An assesment of the Docker micro container

Darrell Pratt, Architect Leader, GBS

Executive Summary

Virtual machines have always been a large part of any sizeable development organization and their use as a means to virtualize the resources of a physical machine are well known. A new application called Docker is turning this concept of virtualization into immediately accessible micro containers for any organization to easily tie into their workflows. This framework has grown verey rapidly with even Amazon Web Services just recently announcing support for Docker in their Elastic Cloud service.

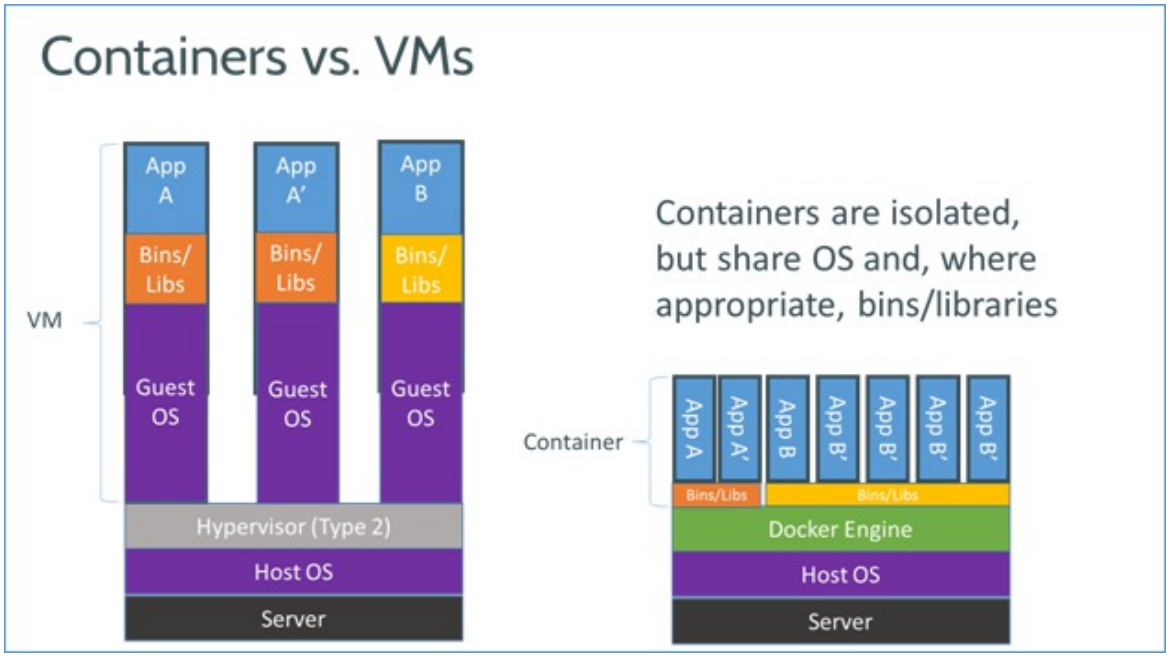
Problem Statement

Virtualization has helped companies make the best use of their physical hardware within data centers. Primarily, this has been the realm of VMWare and few other built in solutions within various operating systems. Although some groups have built quite a bit of tooling around VMWare and the provisioning of images to users, there is still quite a bit of setup involved with this approach and no true method for the developer to declare which resources are needed for the image and how to separate those images based upon the application's needs. At the completion of provisioning, one still has a VM that looks like a physical machine that might be running various application servers or databases. There is no true separation of concerns with this approach.

Docker Background

Docker was created to solve the issue of build once, run anywhere. Docker automates the creation of a self-sufficient container that can run anywhere, but displays the same outward behaviour to the developer or ops team regardless of location. A central difference between Docker and traditional virtual machines, is that the Docker image is extremely lightweight as it has no host OS (it uses the host OS) and it can be deployed and started with very minimal delay. Docker itself uses LXC (Linux Containers) which acts as a direct interface to the containments features which are part of the linux kernel. This acts at a lower level than Xen which (a hypervisor) and much lower than the VM level such as VMWare uses.

An illustration of the difference between a traditional container and Docker is illustrated below.



Docker vs. VMs

Docker manages to scale in the sense of maintaining many images of different applications or version of those applications because of this design pattern. The container is essentialy the application and its dependent binaries and libraries. Docker is intelligent enough to only store the differential between any of these layers across the same base images.

What does this mean? For example, imagine a large Java Application which is maintained in git with a build process hosted within Jenkins. The build process could easily take a given base Dockerfile and load a different codebase into that container, instantiate it, run a full test suite and then decommission that container. The build could do this is a completely automated fashion. These containers can also run in parallel on a host or host sharing the unchanged components between each container. This allows for scaling the number of containers with very little disk space or resources being used. With a traditional VM infrastructure, this would require unique guest host images, which are several gigabytes in size each and take time to build and startup. Docker makes this trivial and extremely fast.

The Docker name itself comes from the idea of shipping. As the company explains it, shipping used to be done in a break bulk fashion. Simply put, cargo was shipped as it came, without any standardized boxing. Shipping companies needed to worry about interactions between any items that might be onboard a ship at any time. This was revolutionized after 1960 as shipping containers were introduced. These containers are common sight now, but this one change allowed for effeciencies in loading and unloading of cargo, and relay of those items through the other transit systems. This analogy holds true to what Docker is attempting with system virtualization as well. Docker generally attempts to alleviate the concerns over what might be contained within. This containment solves many problems with library and dependency hell that are typically seen with standard virtualization and certainly bare metal deployments. This allows any application and its various dependencies to be packaged up as a lightweight container that can be deployed virtually

anywhere.

That should suffice for a background of Docker and its goals. The power of Docker is truly its ease of use and configurability. The following sections will explore how to setup a Docker image and run a container as well as some use cases for what Docker could be used for at Nielsen.

Configuration

Docker makes use of a simple configuration file to build images. The file is plain text and has a fairly small set of commands. The configuration is a set of instructions. These instructions are all in the form below.

```
# comments being with hash
INSTRUCTION arguments
```

The first instruction in any dockerfile must be the FROM instruction. This tells docker which image is considered to be the base image of the file. Think of this as a state at which you wish to further customize the container.

```
# Start with a base Ubuntu image
FROM ubuntu:13.03
```

This would use the 13.03 release of Ubuntu as our base image. There are several distributions one can choose from, such as centos, smartos and many others.

After specifiying the base image with the **FROM** instruction the container more than likely needs to be customized with your preferred libraries and binaries needed to run your application. The **RUN** instruction is used here to run any arbitrary command on the new image.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
```

The above dockerfile will load Ubuntu and then run apt-get to install a few libraries onto the new image. Note the ***-y*** on **apt-get**. This is important so as to make the apt-get command run non-interactively.

The RUN instruction can also take the following form.

```
RUN ['command', 'param1', 'param2' ...]
```

This is useful when running a command on the image that takes a complex set of arguments. In our case above it is not necessary and the first form appears a bit cleaner to the author. To add to our example above, we will add additional RUN instructions to build out a nodejs environment for our new image.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
#Add node repository to sources.list and update apt
RUN     add-apt-repository ppa:chris-lea/node.js && apt-get update

#Install node.js
RUN     apt-get -y install nodejs
```

With 10 lines of text, we now have an ubuntu machine that will have nodejs installed and ready for use. We still need to get our application onto the image however. The **ADD** instruction allows us to mount resources onto the image. This command takes the following form.

```
ADD <src> <dest>
```

Any number of these ADD instructions can be in the dockerfile. Note that the files are added to the new image as 755 0:0. This is important to understand if your application has any complex file system permission requirements. With the ADD instruction we can now add in our application from the local machine into the new image. Assume that our dockerfile is at the project root of our application then the following will add our application's files to the new image.

```
# Add project from current directory to /src dir on image
ADD . /src
```

As we are building a nodejs image we can now use the RUN command to add our node modules needed for the application using npm. (If you have not used node or npm, go do it now). The following instructions will install our dependencies on the image.

```
# Install app dependencies
RUN cd /src/server; npm install
# Install nodemon to run our application forever
RUN npm install nodemon
```

Now that we have our image set to run our application, the next important topic is that of port mapping and linking. Linking is a bit complex so it will be covered later in the document, but port mapping is straight forward. Our application we want to run in our container will listen on some port (if it is a network application). In order to map that internal port of the container to the host OS, we use the **EXPOSE** command as follows.

```
# Expose our container port 300 to the host
EXPOSE 3000
```

Note that we have only specified that this port is available to the host OS. In order to map from our host port to this exposed port that will be specified when we start the container. That will be covered in the command line section later in this document.

The final instruction to use in this example is **CMD**. This instruction takes the following form.

```
CMD ["executable", "arg1", "arg2", ...]
```

This instruction can only exist once within the dockerfile as this is informing the container to run this command and then wait for connections to the container. We've basically created an image that will load the OS, libraries needed, applications needed, loaded our own application and then started our application and exposed it to the host OS in a small amount of configuration code. The full example is below.

```
# Start with a base Ubuntu image
FROM    ubuntu:13.03
# Load some libraries
RUN     apt-get install -y python-software-properties python g++
RUN     apt-get install -y make software-properties-common
#Add node repository to sources.list and update apt
RUN     add-apt-repository ppa:chris-lea/node.js && apt-get update

# Install node.js
RUN     apt-get -y install nodejs

# Map our local files to the image
ADD . /src

# Install app dependencies
RUN cd /src/server; npm install
# Install nodemon to run our application forever
RUN npm install nodemon
# Expose our nodejs port to the host
EXPOSE 3000
# Run our server using nodemon
CMD ["nodemon", "/src/server/server.js"]
```

There are few other instructions that were not used in this example. The **ENV** instruction allows the container to be set with any set of environment variables. ENV takes the following form.

```
ENV <key> <value>
```

Note that while this could be helpful for a set of well known environments settings, one can pass environment key value pairs to the container on running as well.

The **ENTRYPOINT** instruction is similar to the **CMD** instruction, but it is used to basically tell docker to run the image as an executable. There can be more than one **ENTRYPOINT** command in a dockerfile but the last one is the only one which will run. This instruction takes the following form.

```
ENTRYPOINT      <cmd> <args>
```

A possible example would be to use an **ENTRYPOINT** instruction to create a container that would run a data import process. This would allow any number of these containers to be activated when needed and to be torn down and cleansed from the system when not in use.

Now that we have written a dockerfile to configure an image, we need to actually build that image to use it. The next section will detail the docker commands and give some guidance on how to setup

an environment to work with docker effectively.

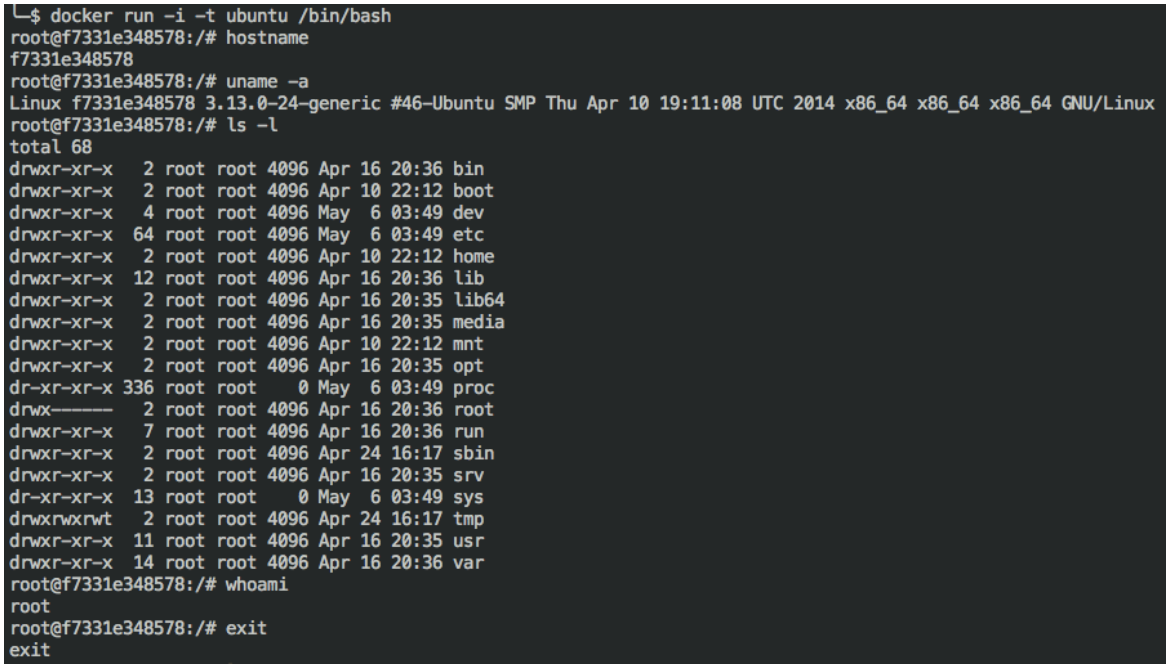
Image Management and Creation

The docker client and daemon process are easy to install on a number of operating systems. The following set of commands will install docker on an unbuntu or ubuntu like system.

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

If you have made it this far, note that most examples will start to look repetitive with the use of *sudo*. Docker does need to be run as root or in a privelaged group, but if a docker group exists on the host OS (which it should if it is installed properly) then if the user (you in this case) is a member of the docker group then sudo is not required. This is highly recommeded for learning docker, but in production one would probably want to run with sudo to preserve security.

Once docker is installed you can test the installation by running the simplest container possible. The following command will run a base ubuntu container and give you an interactive shell to run commands from. Note the load time of the container upon the first run and then exit the container and run it again and note the difference. This is the effect of image caching and this will begin to demonstrate the speed of docker.



docker run command

Image Linking

Scripting Docker with Ansible

Use Case

Benefits

Conclusion