AMIS, Data Driven Blog – Oracle & Microsoft Azure

Getting value from IoT, Integration and Data Analytics

Provisioning Virtualization and Oracle VM

First steps with provisioning of Docker containers using Vagrant as provider

Lucas Jellema *August 22, 2015* *1*

Note: the next set of (more advanced) steps with Vagrant and Docker are discussed in this article: Vagrant and Docker – Next and Advanced steps with folders, ports, volumes, linking and more.

Automated environment provisioning and use of virtualization isolate environments is a topic that has become quite important to me – and to many others. From virtual machine technology, such as VMWare, VirtualBox and Oracle VM, to more recently the use of container technology and especially Docker, I am constantly trying to wrap my head around things and find ways of efficiently, smoothly hence automatedly working with environments. Vagrant has been a great tool for me to produce VirtualBox machine images that I use for a host of things including development environments. I have frequently used Puppet in conjunction with Vagrant to do the detailed configuration of the Virtual Machine – including the installation of various software packages.

containers can provide (almost) the same isolation as a series of Virtual Machines but with much smaller overhead – in terms of usages of disk, CPU and memory and in terms of time required for starting up and shutting down. Additionally, the management of Docker containers – from 10Ks of images to start from and a layered mechanism to extend and revert – has a number of attractive options. A Docker container – once configured and runnable – can be distributed fairly easily – using a public or private registry – and can also be deployed to a growing number of enterprise stacks (such as OpenStack cluster) and public cloud providers.

Docker requires a Linux host as Docker itself leverages the LXC (Linux Containers) mechanism in Linux. This means that in order to work with Docker on non-Linux systems – Windows, Mac OS X, Solaris – we first need to set up a Virtual Machine running Linux. The special boot2docker package is a lightweight Linux distribution made specifically to run Docker containers. A quick way on Windows and OS X to get started with Docker is by using boot2docker as an intermediate mechanism.

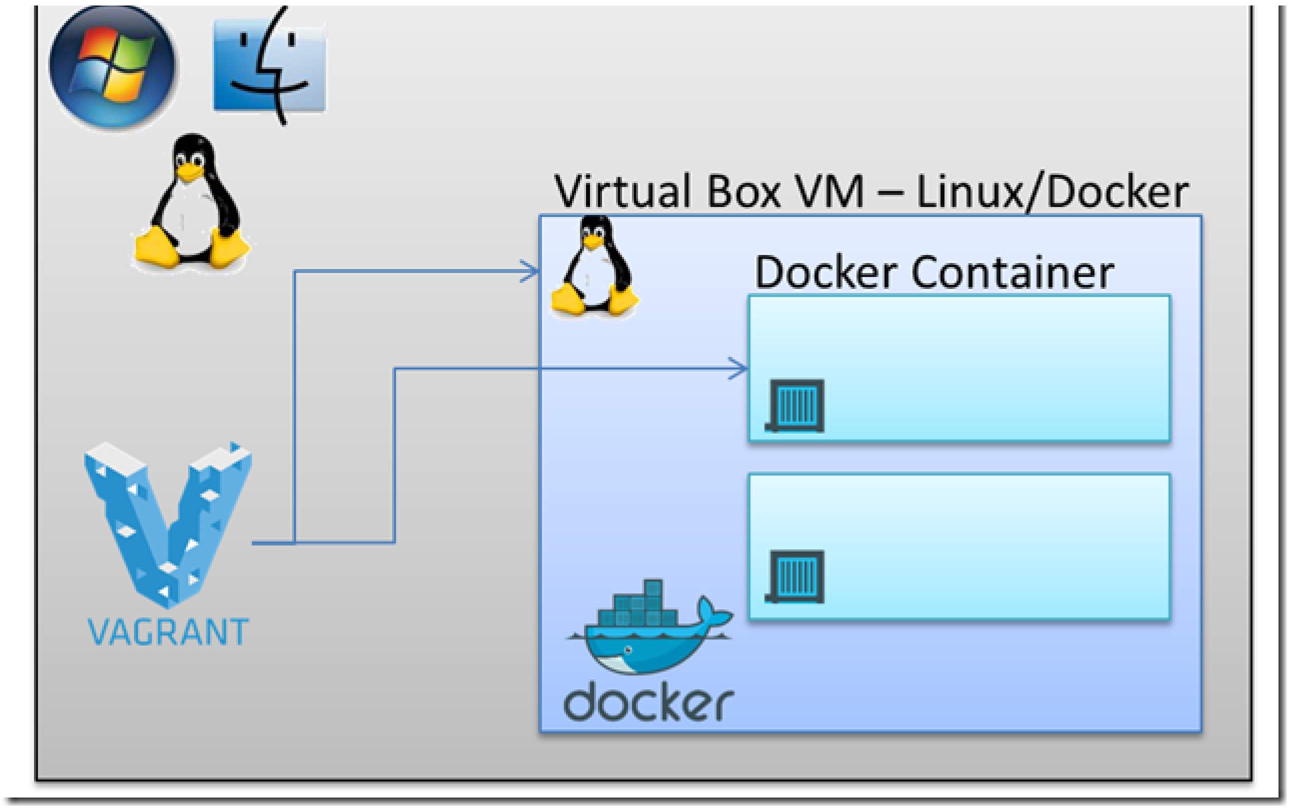
The association between Vagrant and Docker is threefold:

1. automate the building of Docker containers
2. coordinate running of Docker containers
3. provide a Docker enabled host VM on which Docker is run

1. Vagrant can be used to run Docker containers – with vagrant up resulting in the configured container(s) being run. Vagrant supports special commands to produce logging from Docker containers and to run one-off commands in a container. The Vagrant way of defining folder mapping is leveraged through Docker volume instructions, resulting in host folders being accessible inside the Docker container.

2. Vagrant has embraced Docker as a provider, just as it supports providers for VirtualBox and VMWare. This means that a Vagrant configuration file can describe a Docker container just as it can describe the configuration of a VirtualBox VM. When Vagrant runs with a configuration for Docker, it will start provisioning the Docker container as specified. This means that anyone comfortable using Vagrant as mechanism for describing and provisioning environments can quickly apply their existing knowledge and skills to Docker containers instead for classic VM images. The port forwarding and host-to-guest folder mapping that we configure in the Vagrant file is now applied to host-to-Docker container interaction.

3. One very nice additional touch is that Vagrant is aware of the fact that Docker containers cannot run natively at present on Windows or Mac OS X. When Vagrant is asked to provision a Docker container on one of these operating systems, it can either automatically engage

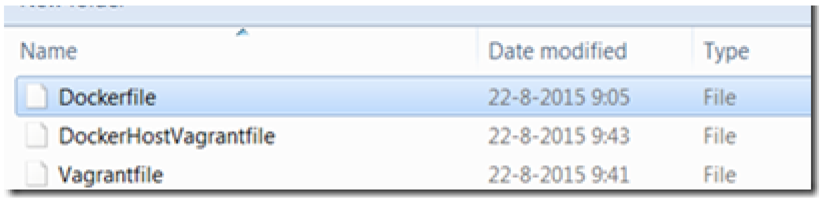


boot2docker as a vehicle to create and run the Docker container in or provision a Linux based VM image that it then enables for Docker and creates the Docker container into. Using a traditional Vagrant file, we can describe the Docker host VM that we want Vagrant to create or leverage (if it already exists). This host VM can of course be made to look much more like a production Linux host than the boot2docker environment. The same Docker host VM that Vagrant creates can be reused for many Docker containers provided by Vagrant: a single Linux VM suffices for running multiple containers that can mutually be linked and share file system folders. Vagrant can also provision such a Docker VM when it is running on Linux on which it could also natively install Docker can build containers.

This article is an overview of my first steps with the combination of Vagrant and Docker (on Windows) – and really not much more than a rehash of the best parts provided in several great articles listed in the Resources section. It is a stepping stone towards automated environment provisioning using Vagrant, Docker and Puppet and provides some examples that address the main challenges I faced in getting started.

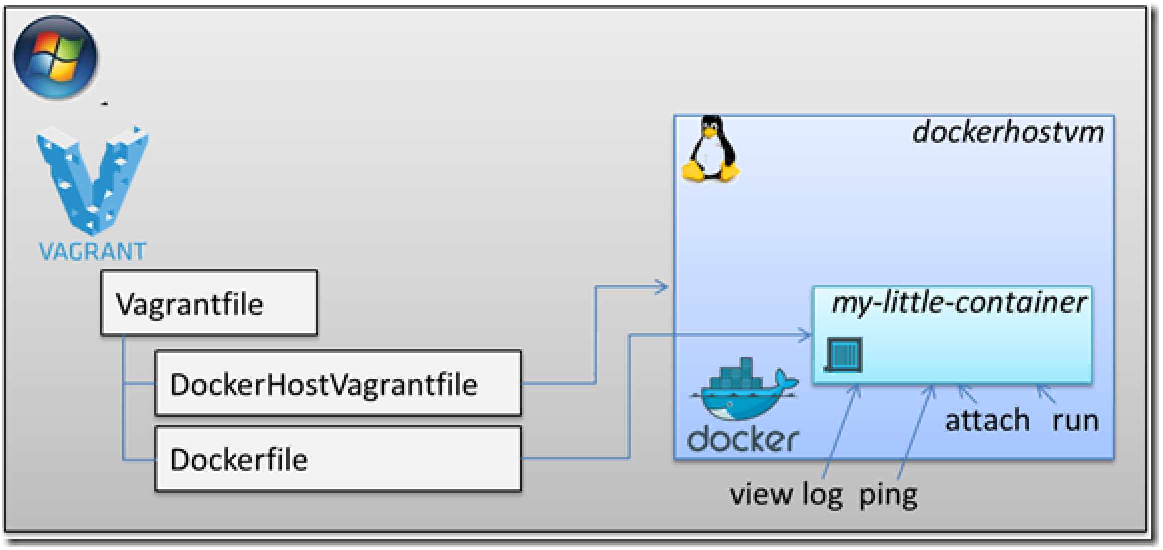
Simple first step – build and run a Docker container

Using three files on my Windows laptop that has both VirtualBox and Vagrant installed



Name	Date modified	Type
Dockerfile	22-8-2015 9:05	File
DockerHostVagrantfile	22-8-2015 9:43	File
Vagrantfile	22-8-2015 9:41	File

I will create and run a very simple Docker container, based on the Ubuntu 14.04 image. The container will do very little of interest: it pings localhost 51 times and exits again. This gives us some time to inspect the container, ping it, attach to it and look at its output. Later on in this article, things will get a little more interesting.



The contents of the Vagrantfile:

```
ENV['VAGRANT_DEFAULT_PROVIDER'] = 'docker'

Vagrant.configure("2") do |config|

config.vm.define "my-little-container" do |m|

m.vm.provider :docker do |d|
d.name = 'my-little-container'
d.build_dir = "."
d.cmd = ["ping", "-c 51", "127.0.0.1"]
d.remains_running = true
d.vagrant_machine = "dockerhostvm"
d.vagrant_vagrantfile = "./DockerHostVagrantfile"
end
end
end
```

The DockerHostVagrantFile that it refers to:

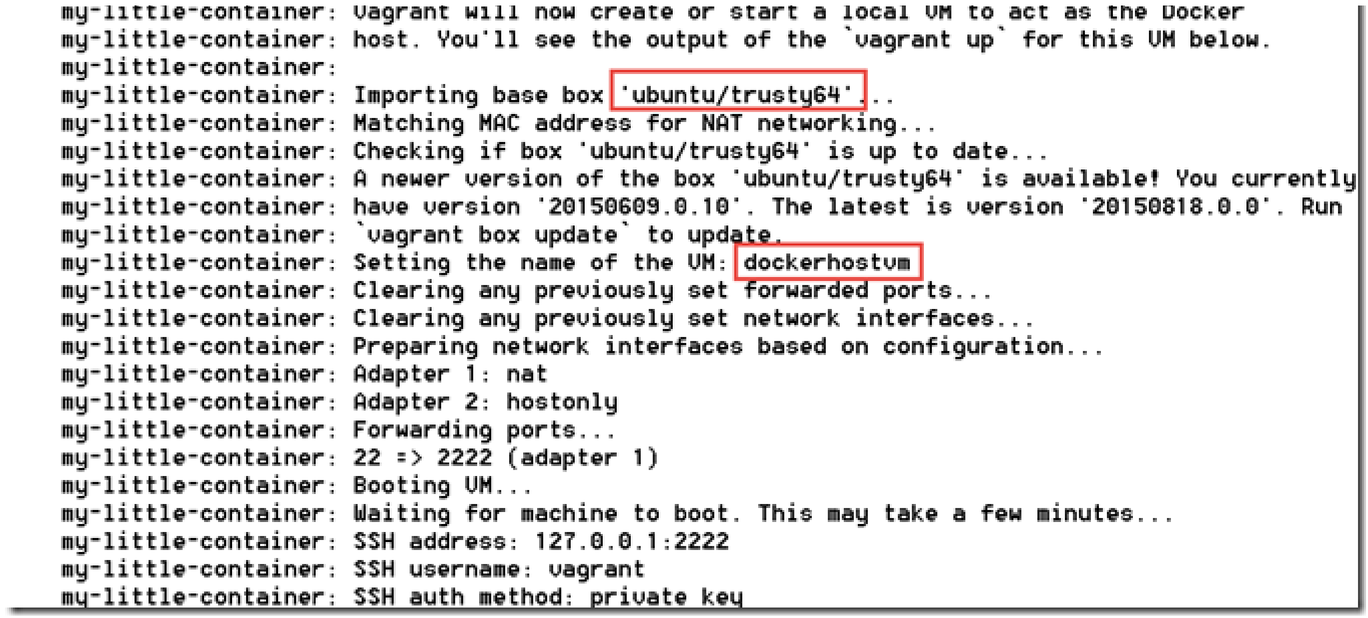
```
Vagrant.configure("2") do |config|
# The following line terminates all ssh connections. Therefore Vagrant
# That's a workaround to have the docker command in the PATH
# Command: "docker" "ps" "-a" "-q" "--no-trunc"
# without it, I run into this error:
# Stderr: Get http:///var/run/docker.sock/v1.19/containers/json?all=1:
# Are you trying to connect to a TLS-enabled daemon without TLS?

config.vm.provision "shell", inline:
"ps aux | grep 'sshd:' | awk '{print $2}' | xargs kill"

config.vm.define "dockerhostvm"
config.vm.box = "ubuntu/trusty64"

config.vm.provider :virtualbox do |vb|
vb.name = "dockerhostvm"
end

end
```

Finally the Docker file:

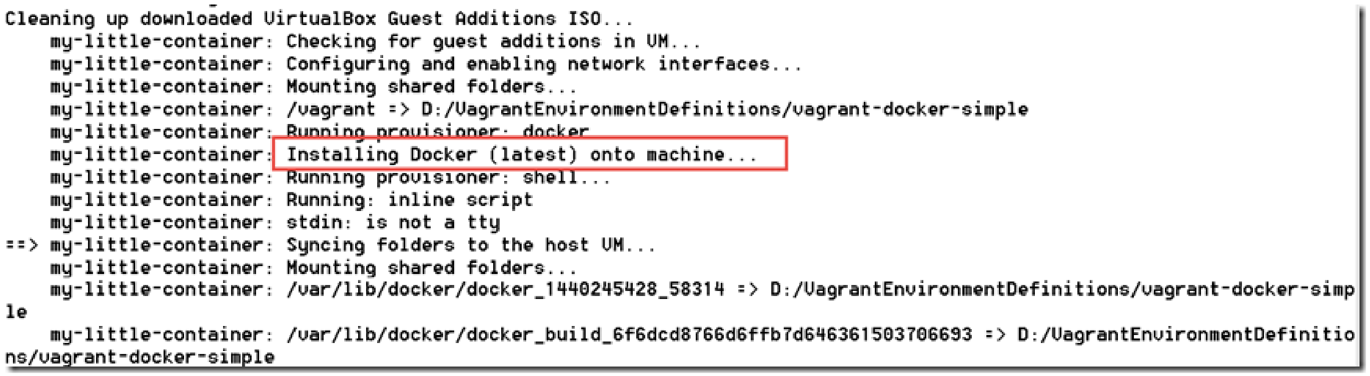
```
FROM ubuntu:14.04

RUN mkdir /u01 &amp;&amp; \
chmod a+xr /u01
COPY /files/readme.txt /u01/
```

The command to get things started is *vagrant up*. This will have vagrant start provisioning, based on the Vagrantfile. Vagrant will realize that we ask for Docker and that the operating system is Windows that does not (yet) natively support Docker. Therefore, a Docker enabled host VM is required. Instead of creating such a VM based on boot2docker – which is default behavior – Vagrant will interpret the reference to the DockerHostVagrantFile. This file contains a straightforward definition for a VirtualBox VM based on an Ubuntu base box (ubuntu/trusty64). The file can also specify a private network configuration with a preset IP address that we can later on use to connect from the Windows host into the Docker Host (VM). We could also specify other VM settings – such as CPUs and memory – just as we can do for any VM created by Vagrant.

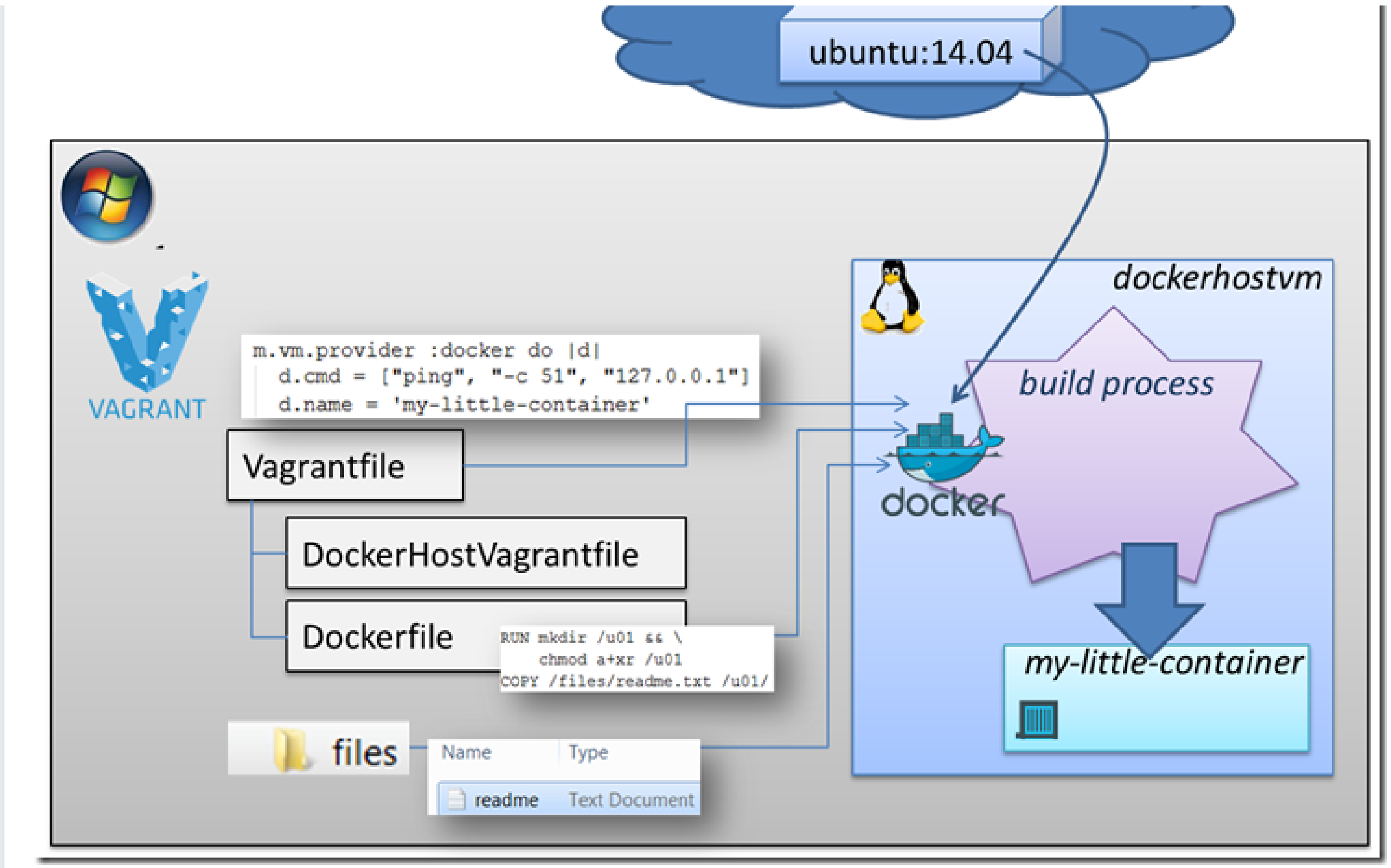
The Vagrant logging shows the creation of the *dockerhostvm* Virtual Box VM:


```
    my-little-container: Vagrant will now create or start a local VM to act as the Docker
    my-little-container: host. You'll see the output of the `vagrant up` for this VM below.
    my-little-container:
    my-little-container: Importing base box 'ubuntu/trusty64'...
    my-little-container: Matching MAC address for NAT networking...
    my-little-container: Checking if box 'ubuntu/trusty64' is up to date...
    my-little-container: A newer version of the box 'ubuntu/trusty64' is available! You currently
    my-little-container: have version '20150609.0.10'. The latest is version '20150818.0.0'. Run
    my-little-container: `vagrant box update` to update.
    my-little-container: Setting the name of the VM: dockerhostvm
    my-little-container: Clearing any previously set forwarded ports...
    my-little-container: Clearing any previously set network interfaces...
    my-little-container: Preparing network interfaces based on configuration...
    my-little-container: Adapter 1: nat
    my-little-container: Adapter 2: hostonly
    my-little-container: Forwarding ports...
    my-little-container: 22 => 2222 (adapter 1)
    my-little-container: Booting VM...
    my-little-container: Waiting for machine to boot. This may take a few minutes...
    my-little-container: SSH address: 127.0.0.1:2222
    my-little-container: SSH username: vagrant
    my-little-container: SSH auth method: private key
```

Once the installation of the Virtual Box Guest Additions on this VM is complete, then the default mapping of the Vagrant directory to the VM is created. Next, the VM is Docker-enabled. A single in line script command is executed on the *dockerhostvm* to ensure we do not run into a permission denied error (see comment in DockerHostVagrantFile). In order to make the Dockerfile directory available during container build, this directory also needs to be mapped into the *dockerhostvm*.


```
Cleaning up downloaded VirtualBox Guest Additions ISO...
    my-little-container: Checking for guest additions in VM...
    my-little-container: Configuring and enabling network interfaces...
    my-little-container: Mounting shared folders...
    my-little-container: /vagrant => D:/VagrantEnvironmentDefinitions/vagrant-docker-simple
    my-little-container: Running provisioner: docker
    my-little-container: Installing Docker (latest) onto machine...
    my-little-container: Running provisioner: shell...
    my-little-container: Running: inline script
    my-little-container: stdin: is not a tty
==> my-little-container: Syncing folders to the host VM...
    my-little-container: Mounting shared folders...
    my-little-container: /var/lib/docker/docker_1440245428_58314 => D:/VagrantEnvironmentDefinitions/vagrant-docker-simp
le
    my-little-container: /var/lib/docker/docker_build_6f6dcd8766d6ffb7d646361503706693 => D:/VagrantEnvironmentDefinitio
ns/vagrant-docker-simple
```

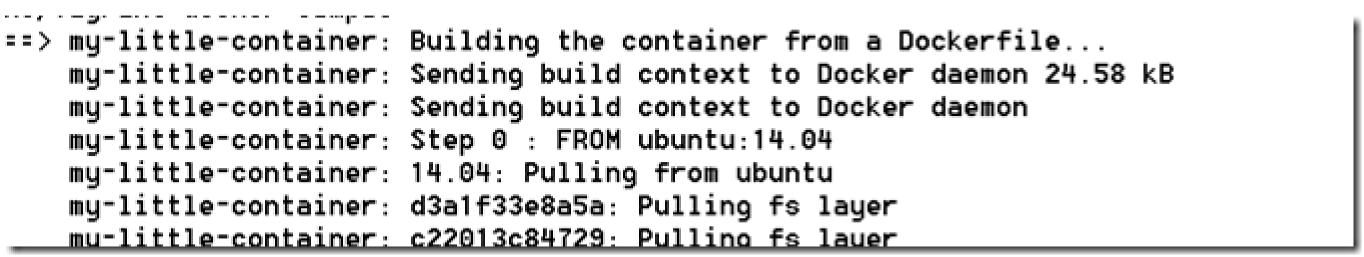
Then, the provisioning of the Docker container can take place. It is visualized in this next image:



```
m.vm.provider :docker do |d|
  d.cmd = ["ping", "-c 51", "127.0.0.1"]
  d.name = 'my-little-container'
```

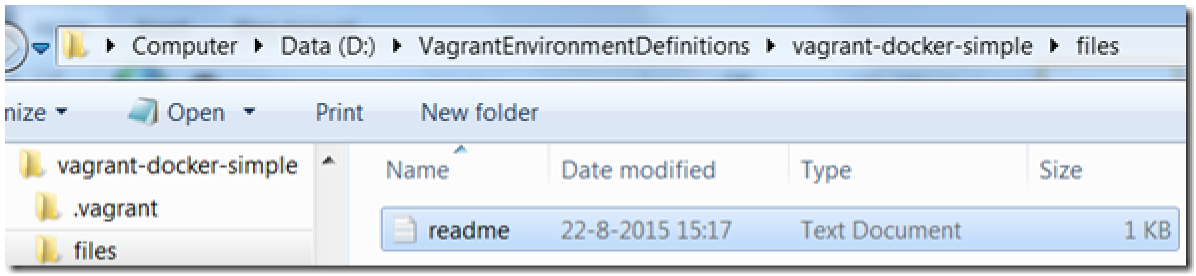
```
RUN mkdir /u01 && \
    chmod a+xr /u01
COPY /files/readme.txt /u01/
```

First, the Ubuntu image is pulled in :



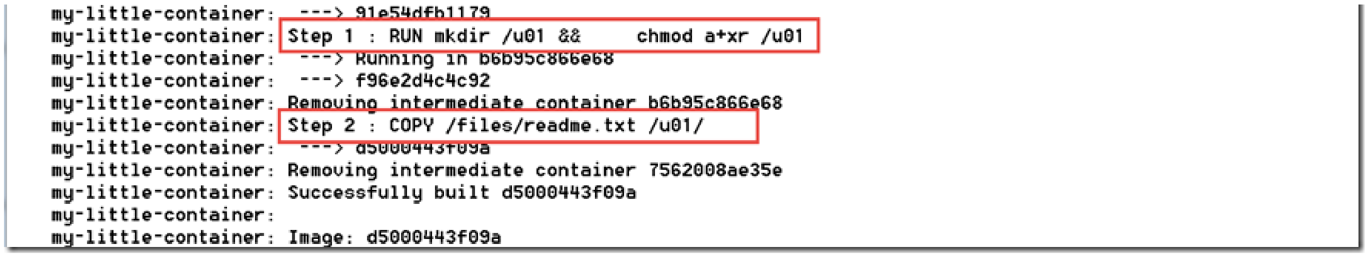
```
==> my-little-container: Building the container from a Dockerfile...
    my-little-container: Sending build context to Docker daemon 24.58 kB
    my-little-container: Sending build context to Docker daemon
    my-little-container: Step 0 : FROM ubuntu:14.04
    my-little-container: 14.04: Pulling from ubuntu
    my-little-container: d3a1f33e8a5a: Pulling fs layer
    my-little-container: c22013c84729: Pulling fs layer
```

Next the statements in the Dockerfile – such as COPY and RUN are executed. Note that the directory that contains the Dockerfile is available in the build context as the root directory. I have added a *files* subdirectory to this directory and created a text file called readme.txt in it.


Computer ▸ Data (D:) ▸ VagrantEnvironmentDefinitions ▸ vagrant-docker-simple ▸ files
nize
Open
Print
New folder
vagrant-docker-simple
.vagrant
files

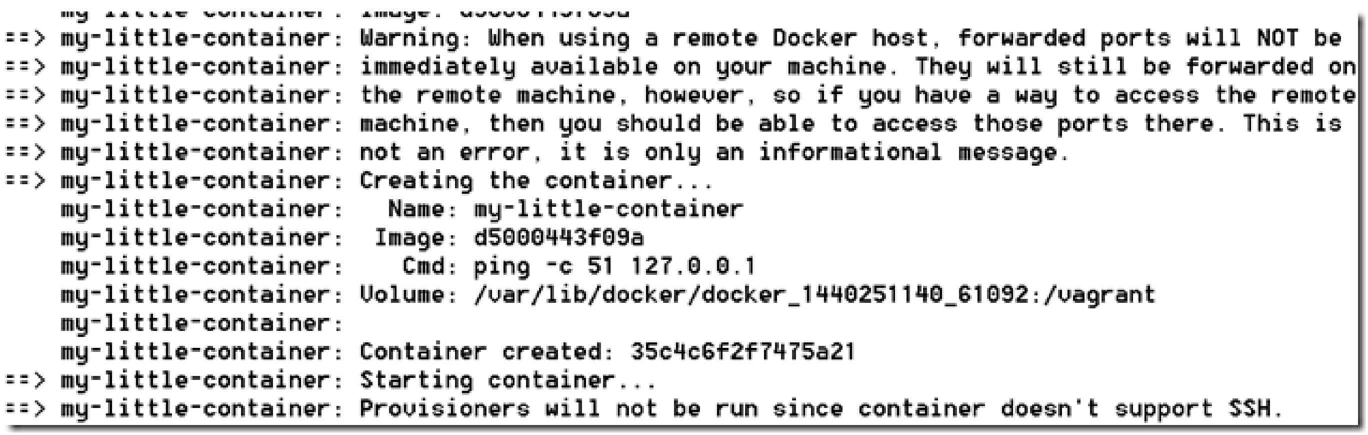
Name	Date modified	Type	Size
readme	22-8-2015 15:17	Text Document	1 KB

The steps in the Dockerfile have Docker create a directory /u01 and copy the file readme.txt from the /files/ directory in the build contxt to this new /u01 target directory.



```
    my-little-container:  ---> 91e54dfb1179
    my-little-container: Step 1 : RUN mkdir /u01 &&     chmod a+xr /u01
    my-little-container:  ---> Running in b6b95c866e68
    my-little-container:  ---> f96e2d4c4c92
    my-little-container: Removing intermediate container b6b95c866e68
    my-little-container: Step 2 : COPY /files/readme.txt /u01/
    my-little-container:  ---> d5000443f09a
    my-little-container: Removing intermediate container 7562008ae35e
    my-little-container: Successfully built d5000443f09a
    my-little-container:
    my-little-container: Image: d5000443f09a
```

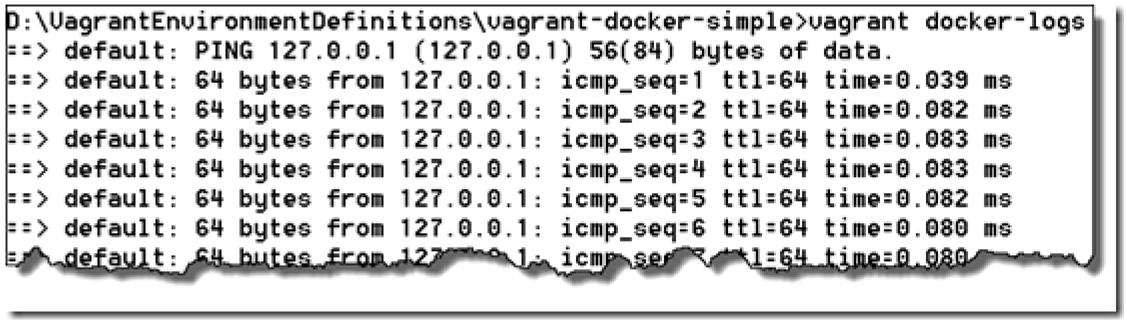
Finally the container is created and started.



```
==> my-little-container: Warning: When using a remote Docker host, forwarded ports will NOT be
==> my-little-container: immediately available on your machine. They will still be forwarded on
==> my-little-container: the remote machine, however, so if you have a way to access the remote
==> my-little-container: machine, then you should be able to access those ports there. This is
==> my-little-container: not an error, it is only an informational message.
==> my-little-container: Creating the container...
    my-little-container:   Name: my-little-container
    my-little-container:  Image: d5000443f09a
    my-little-container:    Cmd: ping -c 51 127.0.0.1
    my-little-container: Volume: /var/lib/docker/docker_1440251140_61092:/vagrant
    my-little-container:
    my-little-container: Container created: 35c4c6f2f7475a21
==> my-little-container: Starting container...
==> my-little-container: Provisioners will not be run since container doesn't support SSH.
```

The CMD parameter in the Docker provider section in the Vagrantfile – ["ping", "-c 51", "127.0.0.1"] – tells Docker to have the container execute the ping statement as soon as it is running – pinging the localhost (127.0.0.1). This statement is repeated 51 times.

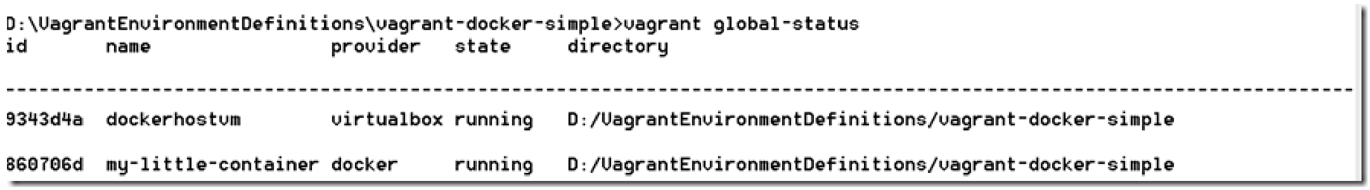
We can see the output from the container on the Windows host using the command *vagrant docker-logs*:



```
D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant docker-logs
==> default: PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
==> default: 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.039 ms
==> default: 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.082 ms
==> default: 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.083 ms
==> default: 64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.083 ms
==> default: 64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.082 ms
==> default: 64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.080 ms
==> default: 64 bytes from 127
```

This tells us that the container is running and producing output (the output from the ping command).

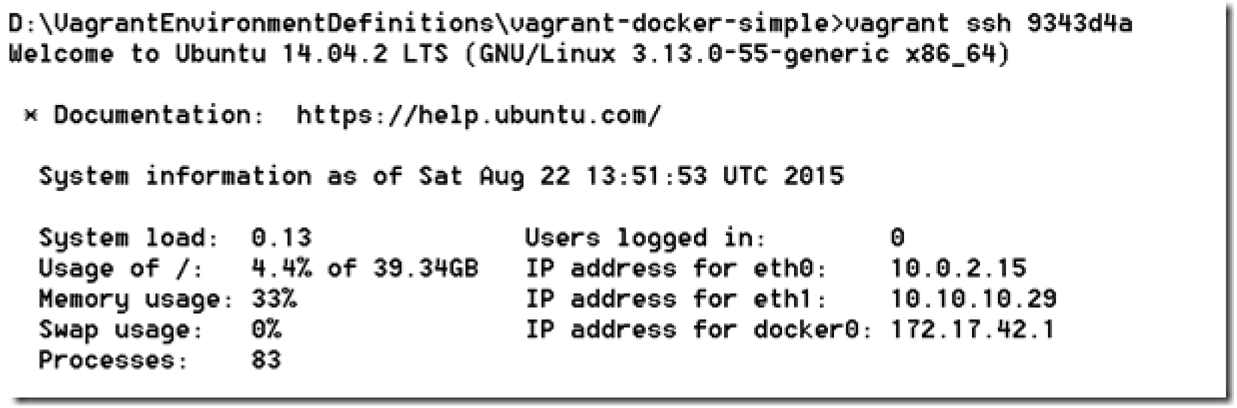
With *vagrant global-status*, we can check on the machines that Vagrant controls. The result from our recent activities is that two entries have been added: the *dockerhostvm* (created by



```
D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant global-status
id       name               provider   state     directory
--------------------------------------------------------------------------------------------------------
9343d4a  dockerhostvm       virtualbox running   D:/VagrantEnvironmentDefinitions/vagrant-docker-simple

860706d  my-little-container docker    running   D:/VagrantEnvironmentDefinitions/vagrant-docker-simple
```

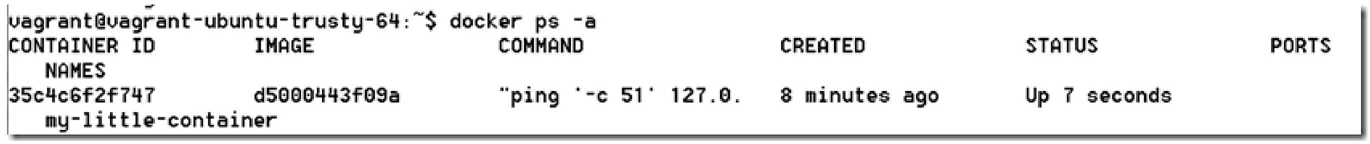
Note that all vagrant commands by default act on the Docker container – listed here as *default*. Vagrant destroy, halt, up etc. all act on that container and not on the *dockerhostvm* Virtual Box VM. If you want vagrant to act on that machine, the commands need to make use of the id of the VM – for example *vagrant halt <machine id>*.

Using *vagrant ssh <machine id for dockerhostvm>* we can connect into the Docker host – where we can issue Docker commands, such as docker ps -a (to check on containers):



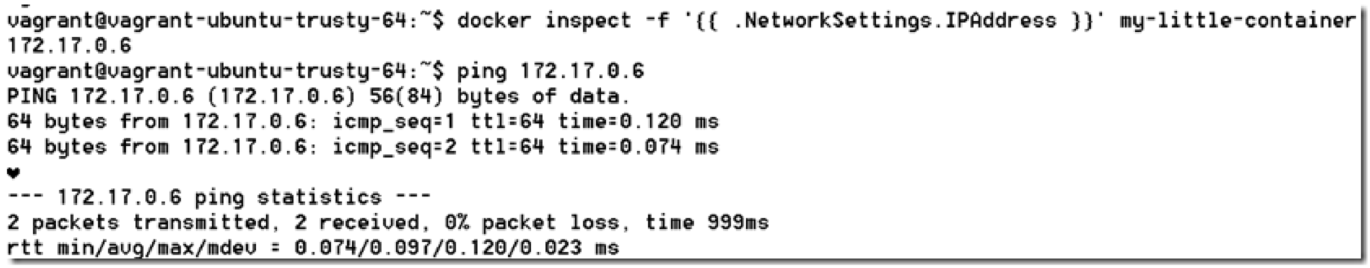
```
D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant ssh 9343d4a
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-55-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Sat Aug 22 13:51:53 UTC 2015

  System load:  0.13              Users logged in:       0
  Usage of /:   4.4% of 39.34GB   IP address for eth0:   10.0.2.15
  Memory usage: 33%               IP address for eth1:   10.10.10.29
  Swap usage:   0%                IP address for docker0: 172.17.42.1
  Processes:    83
```

Check on containers running or exited:



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND              CREATED             STATUS              PORTS
   NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   8 minutes ago       Up 7 seconds
   my-little-container
```

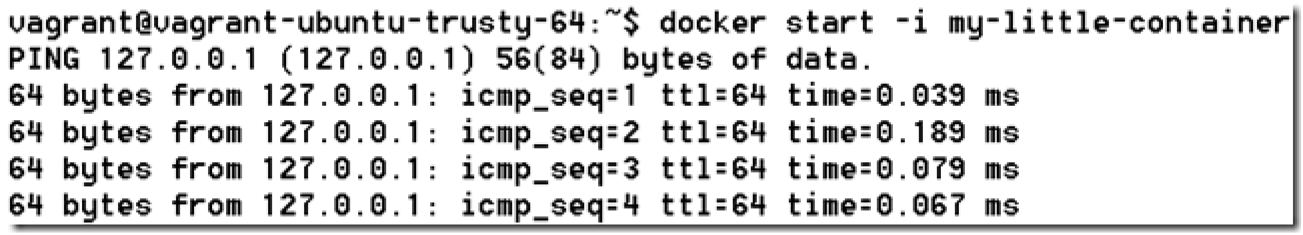
Start the container, check on its IP address using *docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container id>* and ping to it to verify its accessibility:



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' my-little-container
172.17.0.6
vagrant@vagrant-ubuntu-trusty-64:~$ ping 172.17.0.6
PING 172.17.0.6 (172.17.0.6) 56(84) bytes of data.
64 bytes from 172.17.0.6: icmp_seq=1 ttl=64 time=0.120 ms
64 bytes from 172.17.0.6: icmp_seq=2 ttl=64 time=0.074 ms
♥
--- 172.17.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.074/0.097/0.120/0.023 ms
```

Using *docker attach <container id>* we can attach to a running container using the container's ID or name, either to view its ongoing output or to control it interactively.

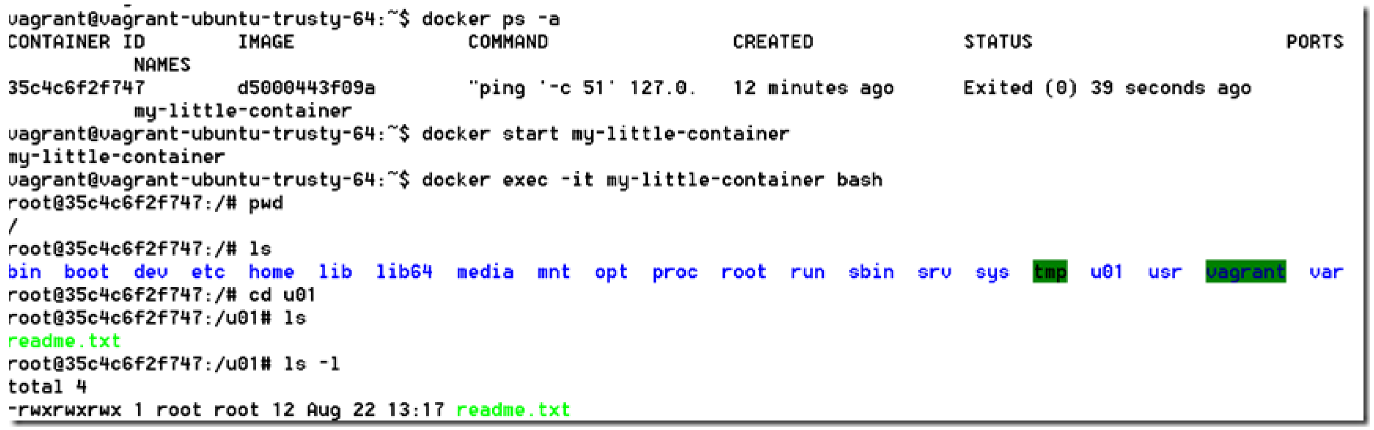
we start viewing the output from the repeated ping command.



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker start -i my-little-container
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.189 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.079 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.067 ms
```

A running container – executing some kind of job – can be simultaneously accessed to perform other processes. Using *docker exec -it <container -id> bash* for example, we start an interactive shell in an already running container. So while the my-little-container is doing its 51 laps of pinging, we can do other things as well. Such as … ping too?

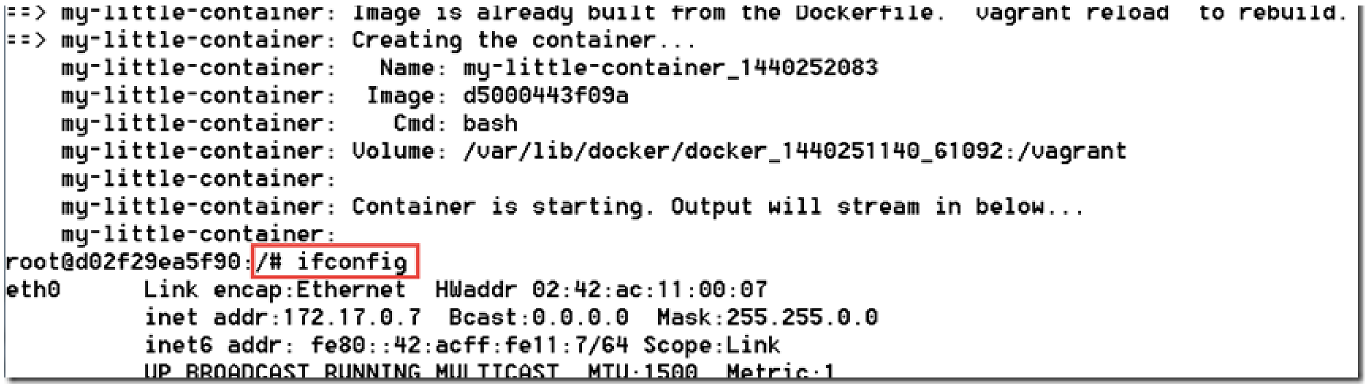
This screenshot shows me listing the containers, starting the container (using its container id) and then when it is running also using *docker exec* to run a command (in this case *bash* to open a shell) against that same container. Also notice that I use the -it flags with this command, to give an interactive session with terminal.



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS                        PORTS
          NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   12 minutes ago      Exited (0) 39 seconds ago
          my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker start my-little-container
my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker exec -it my-little-container bash
root@35c4c6f2f747:/# pwd
/
root@35c4c6f2f747:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  u01  usr  vagrant  var
root@35c4c6f2f747:/# cd u01
root@35c4c6f2f747:/u01# ls
readme.txt
root@35c4c6f2f747:/u01# ls -l
total 4
-rwxrwxrwx 1 root root 12 Aug 22 13:17 readme.txt
```

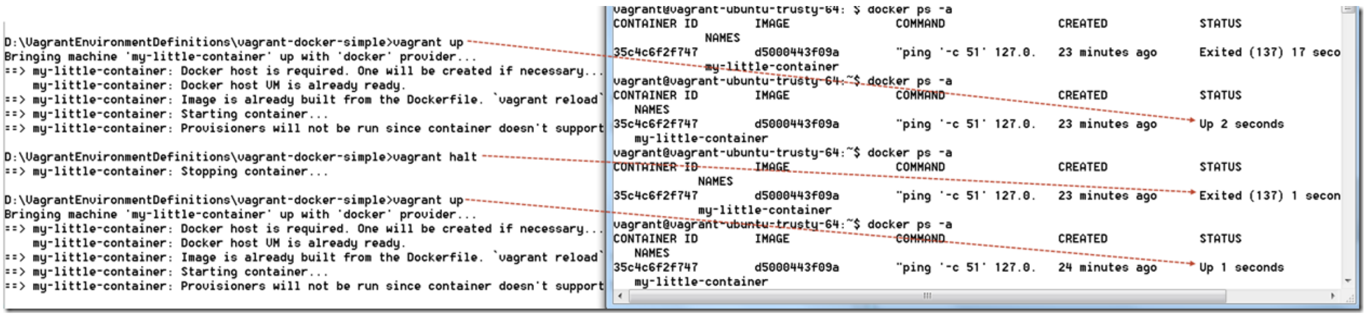
In this session I again verify that the readme.txt has been copied to a newly created directory /u01 inside the container my-little-container. Note that our second session is killed when the initial session (process 1) that ran when the container was started has completed.

Executing commands in a container can also be done from the Vagrant (or Windows) host, using the vagrant instruction docker-run. This will start a container (not reuse an already running one), execute the command and close the container. To open a bash shell into the my-little-container container from the Windows host, use: *vagrant docker-run -t — bash*.



```
==> my-little-container: Image is already built from the Dockerfile.  vagrant reload  to rebuild.
==> my-little-container: Creating the container...
    my-little-container:   Name: my-little-container_1440252083
    my-little-container:  Image: d5000443f09a
    my-little-container:    Cmd: bash
    my-little-container: Volume: /var/lib/docker/docker_1440251140_61092:/vagrant
    my-little-container:
    my-little-container: Container is starting. Output will stream in below...
    my-little-container:
root@d02f29ea5f90:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:07
          inet addr:172.17.0.7  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

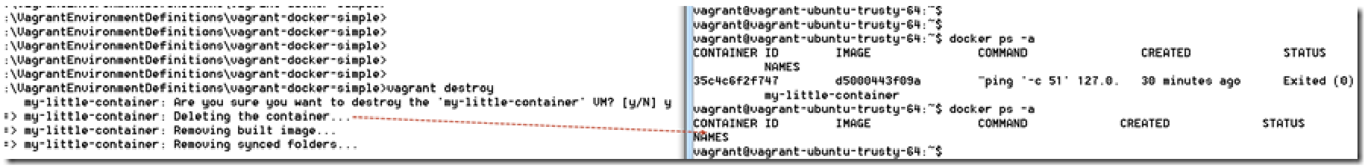
Stopping a running container can be done from the *dockerhostvm* using regular docker commands. From the Vagrant host (the Windows laptop) we can also stop a container, using *vagrant halt*. This next picture shows on the left side the actions on the Windows machine and the status of the Docker containers inspected inside *dockerhostvm* on the right side. It should be obvious how a vagrant up causes the container the run and a vagrant halt stops the container again.



```
D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant up
Bringing machine 'my-little-container' up with 'docker' provider...
==> my-little-container: Docker host is required. One will be created if necessary...
    my-little-container: Docker host VM is already ready.
==> my-little-container: Image is already built from the Dockerfile. `vagrant reload`
==> my-little-container: Starting container...
==> my-little-container: Provisioners will not be run since container doesn't support

D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant halt
==> my-little-container: Stopping container...

D:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant up
Bringing machine 'my-little-container' up with 'docker' provider...
==> my-little-container: Docker host is required. One will be created if necessary...
    my-little-container: Docker host VM is already ready.
==> my-little-container: Image is already built from the Dockerfile. `vagrant reload`
==> my-little-container: Starting container...
==> my-little-container: Provisioners will not be run since container doesn't support
```

```
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
             NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   23 minutes ago      Exited (137) 17 seco
             my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
   NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   23 minutes ago      Up 2 seconds
   my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
             NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   23 minutes ago      Exited (137) 1 secon
             my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
   NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   24 minutes ago      Up 1 seconds
   my-little-container
```

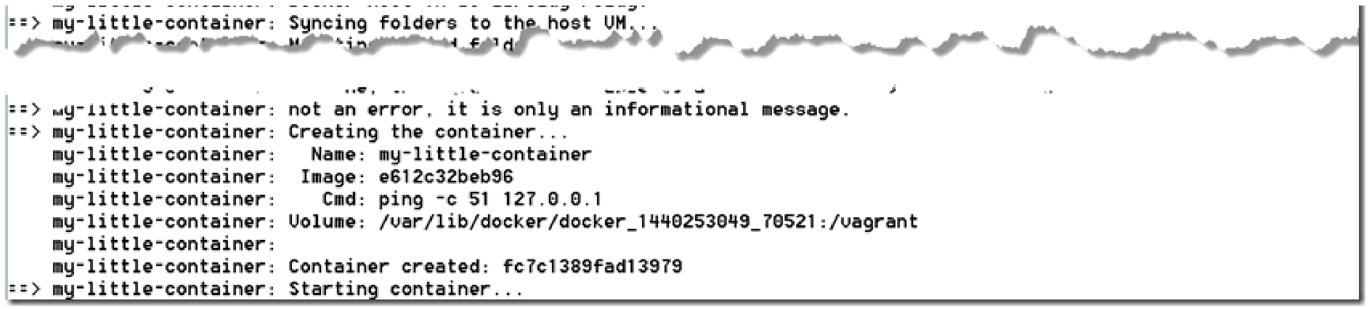
The command vagrant destroy acts on the Docker container – and removes it from the *dockerhostvm*. It can subsequently be restored with a fresh vagrant up:



```
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>
:\VagrantEnvironmentDefinitions\vagrant-docker-simple>vagrant destroy
   my-little-container: Are you sure you want to destroy the 'my-little-container' VM? [y/N] y
=> my-little-container: Deleting the container...
=> my-little-container: Removing built image...
=> my-little-container: Removing synced folders...
```

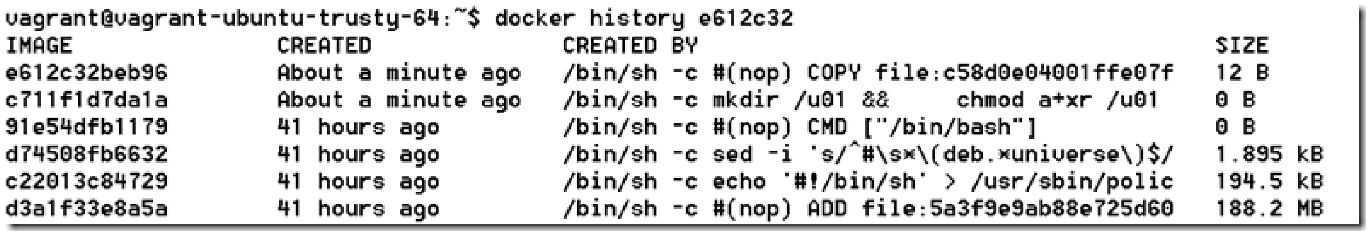
```
vagrant@vagrant-ubuntu-trusty-64:~$
vagrant@vagrant-ubuntu-trusty-64:~$
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS
             NAMES
35c4c6f2f747        d5000443f09a        "ping '-c 51' 127.0.   30 minutes ago      Exited (0)
             my-little-container
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
NAMES
vagrant@vagrant-ubuntu-trusty-64:~$
```

Recreating the container is simple:



```
==> my-little-container: Syncing folders to the host VM...
[illegible]
==> my-little-container: not an error, it is only an informational message.
==> my-little-container: Creating the container...
    my-little-container:   Name: my-little-container
    my-little-container:  Image: e612c32beb96
    my-little-container:    Cmd: ping -c 51 127.0.0.1
    my-little-container: Volume: /var/lib/docker/docker_1440253049_70521:/vagrant
    my-little-container:
    my-little-container: Container created: fc7c1389fad13979
==> my-little-container: Starting container...
```

Note that using the command *docker history <image id>* we can get an insight in the layers that make up (the images that provides the foundation for) our container – including the layers we added ourselves through the statements in the Dockerfile:



```
vagrant@vagrant-ubuntu-trusty-64:~$ docker history e612c32
IMAGE               CREATED             CREATED BY                                      SIZE
e612c32beb96        About a minute ago  /bin/sh -c #(nop) COPY file:c58d0e04001ffe07f   12 B
c711f1d7da1a        About a minute ago  /bin/sh -c mkdir /u01 &&     chmod a+xr /u01    0 B
91e54dfb1179        41 hours ago        /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
d74508fb6632        41 hours ago        /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
c22013c84729        41 hours ago        /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
d3a1f33e8a5a        41 hours ago        /bin/sh -c #(nop) ADD file:5a3f9e9ab88e725d60   188.2 MB
```

Summary

This article showed how you can leverage Vagrant to create and manage a simple Docker container as well as a Virtual Box Linux VM that provides the Docker engine & infrastructure. Execution of simple Docker files was demonstrated including how to make files from the Windows or Vagrant host available in the Docker build context. We have seen the commands to attach to the running container and look what is going on. We have also seen how we can execute command against the Docker container from either the *dockerhostvm* or the Windows/Vagrant host. Some Docker container management and inspection commands where shown – including *inspect* and *history*.

In a next article, I will discuss more complex Docker files that create more interesting Docker containers. I will also discuss more advanced options and operations including the installation of Java, mapping folders (from both the Windows host and the dockerhostvm), using port forwarding to access a Java Web Service running in the Vagrant container from the Windows host, linking containers and using a Data Container to prevent having to create temporary files in the Docker layer structure during the container build process

Resources

Setting up a development environment using Docker and Vagrant – by Mario Loriedo on Blog Zenika

Development Environments With Vagrant and Docker – Christian Soronellas – introducing Vagrant Provisioner to provision Docker Containers (with Nginx and PHP-FPM)

Rails development with Docker and Vagrant Pablo Acuña – very clear overview of getting a Rails and MySQL pair of Docker containers created (built) and running using Vagrant

Rapid Local Development with Vagrant, Docker and Node.js – Kev Zettler

Building Containerized Apps With Vagrant – on Willem's Fizzy Logic

Docker Networking Made Simple or 3 Ways to Connect LXC Containers by Lukas Pustina

Vagrant Documentation – Docker Commands, Basic Usage Docker Provider

A quick introduction to Docker and Using Docker with Vagrant – on Scott's Weblog

Getting Started with Docker on Servers for Hackers.com

Getting Started with the Nginx Web Server by Travis Reddell

Forward Ports from boot2docker using the Vagrant Docker provider – on StackOverflow

Forwarding ports from a docker container with Vagrant 1.6

Vagrant, Docker & CoreOS by Ashley Clarke

vagrant-docker-exec – plugin to run `docker exec` commands from your host

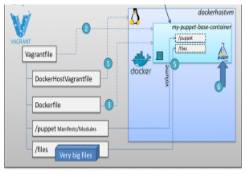
SlideShare presentation Linux containers and docker by Fabio Fumarola

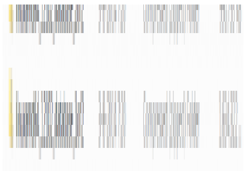
Multi-Service Local Development Environment with Vagrant and Docker by Chris Brantley

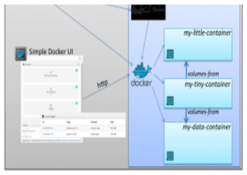
My own earlier articles on Docker: My First Steps with Docker – starting from Windows as host and Docker – Take Two – Starting From Windows with Linux VM as Docker Host

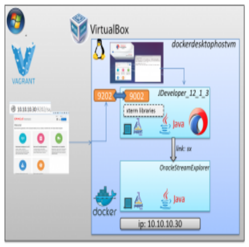
Related Posts:

Vagrant and Docker – Next


Vagrant and Docker


Getting my hands on a


Simple Docker GUI for


Generate Docker

One thought on "First steps with provisioning of Docker containers using Vagrant as provider"

Gaurav Jain says:

December 8, 2016 at 06:29

Thanks for the wonderful article. I have tried the step mentioned above and running into the following problem:

dockerhostvm: /vagrant => D:/workspace/VagrantEnvironmentDefinitions/vagrant-docker-simple
dockerhostvm: Running provisioner: docker…
dockerhostvm: Installing Docker onto machine…
Docker installation failed.

Can you please help? Let me know where I can check the relevant log files for the issue.

Comments are closed.