using Linux System Containers (LXC).
- LXC (Linux System Containers) - LXD is an extention of the LXC hypervisor.

Containers

Containers have some really nice features...

- Portable (mobility of compute)
- Usually Faster than Virtual machines
- Lighter than Virtual Machines

Selecting the right container for the job

- Usage (microservice, enterprise application, development, devops, HPC?)
- Users (are you a developer, admin, power user or a researcher?)
- Support & Community, Popularity.
- Strengths & Weaknesses
- Sharing containers - Image/Container Registry

Singularity

Designed for HPC

- Can be "run" or executed directly by file name.
- Image based containers.
- Must be root outside of contianer to become root inside of container.
- no root daemon owned processes.
- can import docker containers.
- Learning curve

Singularity Users

- MCIN users
- Calcul Quebec / Calcul Canada
- HPC all over the world.

Singularity pipline example

After (a lot of) careful preparation...:

```
singularity run --app cat catdog.simg
```

Output:

```
 Meow , this is Cat
```

Application Containers - Docker

- Designed for isolating applications.
- Docker to HPC via Singularity.
- Use of layers and disabling of persistence results in lower disk IO.
- Issues using apps that expect cron, ssh, daemons, logging and other system stuff.
- Can use copy-on-write (CoW) for images and containers.

Docker Users

- MCIN
- Development and test organizations.
- Many, many other organizations.

LXC & LXD (Linux System Containers)

- Acts like a normal OS environment: hostname, IP address, file systems, init.d, SSH access.
- Nearly as fast as bare metal, parallelism possible.
- Efficiently run one or more multi-process applications.
- Linux-native, can leverage CoW using ZFS backing.
- Windows (10 with Linux subsystem enabled) and OSX clients*

Who uses LXC / LXD

- IT Operators / DevOps
- MCIN
- Most Canonical Websites

LXC/LXD usage

- Will will cover this in the next section

Choosing your virtualization target(s)

- Do you have / require root access on your virtualization project?
- What will you be virtualizing? An application, many applications, part of a pipeline?
- Host - Where will your container be running?
- Which OS's will be running on your host and guest systems?

Molecule

Enables the development of provider agnostic Ansible playbooks and roles.

- Includes built in provider specific Ansible tasks created using Ansible provider modules.
- Allows for the creation of custom providers when Ansible provider modules are not available.

molecule init

- **molecule init role** is used to create a new provider agnostic Ansible role and a default scenario.
- **molecule init scenario** is used to create additional scenarios for an existing Ansible role.

```
molecule init role -r myrole
cd myrole
rm -R molecule/default
molecule init scenario -s default -d vagrant -r myrole
molecule init scenario -s lxd -d lxd -r myrole
```

Generated files

Generating a **scenario** create a scenario directory and contents. The INSTALL.rst file includes information on any additional requirements molecule has for supporting a particular scenario:

```
ls -al molecule/default/
-rw-r--r-- 1 cjs cjs  260 Dec  5 20:08 INSTALL.rst
-rw-r--r-- 1 cjs cjs  303 Dec  5 20:23 molecule.yml
-rw-r--r-- 1 cjs cjs   64 Dec  5 20:08 playbook.yml
-rw-r--r-- 1 cjs cjs  235 Dec  5 20:08 prepare.yml
drwxr-xr-x 2 cjs cjs 4096 Dec  5 20:08 tests
```

molecule.yml

- Each scenario has a **molecule.yml** file.
- By default a single instance is created called instance.
- You may want to call it something more meaningful.

```
nano molecule/default/molecule.yml
nano molecule/lxd/molecule.yml
```

molecule create

I use vagrant/virtualbox and lxd/lxd based scenarios for most of my testing. I configured LXC/LXD to be backed with ZFS (as a file). So once I have a local copy of the containers base image when ever **molecule create** is run against an **lxd** scenario it uses Copy On Write (COW) to create the instance as I configured LXC/LXD to be backed using ZFS as a file.

```
# molecule check  -s lxd      # do a dry run
time molecule create          # create default instance
time molecule create -s lxd   # create lxd instance (using COW)
```

Other molecule commands

Molecule has a total of 16 high level commands at this time. Here are three:

```
time molecule converge -s lxd # configure the instance(s) using r
time molecule test -s lxd     # Run all tests...
```

molecule list

```
molecule list
```

molecule destroy

```
molecule destroy
```

LXC/LXD

Our storage pool

```
lxc storage info lxd
```

Launching a Test Container

Checking our space usage:

```
lxc storage info lxd
```

List our containers

```
lxc list
```

Some Key Points

In Closing

- We have learned a lot and we are still learning about our new stack.
- It makes a lot of things that used to be impractical practical.
- High levels of abstraction rock.
- Use the right kind of Virtualization for the job at hand.
- For extensive testing you will want to leverage the fastest appropriate provider driver
- Use one or more scenarios per driver depending on your needs.

Raising the level of abstraction makes automation much easier as it hides many levels of complexity.

In this stack this includes:

- LXC and LXD enabling the configuration and leveraging of a ZFS backend.
- molecule allowing use to ignore provider specifics once the provider and instance is configured.

This in turn means:

- Allows us to provisioning to multiple targets.
- Allows us to develop target agnostic Ansible roles.
- Enables a significant increase in development velocity.

References

Comparisons

Web pages

- https://robin.io/blog/linux-containers-comparison-lxc-docker/
- https://robin.io/blog/containers-deep-dive-lxc-vs-docker-comparison/
- https://www.xenproject.org/users/virtualization.ht

Papers

- Formal Requirements for Virtualizable Third Generation Architectures-10.1.1.141.4815
- Analysis of Virtualization Technologies for High Performance Computing Environments
- Performance Evaluation of Container-based Virtualization for High Performance Computing Environments
- NIST.SP.800-125A-F - Security Recommendations for Hypervisor Deployment
- VIRTUALIZATION TECHNIQUES & TECHNOLOGIES: STATE-OF-THE-ART

Presentations

MCIN