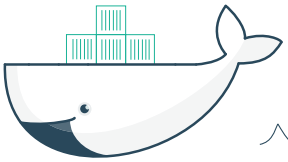
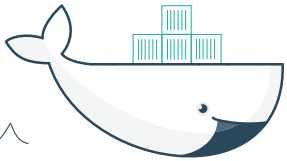
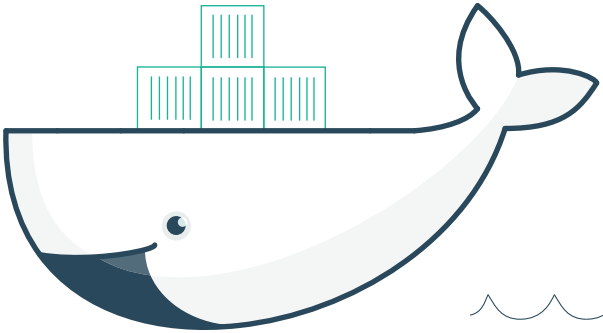


Containers for Everyone

a free ebook from

Linux Academy





Containers are an incredibly powerful technology that can provide you and/or your engineering team with huge productivity gains. Using containers, you can deploy, back up, replicate, and move apps and their dependencies quickly and easily.

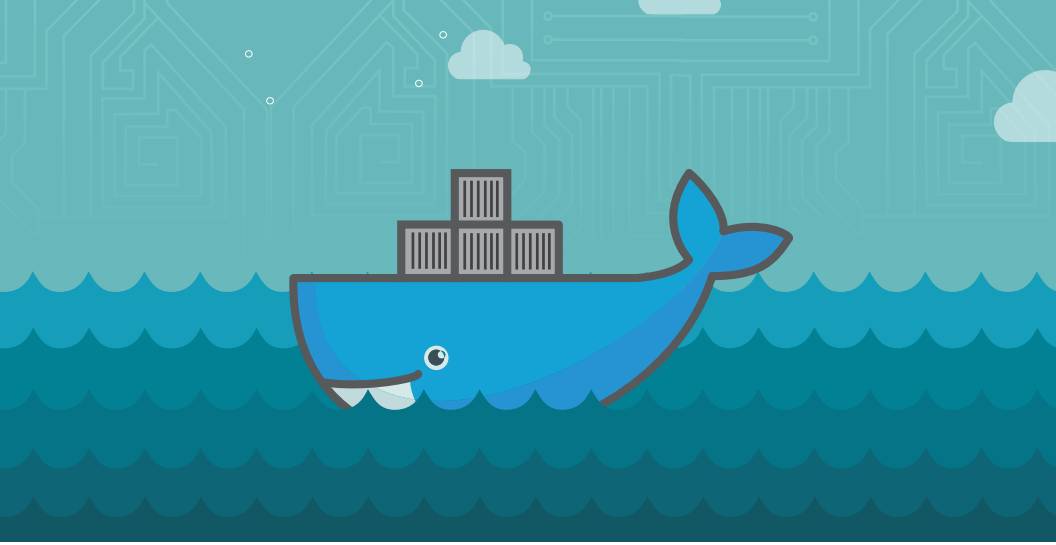
We at Linux Academy are thrilled to share this free ebook about containers and Docker to teach you the basics and to get you up and running quickly.

Keep in mind that if you decide to continue exploring Docker, you can access more advanced training and in-depth courses by joining Linux Academy (more info here). We've linked to several of our members-only courses in this ebook so you can get an idea of what your continued training in Docker should look like. And, if you do want to give it a try, we're always around in our community section to help out and answer questions.

I hope to see you around the community, and have fun learning about containers!

—*Anthony James*, Founder of Linux Academy

Contents



Understanding Containers

A look into what containers are and why you may wish to use them.

Introduction

Containers are a type of virtualization technology that enables the reuse of certain system resources that are not required to be duplicated. This can add a few different benefits depending on your needs. The most significant difference is that you are going to being able to spin up a higher quantity of containers on the same hardware or be able to assign more resources to your containers versus traditional virtual machines.

Starting from a bare-metal server, just the hardware, we need a way to interact with the machine. The Operating System (OS) is comprised of pieces of software that are typically made up of a number of underlying parts. While it's a bit out of scope for this guide to go over nitty gritty details about each part, an understanding of what the parts are and what they do is integral to a working understanding of containers, why they are useful, and exactly how they get the benefits they offer. In this guide, we will go over the components that make up containers and touch a little bit on how your application infrastructure can benefit from them.

Basics of the Operating System (OS)

There are a number of components that are important for an OS to function properly. The parts we are going to focus on are the ones that are handled differently between traditional Virtual Machine (VM) configurations and container implementations.

When power is run to a computer system (typically when the power button is turned on, or a piece of equipment is plugged in) the very first thing that happens is power runs to the NVRAM (Non-Volatile RAM), which is a particular kind of chip that stores a dedicated section of code called the BIOS (Basic Input/Output System). The BIOS is a set of code that loads a basic set of drivers that don't need to be changed often. The BIOS is the reason you are able to use your keyboard and anything else that interfaces with the computer before the actual operating system is loaded. The BIOS is configured to look for an MBR (Master Boot Record) in the first section of a formatted drive.

Note: Network/PXE booting is beyond the scope of this article.

Despite not *technically* being part of the operating system,

we can think of the bootloader as the first part of the OS. The bootloader sits on a special section of the boot disk called the MBR (Master Boot Record) and contains executable code that kicks off the booting process of the OS code. Some popular bootloaders that you may have seen are LILO (LInux LOader), LOADLIN (LOAD LINux), GRUB (GRand Unified Bootloader) and of course, the Windows bootloader. Bootloaders can point to multiple operating systems, or can even be chained together in some cases. This is most commonly done when dual-booting an OS where modifying the bootloader can cause problems. For example, running GRUB to run Linux, but passing to the Windows bootloader if the Windows OS is selected from the GRUB menu).

Note: In modern systems that use a BIOS alternative named UEFI (Unified Extensible Firmware Interface), the MBR is replaced with GPT (GUID Partition Table). This functions almost the same from a top-down perspective, the difference being MBR sits on the first part of a drive, and GPT locations are stored in UEFI.

The function of the bootloader is to load the OS Kernel. The Kernel is the core set of code that handles interfacing with the lower levels of the system and allows all of the

abstraction that modern operating systems offer, such as graphical windowing systems (commonly referred to as GUI's, Desktop Environments, or Window Managers, depending on which part is being referenced).

The kernel of an operating system contains the core group of libraries and binaries required to run the OS. This set of code includes all of the underlying tools used by every other application running under the OS. This combination of code and resources is commonly referred to as Bins/Libs, for binaries and libraries.

Where Containers Shine

In order to execute the bootloader and operating system in a traditional virtual machine environment, a Virtualization layer sits on top of the Host OS (the Operating system that you have installed onto the physical device). This layer then emulates hardware, presenting a virtualized version of a set of hardware to the Guest OS (the OS being installed inside of the Virtualization Layer).

The virtualization layer essentially presents a fake version

of physical hardware, copying how the hardware would usually function as closely as possible. This means you are presenting an emulated storage device for the bootloader, kernel, the OS bins/libs, and any other default applications installed by the Guest OS.

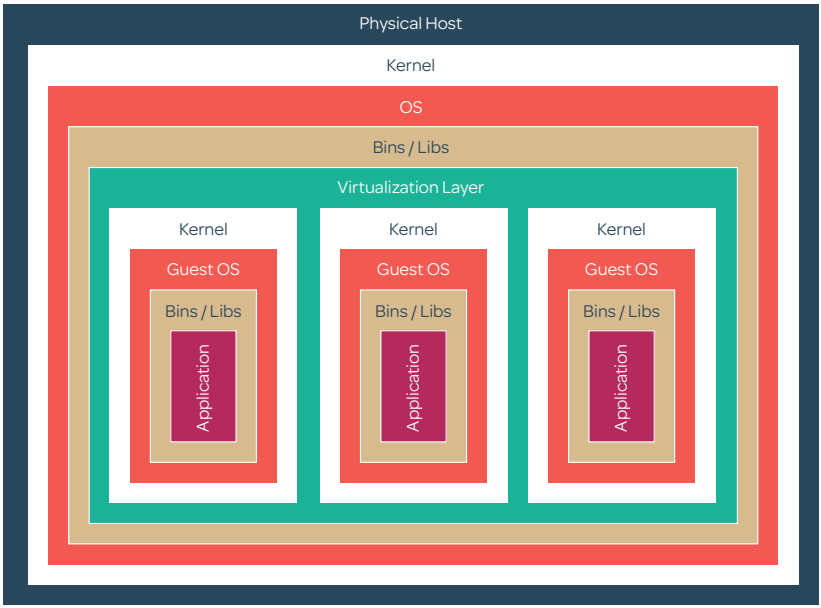
Please note: Below is a simple diagram, but there will be differences depending on the type of virtualization. Notably paravirtualization, which is a specialized kernel that is configured to send commands to a hypervisor rather than directly to hardware. The nuanced differences are difficult to explain, but the core differences come down to the following. If the guest is aware that it is running in a virtualized environment, it purposefully routes requests to a hypervisor. If instead, the Host OS makes special allocations to run a higher execution ring, it allows the guest to execute in ring 0.

If this doesn't make too much sense to you at this point, do not worry. If you would like to read more about the topic, you can look up "protection rings." More specifically, "hierarchical protection domains" relative to computer science.

This is really where containers shine:

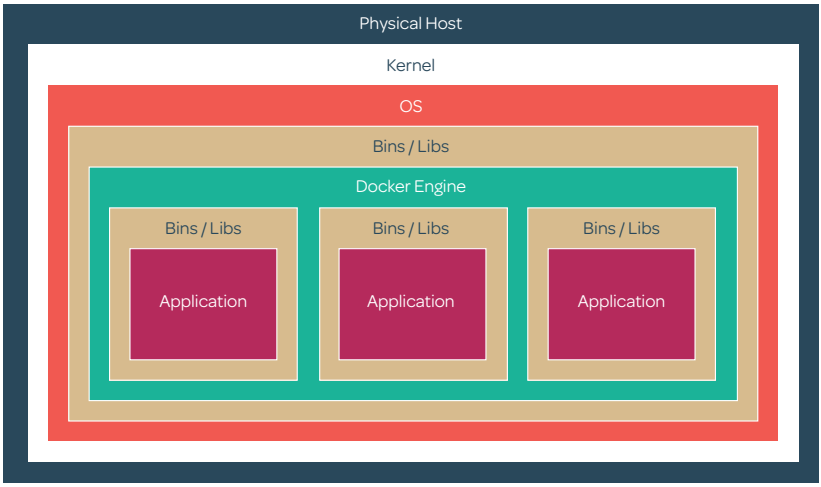
When you load a container layer, such as Docker Engine or Google Container Engine, the engine does not provide a typical virtualization.

Traditional Virtualization



Instead, it provides the guest OS a wrapper to access the Host OS's existing kernel, scheduler, and memory manager:

Container Architecture



In reality, containers are not running a virtualized OS at all. Instead, they are allowed limited access to the existing kernel, binaries, and libraries that exist on the host operating system. The engine loads any additional bins/libs that are required by the application and groups the running processes of the container together, very similarly to the way that chroot jails function.

The reason containers can run in this manner (as opposed to traditional styles of virtualization) is in large part due to cgroups and namespaces. If you have an understanding of Object Oriented Programming (OOP), then you will likely find the way containers group resources and access them to be

quite intuitive. If you don't have an understanding of OOP concepts, you do not necessarily need to. However, a lot of the concepts carry over quite nicely.

Namespaces and cgroups

Namespaces and cgroups are a way of grouping system resources. In Linux, the Namespace wraps global resources in a layer that allows access to those resources in a manner that separates calls to the resources into groups. We call groupings with such an implementation cgroups, which are a particular namespace themselves, specifically handling resource grouping. The key to container technology is that access to resources within a namespace is limited to those processes that belong to a cgroup within the namespace. On a more technical note, namespaces can be found on the filesystem just like other Linux resources. If you know the process you are trying to investigate, you can view the namespaces and cgroups the process is a part of. This allows a lot of deep level investigation if you were interested in the system calls and nitty details going on behind the scenes. Those interested can find this information in:

Namespace	Constant	FS path	Function
Cgroup	`CLONE_NEWCGROUP`	`/proc/[PID]/[namespace]/cgroup`	Cgroup root directory
IPC	`CLONE_NEWIPC`	`/proc/[PID]/[namespace]/ipc`	System V IPC, POSIX message queues
Network	`CLONE_NEWNET`	`/proc/[PID]/[namespace]/net`	Network devices, stacks, ports, etc.
Mount	`CLONE_NEWNS`	`/proc/[PID]/[namespace]/mnt`	Mount points
PID	`CLONE_NEWPID`	`/proc/[PID]/[namespace]/pid`	Process IDs
User	`CLONE_NEWUSER`	`/proc/[PID]/[namespace]/user`	User and group IDs
UTS	`CLONE_NEWUTS`	`/proc/[PID]/[namespace]/uts`	Hostname and NIS domain name

For more information regarding Linux Namespaces, you can find the Man page here: Namespaces.

Why it all Matters

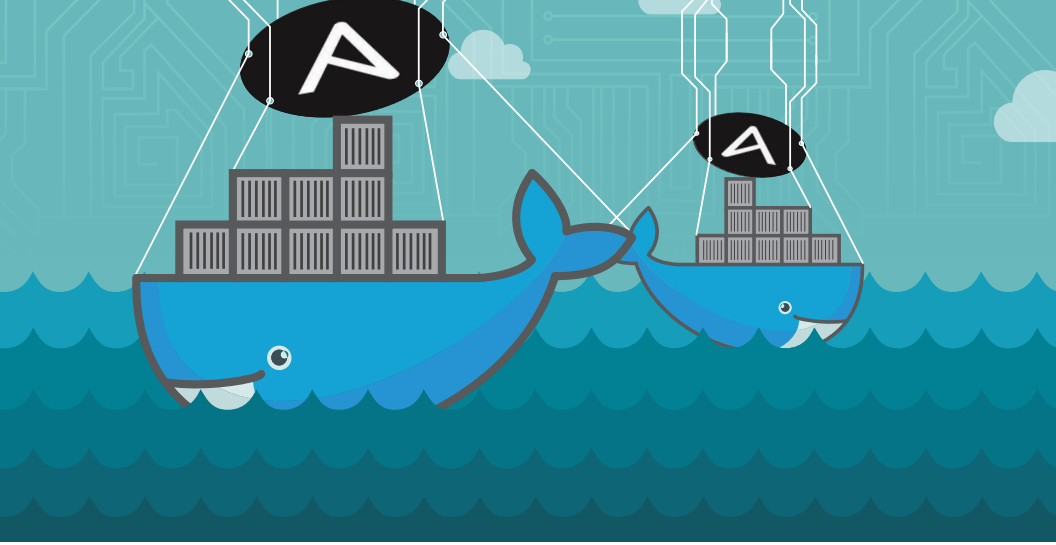
The introduction of Namespaces and cgroups gave way to the ability to share resources without opening up the application's memory space to being read by other processes needing access to the same resource. This was a concern prior to the two, which led to virtualization technologies evolving the way that they did. The best example of what happens when you don't have resource

namespaces can be seen with hypervisor style virtualization, such as Xen and VMware ESXi. These VM technologies make use of a hypervisor layer, which is essentially doing the work that namespaces do.

One way or the other, running multiple applications on the same hardware is going to require access to resources that the other applications use at some point. Hypervisors allow access to these resources by sitting between an entire copy of a Guest OS and the hardware that the guest needs to execute its code on. It takes calls from the guest, making sure the calls are sandboxed from other Guest OS's, ultimately executing the requested code on the provided hardware, and finally returning the results back to the Guest that requested it.

On the flip side, containers make an effort to use the same resources rather than duplicating all of them, instead electing to use built-in functions to limit access to the resources in a reasonable manner. This is an entirely different way to think about virtualization and resource sharing, and thus allows different ways of doing things that can benefit anything from the prototyping process to improving the entirety of your companies operations

structure. Containers are a tool to be used in conjunction with other aspects of the technological spectrum. Their ease of use, customization, and ephemerality make them an irreplaceable piece of the modern engineer's toolset.



Managing Docker Containers with Ansible

A step-by-step guide to configuring and managing Docker containers with Ansible.

Docker and Ansible Overview

As leading DevOps technologies, Docker and Ansible complement each other very well. Docker is used as a way to encapsulate applications in predictable environments within a lightweight container, while Ansible can be used to configure the host server to support and orchestrate Docker deployments. In this guide, we will walk through steps to configure and manage Docker containers with Ansible.

Technology Note

The guide is drafted based on the current course material at Linux Academy, which allows you to get up to speed fairly quickly. However, as Docker tends to evolve very rapidly due to heavy development, there is a chance that whatever Docker implementation you are using could differ from the course. If that's the case, we encourage you to post comments in our community forum if you have questions about the guide.

Requirements

The following skills would be useful in following this guide, but aren't strictly necessary:

- Managing Linux or Unix-like systems (we will be mostly working within the terminal).
- Having a working knowledge of Docker.
- A basic understanding of Ansible.

Feel free to take advantage of the following resources available from Linux Academy to get up to speed:

- Linux Essentials Certification
- Docker Quick Start
- Docker Deep Dive
- Ansible Quick Start
- Using Ansible for Configuration Management and Deployments

In addition, you should have the following:

- A workstation environment with Docker and Ansible installed.
- An OS server that supports Docker. Note that for this guide, we are using the following deployment configuration:
  - Ubuntu 16.04

    - » SSH access via the user Ubuntu with password-less sudo privileges
- A Docker Hub account. Instructions can be found here.

Assuming that you have all of that, you may proceed.

Docker Development

As a first step, at the base of your home directory, create the directory `src/docker/demo` and then change over to that path.

```
stardust:~ rilindo$ mkdir -p src/docker/demo
stardust:~ rilindo$ cd src/docker/demo
```

Next, we will download the latest version of CentOS from Docker by running docker pull centos:latest:

```
stardust:demo rilindo$ docker pull centos:latest
```

The output should look similar to:

```
stardust:demo rilindo$ docker pull centos:latest
latest: Pulling from library/centos
8d30e94188e7: Pull complete
Digest: sha256:2ae0d2c881c7123870114fb9cc7afa
bd1e31f9888dac8286884f6cf59373ed9b
Status: Downloaded newer image for centos:latest
```

Now we will test the image by instantiating it with the

command:

```
stardust:demo rilindo$ docker run -d --name
apacheweb1 centos/apache:v1
```

It will return a long uuid string:

```
stardust:demo rilindo$ docker run -d --name
apacheweb1 centos/apache:v1
f4cd168eeef1c4533770523891a71276103c505b6846a52
9cfc8e82bbaf2f75d
```

We should be able to verify that it is running using the command:

```
stardust:demo rilindo$ docker ps -a
```

Output should be similar to the following:

```
stardust:demo rilindo$ docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS
PORTS               NAMES
f4cd168eeef1        centos/apache:v1    “/bin/sh
-c ‘apachect”   9 seconds ago       Up 8 seconds
```

With basic functionality verified, we can go ahead and create a custom Docker image.

Customizing a Docker Image

In the directory `src/docker/demo`, create the following

Dockerfile:

```
FROM centos:latest
Maintainer rilindo.foster@monzell.com
RUN yum update -y
RUN yum install -y httpd net tools
RUN echo “This is a custom index file built
during the image creation” > /var/www/html/
index.html
ENTRYPOINT apachectl “-DFOREGROUND”
```

When we build the image, the following actions will take place:

- Update CentOS to the latest version
- Install the httpd (apache) application and supporting tools
- Create a custom index page
- Setup the container to start Apache in the foreground

Save the file and build the image with this command:

```
docker build -t centos/apache:v1
```

The output should be similar to the following:

```
stardust:demo rilindo$ docker build -t centos/
apache:v1 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM centos:latest
```

```
 ---> 980e0e4c79ec
Step 2 : MAINTAINER rilindo.foster@monzell.com
 ---> Using cache
 ---> 418ea7dd7050
Step 3 : RUN yum update -y
 ---> Using cache
 ---> e2455d5eefd2
Step 4 : RUN yum install -y httpd net tools
 ---> Using cache
 ---> c81b8cd54ce8
Step 5 : RUN echo “This is a custom index file
build during the image creation” > /var/www/
html/index.html
 ---> Using cache
 ---> 362d569c3803
Step 6 : ENTRYPOINT apachectl “-DFOREGROUND”
 ---> Using cache
 ---> abcf05ec382a
Successfully built abcf05ec382a
```

Once the image is finished building, we are going to test it by spinning up a container based off of the image with:

```
docker run -d --name apacheweb1 -p 8080:80
centos/apache:v1
```

When you run the above command, you should be able to see it in the Docker process list by running `docker ps -a`. The output should look similar to this:

```
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS                    NAMES
b049ba2e829f        rilindo/myapacheweb:v1   “/
```

```
bin/sh -c ‘apachect”   4 days ago          Up 4
seconds        0.0.0.0:8080->80/tcp   apacheweb1
stardust:~ rilindo$
```

You should be able to view the custom index page by browsing to it or downloading it with a web utility. In our case, we are able to view the index page by simply using curl:

```
stardust:~ rilindo$ curl localhost:8080
This is a custom index file build during the
image creation
```

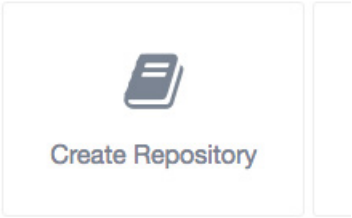
With your setup, you may need to pull the page against the Docker ip, which you can retrieve by parsing for the IPAddress against the output of docker inspect:

```
stardust:~ rilindo$ docker inspect apacheweb1 |
grep IPAddress
            “SecondaryIPAddresses”: null,
            “IPAddress”: “172.17.0.2”,
```

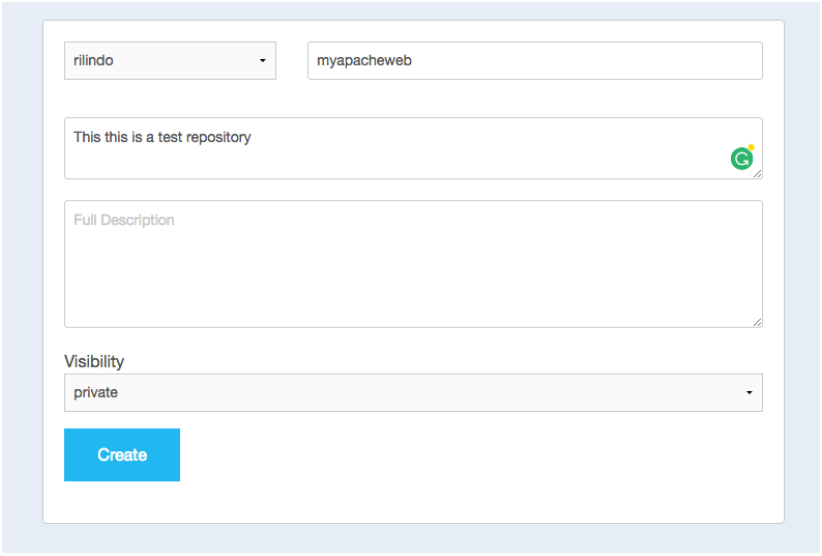
Uploading The Custom Container

Now that you created a custom image, it is time to upload it to a private repository. If you haven’t done so, sign up for a

Docker Hub account. When you are done, click on the button called “Create Registry”:



Then specify the name of the registry, a brief description of the repository, and mark the repository as “private” (we will get to the reasons why in just a moment):



Once you are done, go back to your terminal and run `docker login` to authenticate against your account:

```
stardust:~ rilindo$ docker login
```

Enter your Docker Hub username and password and it should return with Login Succeeded message:

```
Login with your Docker ID to push and pull
images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com
to create one.
Username: rilindo
Password:
Login Succeeded
stardust:~ rilindo$
```

Now list the images you have locally with the command Docker images (it should return most of the images you have locally)

```
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
<none>              <none>
28ade7a75856        23 hours ago        485.4 MB
centos/apache       v1
abcf05ec382a        23 hours ago        485.4 MB
nginx               latest
ba6bed934df2        3 days ago          181.4 MB
centos              latest
980e0e4c79ec        2 weeks ago         196.8 MB
```

We will then tag the image we created using its Image ID that was just returned to us from the previous command. The tag should match the name of the repository, like so:

```
stardust:~ rilindo$ docker tag abcf05ec382a
rilindo/myapacheweb:v1
```

When that is done, upload it with:

```
docker push  rilindo/myapacheweb:v1
```

It will take some time to upload, depending on your bandwidth. When it finishes, it should return with a hash at the end (note the bolded line):

```
stardust:~ rilindo$ docker push  rilindo/
myapacheweb:v1
The push refers to a repository [docker.io/
rilindo/myapacheweb]
abfaedfcb05d: Mounted from rilindo/apache
c8bbf58a3299: Mounted from rilindo/apache
49715a5feaeb: Mounted from rilindo/apache
0aeb287b1ba9: Mounted from rilindo/apache
v1: digest: sha256:22136f81d1938870f19e-
c3a4773595de06
60c39c1645e1a376855ec8d9897a6f size: 1160
```

Congratulations! You have successfully uploaded a Docker image to your account.

Configuring Ansible with Docker

In addition to ensuring that Ansible is installed on your workstation, you also need install the following module in order for Ansible to talk to the docker host:

```
stardust:ansible_docker_demo rilindo$ sudo pip
install 'docker-py>=1.7.0'
Downloading/unpacking docker-py>=1.7.0
  Downloading docker-py-1.10.4.tar.gz (83kB):
83kB downloaded
  Running setup.py egg_info for package
docker-py

Downloading/unpacking requests!=2.11.0,>=2.5.2
(from docker-py>=1.7.0)
  Downloading requests-2.11.1.tar.gz (485kB):
485kB downloaded
  Running setup.py egg_info for package requests

    warning: no files found matching 'test_
requests.py'
Requirement already satisfied (use --upgrade to
upgrade): six>=1.4.0 in /Library/Python/2.7/
site-packages (from docker-py>=1.7.0)
Downloading/unpacking websocket-client>=0.32.0
(from docker-py>=1.7.0)
  Downloading websocket_client-0.37.0.tar.gz
(194kB): 194kB downloaded
  Running setup.py egg_info for package
websocket-client

Downloading/unpacking docker-pycreds>=0.2.1
(from docker-py>=1.7.0)
```

```
  Downloading docker-pycreds-0.2.1.tar.gz
  Running setup.py egg_info for package
docker-pycreds

Installing collected packages: docker-py,
requests, websocket-client, docker-pycreds
  Running setup.py install for docker-py

  Found existing installation: requests 1.2.0
    Uninstalling requests:
      Successfully uninstalled requests
  Running setup.py install for requests

    warning: no files found matching 'test_
requests.py'
  Running setup.py install for websocket-client
    changing mode of build/scripts-2.7/wsdump.py
from 644 to 755

    changing mode of /usr/local/bin/wsdump.py to
755
  Running setup.py install for docker-pycreds

Successfully installed docker-py requests
websocket-client docker-pycreds
Cleaning up...
```

Then for Docker service, we will need to install:

```
stardust:ansible_docker_demo rilindo$ pip
install 'docker-compose>=1.7.0'
Downloading/unpacking docker-compose>=1.7.0
  Downloading docker-compose-1.9.0rc1.tar.gz
(155kB): 155kB downloaded
  Running setup.py egg_info for package
docker-compose
    /System/Library/Frameworks/Python.framework/
```

```
Versions/2.7/Extras/lib/python/setuptools/dist.
py:285: UserWarning: Normalizing '1.9.0-rc1' to
'1.9.0rc1'
      normalized_version,

    warning: no previously-included files found
matching 'README.md'
    warning: no previously-included files
matching '*.pyc' found anywhere in distribution
    warning: no previously-included files
matching '*.pyo' found anywhere in distribution
    warning: no previously-included files
matching '*.un~' found anywhere in distribution
Downloading/unpacking cached-property<2,>=1.2.0
(from docker-compose>=1.7.0)
  Downloading cached-property-1.3.0.tar.gz
  Running setup.py egg_info for package
cached-property

    warning: no previously-included files
matching '__pycache__' found under directory '*'
    warning: no previously-included files
matching '*.py[co]' found under directory '*'
    warning: no files found matching '*.rst'
under directory 'docs'
    warning: no files found matching 'conf.py'
under directory 'docs'
    warning: no files found matching 'Makefile'
under directory 'docs'
    warning: no files found matching 'make.bat'
under directory 'docs'
Downloading/unpacking docopt<0.7,>=0.6.1 (from
docker-compose>=1.7.0)
  Downloading docopt-0.6.2.tar.gz
  Running setup.py egg_info for package docopt
```

```
Requirement already satisfied (use --upgrade
to upgrade): PyYAML<4,>=3.10 in /Library/
Python/2.7/site-packages (from docker-
compose>=1.7.0)
Requirement already satisfied (use --upgrade to
upgrade): requests!=2.11.0,<2.12,>=2.6.1 in /
Library/Python/2.7/site-packages (from docker-
compose>=1.7.0)
Downloading/unpacking texttable<0.9,>=0.8.1
(from docker-compose>=1.7.0)
  Downloading texttable-0.8.6.tar.gz
  Running setup.py egg_info for package
texttable

Requirement already satisfied (use --upgrade to
upgrade): websocket-client<1.0,>=0.32.0 in /
Library/Python/2.7/site-packages (from docker-
compose>=1.7.0)
Requirement already satisfied (use --upgrade
to upgrade): docker-py<2.0,>=1.10.4 in /
Library/Python/2.7/site-packages (from docker-
compose>=1.7.0)
Downloading/unpacking dockerpty<0.5,>=0.4.1
(from docker-compose>=1.7.0)
  Downloading dockerpty-0.4.1.tar.gz
  Running setup.py egg_info for package
dockerpty

Requirement already satisfied (use --upgrade to
upgrade): six<2,>=1.3.0 in /Library/Python/2.7/
site-packages (from docker-compose>=1.7.0)
Downloading/unpacking jsonschema<3,>=2.5.1 (from
docker-compose>=1.7.0)
  Downloading jsonschema-2.5.1.tar.gz (50kB):
50kB downloaded
  Running setup.py egg_info for package
```

```
jsonschema
    zip_safe flag not set; analyzing archive
contents...

    Installed /private/var/folders/9x/
tsk9k8s56sg570g66qxr240w0000gn/T/
pip-build-rilindo/jsonschema/.eggs/vcversion-
er-2.16.0.0-py2.7.egg

Downloading/unpacking enum34<2,>=1.0.4 (from
docker-compose>=1.7.0)
  Running setup.py egg_info for package enum34

Requirement already satisfied (use --upgrade
to upgrade): docker-pycreds>=0.2.1 in /
Library/Python/2.7/site-packages (from dock-
er-py<2.0,>=1.10.4->docker-compose>=1.7.0)
Installing collected packages: docker-compose,
cached-property, docopt, texttable, dockerpty,
jsonschema, enum34
  Running setup.py install for docker-compose
    /System/Library/Frameworks/Python.framework/
Versions/2.7/Extras/lib/python/setuptools/dist.
py:285: UserWarning: Normalizing '1.9.0-rc1' to
'1.9.0rc1'
      normalized_version,

    warning: no previously-included files found
matching 'README.md'
    warning: no previously-included files
matching '*.pyc' found anywhere in distribution
    warning: no previously-included files
matching '*.pyo' found anywhere in distribution
    warning: no previously-included files
matching '*.un~' found anywhere in distribution
    error: could not create '/Library/
```

```
Python/2.7/site-packages/compose': Permission
denied
    Complete output from command /usr/bin/python
-c "import setuptools;__file__='/private/var/
folders/9x/tsk9k8s56sg570g66qxr240w0000gn/T/
pip-build-rilindo/docker-compose/setup.py';ex-
ec(compile(open(__file__).read().replace('\r\n',
'\n'), __file__, 'exec'))" install --record /var/
folders/9x/tsk9k8s56sg570g66qxr240w0000gn/T/
pip-XnOJ2u-record/install-record.txt --sin-
gle-version-externally-managed:
    /System/Library/Frameworks/Python.framework/
Versions/2.7/Extras/lib/python/setuptools/dist.
py:285: UserWarning: Normalizing '1.9.0-rc1' to
'1.9.0rc1'
  normalized_version,
running install
running build
running build_py
creating build
creating build/lib
creating build/lib/compose
copying compose/__init__.py -> build/lib/compose
copying compose/__main__.py -> build/lib/compose
copying compose/bundle.py -> build/lib/compose
copying compose/const.py -> build/lib/compose
copying compose/container.py -> build/lib/
compose
copying compose/errors.py -> build/lib/compose
copying compose/network.py -> build/lib/compose
copying compose/parallel.py -> build/lib/compose
copying compose/progress_stream.py -> build/lib/
compose
copying compose/project.py -> build/lib/compose
copying compose/service.py -> build/lib/compose
copying compose/state.py -> build/lib/compose
```

```
copying compose/utils.py -> build/lib/compose
copying compose/volume.py -> build/lib/compose
creating build/lib/compose/cli
copying compose/cli/__init__.py -> build/lib/
compose/cli
copying compose/cli/colors.py -> build/lib/
compose/cli
copying compose/cli/command.py -> build/lib/
compose/cli
copying compose/cli/docker_client.py -> build/
lib/compose/cli
copying compose/cli/docopt_command.py -> build/
lib/compose/cli
copying compose/cli/errors.py -> build/lib/
compose/cli
copying compose/cli/formatter.py -> build/lib/
compose/cli
copying compose/cli/log_printer.py -> build/lib/
compose/cli
copying compose/cli/main.py -> build/lib/
compose/cli
copying compose/cli/signals.py -> build/lib/
compose/cli
copying compose/cli/utils.py -> build/lib/
compose/cli
copying compose/cli/verbose_proxy.py -> build/
lib/compose/cli
creating build/lib/compose/config
copying compose/config/__init__.py -> build/lib/
compose/config
copying compose/config/config.py -> build/lib/
compose/config
copying compose/config/environment.py -> build/
lib/compose/config
copying compose/config/errors.py -> build/lib/
compose/config
```

```
copying compose/config/interpolation.py -> build/
lib/compose/config
copying compose/config/serialize.py -> build/lib/
compose/config
copying compose/config/sort_services.py -> build/
lib/compose/config
copying compose/config/types.py -> build/lib/
compose/config
copying compose/config/validation.py -> build/
lib/compose/config
running egg_info
writing requirements to docker_compose.egg-info/
requires.txt
writing docker_compose.egg-info/PKG-INFO
writing top-level names to docker_compose.
egg-info/top_level.txt
writing dependency_links to docker_compose.
egg-info/dependency_links.txt
writing entry points to docker_compose.egg-info/
entry_points.txt
warning: manifest_maker: standard file '-c' not
found
reading manifest file 'docker_compose.egg-info/
SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files found
matching 'README.md'
warning: no previously-included files matching
'*.pyc' found anywhere in distribution
warning: no previously-included files matching
'*.pyo' found anywhere in distribution
warning: no previously-included files matching
'*.un~' found anywhere in distribution
writing manifest file 'docker_compose.egg-info/
SOURCES.txt'
copying compose/GITSHA -> build/lib/compose
```

```
copying compose/config/config_schema_v1.json ->
build/lib/compose/config
copying compose/config/config_schema_v2.0.json ->
build/lib/compose/config
copying compose/config/config_schema_v2.1.json ->
build/lib/compose/config
running install_lib
creating /Library/Python/2.7/site-packages/
compose
error: could not create '/Library/Python/2.7/
site-packages/compose': Permission denied
----------------------------------------
Command /usr/bin/python -c "import
setuptools;__file__='/private/var/folders/9x/
tsk9k8s56sg570g66qxr240w0000gn/T/
pip-build-rilindo/docker-compose/setup.py';ex-
ec(compile(open(__file__).read().replace('\r\n',
'\n'), __file__, 'exec'))" install --record /var/
folders/9x/tsk9k8s56sg570g66qxr240w0000gn/T/
pip-XnOJ2u-record/install-record.txt --sin-
gle-version-externally-managed failed with error
code 1 in /private/var/folders/9x/tsk9k8s56s-
g570g66qxr240w0000gn/T/pip-build-rilindo/
docker-compose
Storing complete log in /var/folders/9x/
tsk9k8s56sg570g66qxr240w0000gn/T/tmpgVicV2
```

With that done, we'll go ahead and create a working directory for us to write our ansible code:

```
stardust:~ rilindo$ mkdir src/ansible_docker_
demo
stardust:~ rilindo$ cd src/ansible_docker_demo/
```

Next, we will create a role directory:

```
stardust:ansible_docker_demo rilindo$ mkdir
roles
```

Then we will create a configuration file that tells ansible to look into that role directory path:

```
[defaults]
roles_path=~/src/ansible_docker_demo/roles
```

And then now we will point ansible to that configuration file:

```
stardust:ansible_docker_demo rilindo$ export
ANSIBLE_CONFIG=$(pwd)/ansible.cfg
```

Verify that the variable is set:

```
stardust:ansible_docker_demo rilindo$ echo
$ANSIBLE_CONFIG
/Users/rilindo/src/ansible_docker_demo/ansible.
cfg
```

Now we will create an inventory. Create a file called hosts and put in the IP of the docker host (in this case, 192.168.64.7), then set the ANSIBLE_HOST variable:

```
stardust:ansible_docker_demo rilindo$ export
ANSIBLE_HOSTS=$(pwd)/hosts
```

Verify that it is set:

```
stardust:ansible_docker_demo rilindo$ echo
$ANSIBLE_HOSTS
/Users/rilindo/src/ansible_docker_demo/hosts
```

Installing Docker on the Host Server

By itself, Ansible will not install docker on the host server, so we'll need to write up some code to go it for us. Fortunately, we can use one of the roles available online. In this case, we'll use this role here:

- https://github.com/angstwad/docker.ubuntu

Create a file call `requirements.yml` with the following content:

```
- src: https://github.com/angstwad/docker.ubuntu
  name: angstwad.docker.ubuntu
  version: master
```

Run the ansible-galaxy command to install the role:

```
stardust:ansible_docker_demo rilindo$
ansible-galaxy install -r requirements.yml
- extracting angstwad.docker.ubuntu to /Users/
rilindo/src/ansible_docker_demo/roles/angstwad.
docker.ubuntu
- angstwad.docker.ubuntu was installed
successfully
stardust:ansible_docker_demo rilindo$
```

Now we will create playbook to install the docker on the host.

Create a playbook to install docker with:

```
---
- hosts: all
  vars:
    docker_opts: >
      -H unix://
      -H tcp://0.0.0.0:2375
      --log-level=debug
  remote_user: ubuntu
  become: yes
  become_method: sudo
  roles:
    - angstwad.docker.ubuntu
```

What is will do is to install Docker and then and expose the docker port API. This will allow ansible to directly communicate with the Docker API to create and destroy docker instances.

Verify that the syntax is correct with ansible-playbook –syntax-check:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check install_ansible.
yml
statically included: /Users/rilindo/src/ansible_
docker_demo/roles/angstwad.docker.ubuntu/tasks/
kernel_check_and_update.yml
playbook: install_ansible.yml
```

Since we did not see any error, we will run the play. The following output should appear:

```
stardust:ansible_docker_demo rilindo$
```

```
ansible-playbook install_ansible.yml
statically included: /Users/rilindo/src/ansible_
docker_demo/roles/angstwad.docker.ubuntu/tasks/
kernel_check_and_update.yml
PLAY [all] ********************************************
*******************************
TASK [setup] ******************************************
*******************************
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Fail if not a new
release of Ubuntu] ************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Fail if not a new
release of Debian] ************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Install
backported trusty kernel onto 12.04] ****
skipping: [192.168.64.8] => (item=[])
TASK [angstwad.docker.ubuntu : Install Xorg
packages for backported kernels (very optional)]
***
skipping: [192.168.64.8] => (item=[])
TASK [angstwad.docker.ubuntu : Install latest
kernel for Ubuntu 13.04+] ********
skipping: [192.168.64.8] => (item=[])
TASK [angstwad.docker.ubuntu : Install
cgroup-lite for Ubuntu 13.10] ************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Reboot instance]
*******************************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Wait for instance
to come online (10 minute timeout)] ***
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Install dmsetup
for Ubuntu 16.04] ***************
```

```
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Run dmsetup for
Ubuntu 16.04] ******************
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Add Docker
repository key] **********************
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Alternative | Add
Docker repository key] ********
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : HTTPS APT
transport for Docker repository] ******
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Add Docker
repository and update apt cache] *****
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Install (or
update) docker package] *************
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Set systemd
playbook var] ***********************
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Set systemd
playbook var] ***********************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Set docker daemon
options] ***********************
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Create systemd
configuration directory for Docker service
(systemd)] ***
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Set docker daemon
options (systemd)] *************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Ensure docker
```

```
daemon options used (systemd)] ****
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Fix DNS in
docker.io] ***************************
skipping: [192.168.64.8]
RUNNING HANDLER [angstwad.docker.ubuntu :
Restart docker] **********************
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : pause] **********
*******************************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Install pip,
python-dev package with apt] *******
changed: [192.168.64.8] => (item=[u'python-dev',
u'python-pip'])
TASK [angstwad.docker.ubuntu : Upgrade latest
pip, setuptools, docker-py and docker-compose
with pip] ***
changed: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'pip', u'install': True})
changed: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'setuptools', u'install':
True})
changed: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'docker-py', u'install':
True})
changed: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'docker-compose',
u'install': True})
TASK [angstwad.docker.ubuntu : Install specific
pip, setuptools, docker-py and docker-compose
with pip] ***
skipping: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'pip', u'install': True})
skipping: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'setuptools', u'install':
```

```
True})
skipping: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'docker-py', u'install':
True})
skipping: [192.168.64.8] => (item={u'version':
u'latest', u'name': u'docker-compose',
u'install': True})
TASK [angstwad.docker.ubuntu : Check if /etc/
updatedb.conf exists] *************
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Ensure updatedb
does not index /var/lib/docker] *
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Check if /etc/
default/ufw exists] ***************
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Change ufw
default forward policy from drop to accept] ***
changed: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Start docker] ***
********************************
ok: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Start docker.io]
********************************
skipping: [192.168.64.8]
TASK [angstwad.docker.ubuntu : Add users to the
docker group] ******************
TASK [angstwad.docker.ubuntu : update facts if
docker0 is not defined] *********
ok: [192.168.64.8]
PLAY RECAP ************************************
********************************
192.168.64.8               : ok=17   changed=10
unreachable=0    failed=0
```

Verify that you are able to access the DOCKER API port. In this case, we'll use netcat to test:

```
stardust:ansible_docker_demo rilindo$ nc -zv
192.168.64.8 2375
found 0 associations
found 1 connections:
     1:  flags=82<CONNECTED,PREFERRED>
   outif bridge100
   src 192.168.64.1 port 55979
   dst 192.168.64.8 port 2375
   rank info not available
   TCP aux info available
Connection to 192.168.64.8 port 2375 [tcp/*]
succeeded!
```

Now we are ready to deploy docker containers.

Managing Docker Images

Let create a playbook called `download_docker_image.yml`:

```
---
- hosts: all
  remote_user: ubuntu
  become: yes
  become_method: sudo
  tasks:
  - name: pull image
    docker_image:
      name: ubuntu
```

Verify the syntax:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check download_docker_
image.yml
playbook: download_docker_image.yml
```

And then run it:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook download_docker_image.yml
PLAY [all] ************************************
*******************************
TASK [setup] **********************************
*******************************
ok: [192.168.64.8]
TASK [pull image] *****************************
*******************************
changed: [192.168.64.8]
PLAY RECAP ************************************
*******************************
192.168.64.8 : ok=2 changed=1 unreachable=0
failed=0
```

You should be able to verify that the image has been downloaded from the docker server:

```
ubuntu@ubuntu:~$ sudo docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu latest f753707788c5 2 weeks ago 127.2 MB
ubuntu@ubuntu:~$
```

Let us move on to creating and destroying containers.

Running and Removing Containers

Create the file create_docker_container.yml with the following code:

```
---
- hosts: all
  remote_user: ubuntu
  become: yes
  become_method: sudo
  tasks:
  - name: create docker container
    docker_container:
      name: mycontainer
      image: ubuntu
      state: started
```

This will create a new container called `mycontainer` using the Ubuntu image.

Check for syntax errors:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check create_docker_
container.yml
playbook: create_docker_container.yml
```

And then run it:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook create_docker_container.yml

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

TASK [setup] ******************************************
*******************************
ok: [192.168.64.8]

TASK [create docker container] ************************
*******************************
changed: [192.168.64.8]

PLAY RECAP ********************************************
*******************************
192.168.64.8               : ok=2    changed=1
unreachable=0    failed=0
```

The fact that it says changed in the task tell us that it is able to create the container. Sure enough, when we logged in, we could see the container in `docker ps -a` output:

```
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS
PORTS               NAMES
ce6a6e6662a4        ubuntu              “/bin/
bash”         4 seconds ago       Exited (0) 3
seconds ago                       mycontainer
```

Now we will remove it. Create the file call `destroy_docker_container.yml` with the following:

```
---
```

```
- hosts: all
  remote_user: ubuntu
  become: yes
  become_method: sudo
  tasks:
  - name: create docker container
    docker_container:
      name: mycontainer
      image: ubuntu
      state: absent
```

The only difference with this and `create_docker_container.yml` is that we are now setting the state to absent, meaning that the container should no longer exist.

Once again, validate the syntax:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check destroy_docker_
container.yml

playbook: destroy_docker_container.yml
stardust:ansible_docker_demo rilindo$
```

And then run it:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook destroy_docker_container.yml

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

TASK [setup] **********************************
*******************************
ok: [192.168.64.8]
```

```
TASK [create docker container] ****************
*******************************
changed: [192.168.64.8]

PLAY RECAP ************************************
*******************************
192.168.64.8               : ok=2    changed=1
unreachable=0    failed=0
```

Where it says changed, it means that it able to remove the container. Sure enough, when we run `docker ps -a`, the container no longer exists:

```
root@ubuntu:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
root@ubuntu:~#
```

At this point, let us now deploy our custom container.

Deploying and Destroying Custom Docker Containers

Since we need to login to Docker registry, we need to be able to put our secrets in the code. To avoid exposing credentials

in the clear, we will use ansible-vault to encrypt them. Create a file called `secrets.yml` with your login info:

```
---
docker_hub_username: yourusername
docker_hub_password: yourpassword
docker_hub_email: youremail@example.com
```

Run ansible-vault against the file. It will prompt for a password to encrypt it:

```
stardust:ansible_docker_demo rilindo$
ansible-vault encrypt secret.yml
New Vault password:
Confirm New Vault password:
Encryption successful
```

The file will look something like this:

```
$ANSIBLE_VAULT;1.1;AES256
366266663332616338343134383362643835343530366331
353338376330353034323661363536 32
383933636230393439346264366536643865343263323230
0a336264353063316136663433343435
393932313338646632303738363038656161633530343232
623330353633363335373265333932
386334353263616166 0a6534356231353364366565333466
326261326561306331363565653031 39
3338
```

Now create the file called `create_custom_docker_container.yml` with the following code:

```
---
```

```
- hosts: all
  remote_user: ubuntu
  vars_files:
    - secret.yml
  become: yes
  become_method: sudo
  tasks:
  - name: login to docker registry
    docker_login:
      username: “{{ docker_hub_username }}”
      password: “{{ docker_hub_password }}”
      email: “{{ docker_hub_email }}”
  - name: create custom docker container
    docker_container:
      name: mycustomcontainer
      image: rilindo/myapacheweb:v1
      state: started
      exposed_ports:
        - 80
      ports:
        “80:80”
```

This will pull the credentials from your encrypted secrets file, login into the docker container and then download our container image, create it and expose the port for access.

Now validate the syntax. Because the credentials are now encrypted, you need to pass the `--ask-vault-pass` parameter to the syntax command. If you do not, you will get the following error:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check create_custom_
docker_container.yml
ERROR! Decryption failed on /Users/rilindo/src/
ansible_docker_demo/secret.yml
```

Otherwise, assuming you did pass the parameter and you entered the correct vault password, the output should return as follows:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook --syntax-check create_custom_
docker_container.yml --ask-vault-pass
Vault password:

playbook: create_custom_docker_container.yml
```

With no errors, run the play (make sure that you pass the `--ask-vault-pass` to decrypt credentials. It should return with:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook create_custom_docker_container.
yml --ask-vault-pass
Vault password:

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

TASK [setup] *********************************
*******************************
ok: [192.168.64.8]
```

```
TASK [login to docker registry] ****************
*******************************
changed: [192.168.64.8]

TASK [create custom docker container] **********
*******************************
changed: [192.168.64.8]

PLAY RECAP *************************************
*******************************
192.168.64.8               : ok=3    changed=2
unreachable=0    failed=0

stardust:ansible_docker_demo rilindo$
```

You should be able to confirm that the customer container is running on the docker host:

```
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS                NAMES
9adffc6184f1        rilindo/myapacheweb:v1
“/bin/sh -c ‘apachect”   About an hour
ago   Up About an hour    0.0.0.0:80->80/tcp
mycustomcontainer
```

And verify that you can access the web port:

```
ubuntu@ubuntu:~$ curl http://localhost
This is a custom index file build during the
image creation
ubuntu@ubuntu:~$
```

To tear down the docker container, you can do the

reverse of what you have done by copying the file `create_custom_docker_container.yml` to `destroy_custom_docker_container.yml` and change the state line to absent in the task create custom docker container:

```
---
- hosts: all
  remote_user: ubuntu
  vars_files:
    - secret.yml
  become: yes
  become_method: sudo
  tasks:
  - name: login to docker registry
    docker_login:
      username: "{{ docker_hub_username }}"
      password: "{{ docker_hub_password }}"
      email: "{{ docker_hub_email }}"
  - name: create custom docker container
    docker_container:
      name: mycustomcontainer
      image: rilindo/myapacheweb:v1
      state: absent
      exposed_ports:
        - 80
      ports:
        "80:80"
```

Run it and it will remove the docker container:

```
stardust:ansible_docker_demo rilindo$
ansible-playbook destroy_custom_docker_
container.yml --ask-vault-pass
Vault password:
```

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

TASK [setup] *********************************
*******************************
ok: [192.168.64.8]

TASK [login to docker registry] ****************
*******************************
ok: [192.168.64.8]

TASK [create custom docker container] **********
*******************************
changed: [192.168.64.8]

PLAY RECAP ***********************************
*******************************
192.168.64.8               : ok=3    changed=1
unreachable=0    failed=0
```

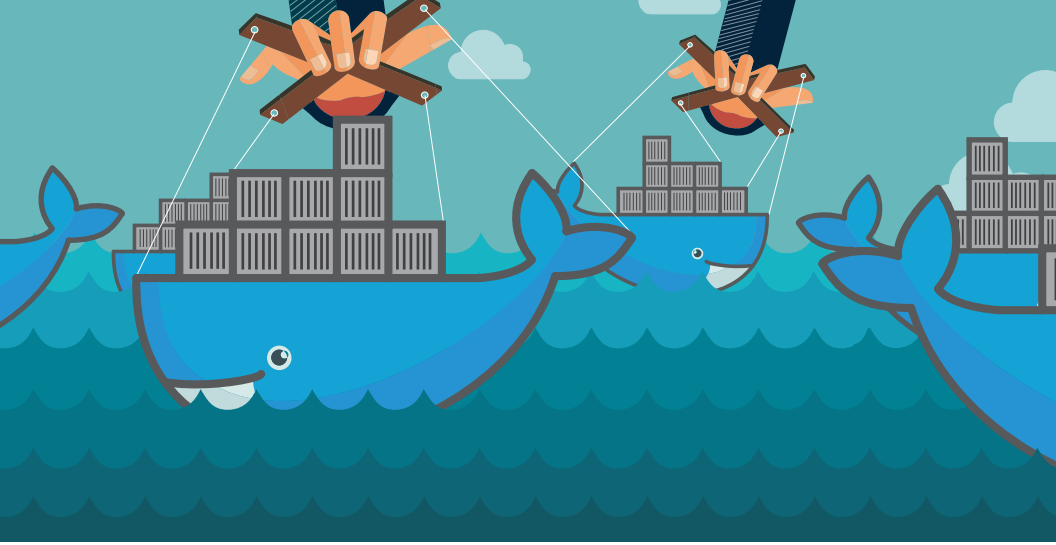
This is how you can manage Docker Containers with Ansible.

Resources

The following Linux Academy resources can be used to assist with following this guide:

- Linux Essentials Certification
- Docker Quick Start
- Docker Deep Dive
- Ansible Quick Start
- Using Ansible for Configuration Management and Deployments

The source code for this guide be found here.



Managing Docker Containers with Puppet

A step-by-step guide to configuring and managing Docker containers with Puppet.

Docker and Puppet Overview

As leading DevOps technologies, Docker and Puppet complement each other very well. Docker is used as a way to encapsulate applications in predictable environments within a lightweight container, while Puppet can be used to configure the host server to support and orchestrate Docker deployments. In this guide, we will walk through steps to configure and manage Docker containers with Puppet.

Technology Note

The guide is drafted based on the current course material at Linux Academy, which allows you to get up to speed fairly quickly. However, as Docker tends to evolve very rapidly due to heavy development, there is a chance that whatever Docker implementation you are using could differ from the course. If that's the case, we encourage you to post comments in our community forum if you have questions about the guide.

Requirements

The following skills would be useful in following this guide, but aren't strictly necessary:

- Managing Linux or Unix-like systems (we will be mostly working within the terminal).
- Having a working knowledge of Docker.
- A basic understanding of Puppet.

Feel free to take advantage of the following resources available from Linux Academy to get up to speed:

- Linux Essentials Certification
- Docker Quick Start
- Docker Deep Dive
- Learning Puppet DevOps Deployment (Puppet Professional Cert)

Other related resources are listed at the end of this guide.

In addition, you should have the following:

- A workstation environment with Docker installed.
- An OS server that:
  - » Supports Docker (in this guide, we are using Ubuntu 14.04)
  - » Has an installed copy of the Puppet agent (we recommend version 4.6.2).
- A Docker Hub account. Instructions can be found here.

Assuming that you have all of that, you may proceed.

Docker Development

As a first step, at the base of your home directory, create the directory `src/docker/demo` and then change over to that path.

```
stardust:~ rilindo$ mkdir -p src/docker/demo
stardust:~ rilindo$ cd src/docker/demo
```

Next, we will download the latest version of CentOS from Docker by running:

```
stardust:demo rilindo$ docker pull centos:latest
```

The output should look similar to:

```
stardust:demo rilindo$ docker pull centos:latest
latest: Pulling from library/centos
8d30e94188e7: Pull complete
Digest: sha256:2ae0d2c881c7123870114fb9cc7afab-
d1e31f9
888dac8286884f6cf59373ed9b
Status: Downloaded newer image for centos:latest
```

Now we will test the image by instantiating it with the command:

```
stardust:demo rilindo$ docker run -d --name
apacheweb1 centos/apache:v1
```

It will return a long uuid string:

```
stardust:demo rilindo$ docker run -d --name
apacheweb1 centos/apache:v1
f4cd168eeef1c4533770523891a71276103c505b6846a529
cfc8e82bbaf2f75d
```

We should be able to verify that it is running using the command:

```
stardust:demo rilindo$ docker ps -a
```

Output should be similar to the following:

```
stardust:demo rilindo$ docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS
PORTS               NAMES
f4cd168eeef1        centos/apache:v1    “/bin/sh
-c ‘apachect”   9 seconds ago       Up 8 seconds
```

With basic functionality verified, we can go ahead and create a custom Docker image.

Customizing a Docker Image

In the directory `src/docker/demo`, create the following Dockerfile:

```
FROM centos:latest
Maintainer rilindo.foster@monzell.com

RUN yum update -y
RUN yum install -y httpd net tools

RUN echo “This is a custom index file built
during the image creation” > /var/www/html/
index.html

ENTRYPOINT apachectl “-DFOREGROUND”
```

When we build the image, the following actions will take place:

- Update CentOS to the latest version
- Install the httpd (apache) application and supporting tools
- Create a custom index page
- Setup the container to start Apache in the foreground

Save the file and build the image with this command:

```
docker build -t centos/apache:v1
```

The output should be similar to the following:

```
stardust:demo rilindo$ docker build -t centos/
apache:v1 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM centos:latest
 ---> 980e0e4c79ec
Step 2 : MAINTAINER rilindo.foster@monzell.com
 ---> Using cache
 ---> 418ea7dd7050
Step 3 : RUN yum update -y
 ---> Using cache
 ---> e2455d5eefd2
Step 4 : RUN yum install -y httpd net tools
 ---> Using cache
 ---> c81b8cd54ce8
Step 5 : RUN echo “This is a custom index file
build during the image creation” > /var/www/
html/index.html
 ---> Using cache
 ---> 362d569c3803
Step 6 : ENTRYPOINT apachectl “-DFOREGROUND”
 ---> Using cache
 ---> abcf05ec382a
Successfully built abcf05ec382a
```

Once the image is finished building, we are going to test it by spinning up a container based off of the image with:

```
docker run -d --name apacheweb1 -p 8080:80
centos/apache:v1
```

When you run the above command, you should be able to see it in the Docker process list by running `docker ps -a`. The output should look similar to this:

```
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS                    NAMES
b049ba2e829f        rilindo/myapacheweb:v1   “/
bin/sh -c ‘apachect”   4 days ago          Up 4
seconds        0.0.0.0:8080->80/tcp   apacheweb1
stardust:~ rilindo$
```

You should be able to view the custom index page by browsing to it or downloading it with a web utility. In our case, we are able to view the index page by simply using curl:

```
stardust:~ rilindo$ curl localhost:8080
This is a custom index file build during the
image creation
```

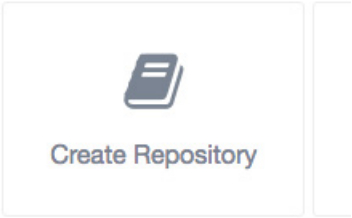
With your setup, you may need to pull the page against the Docker ip, which you can retrieve by parsing for the IPAddress against the output of docker inspect:

```
stardust:~ rilindo$ docker inspect apacheweb1 |
grep IPAddress
            “SecondaryIPAddresses”: null,
            “IPAddress”: “172.17.0.2”,
```

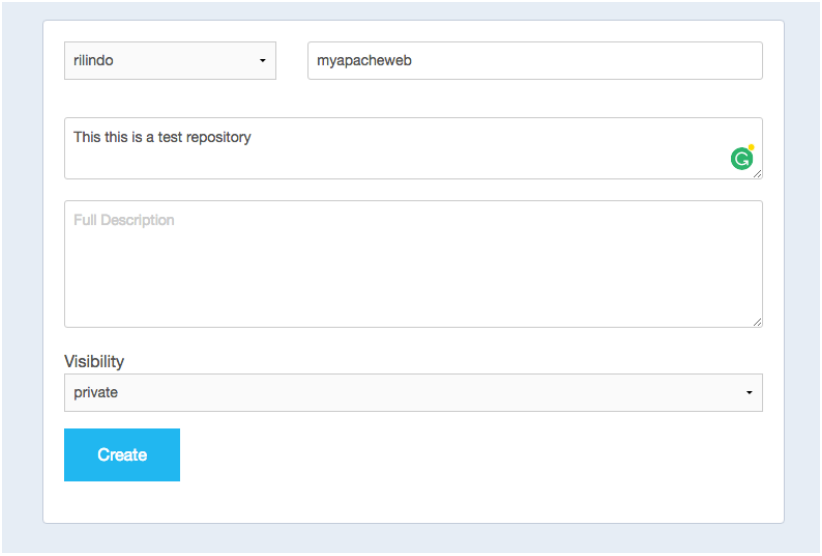
Uploading The Custom Container

Now that you created a custom image, it is time to upload it to a private repository. If you haven’t done so, sign up for a

Docker Hub account. When you are done, click on the button called “Create Registry”:



Then specify the name of the registry, a brief description of the repository, and mark the repository as “private” (we will get to the reasons why in just a moment):



Once you are done, go back to your terminal and run this to authenticate against your account:

```
stardust:~ rilindo$ docker login
```

Enter your Docker Hub username and password and it should return with Login Succeeded message:

```
Login with your Docker ID to push and pull
images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com
to create one.
Username: rilindo
Password:
Login Succeeded
stardust:~ rilindo$
```

Now list the images you have locally with the command Docker images (it should return most of the images you have locally)

```
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
<none>              <none>
28ade7a75856        23 hours ago        485.4 MB
centos/apache       v1
abcf05ec382a        23 hours ago        485.4 MB
nginx               latest
ba6bed934df2        3 days ago          181.4 MB
centos              latest
980e0e4c79ec        2 weeks ago         196.8 MB
```

We will then tag the image we created using its Image ID that was just returned to us from the previous command. The tag should match the name of the repository, like so:

```
stardust:~ rilindo$ docker tag abcf05ec382a
rilindo/myapacheweb:v1
```

When that is done, upload it with:

```
docker push  rilindo/myapacheweb:v1
```

It will take some time to upload, depending on your bandwidth. When it finishes, it should return with a hash at the end (note the bolded line):

```
stardust:~ rilindo$ docker push  rilindo/
myapacheweb:v1
The push refers to a repository [docker.io/
rilindo/myapacheweb]
abfaedfcb05d: Mounted from rilindo/apache
c8bbf58a3299: Mounted from rilindo/apache
49715a5feaeb: Mounted from rilindo/apache
0aeb287b1ba9: Mounted from rilindo/apache
v1: digest: sha256:22136f81d1938870f19ec3a4773
595de0660c39c1645e1a376855ec8d9897a6f size: 1160
```

Congratulations! You have successfully uploaded a Docker image to your account. Now we will go ahead and deploy it on a host server.

Deploying Docker with Puppet on the Host Server

On the host server that contains our Puppet agent, we will first login as root and then install the garethr-docker module:

```
root@ip-10-1-2-3:~# puppet module install
garethr-docker
Notice: Preparing to install into /etc/
puppetlabs/code/environments/production/modules
...
Notice: Downloading from https://forgeapi.
puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/production/
modules
└─┬ garethr-docker (v5.3.0)
  ├── puppetlabs-apt (v2.3.0)
  ├── puppetlabs-stdlib (v4.12.0)
  └── stahnma-epel (v1.2.2)
```

Next, create the directory src/puppet/modules/docker_demo:

```
root@ip-10-1-2-3:~# mkdir -p src/puppet/modules/
docker_demo
```

Change over to that directory and create a puppet manifest to install docker:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# vi install.pp
```

Now, a typical use case for managing Docker with Puppet is to customize the Docker deployment to fit your organization's needs. Therefore, let's use the resource syntax to have Docker installed with a custom DNS server:

```
class { 'docker':
  dns => '8.8.8.8',
}
```

Save the file, running puppet parser validate against the file to ensure that there are no syntax errors:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet parser validate install.pp
```

Assuming that there are no errors, run `puppet apply` against the manifest:

```
puppet apply install.pp
```

The output should be similar to this:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply install.pp
Warning: Scope(Apt::Source[docker]): $include_
src is deprecated and will be removed in the
next major release, please use $include => {
'src' => false } instead
Warning: Scope(Apt::Source[docker]): $required_
packages is deprecated and will be removed in
the next major release, please use package
resources instead.
Warning: Scope(Apt::Source[docker]): $key_source
```

```
is deprecated and will be removed in the next
major release, please use $key => { ‘source’ =>
http://apt.dockerproject.org/gpg } instead.
Warning: Scope(Apt::Key[Add key: 58118E89F3A-
912897C070ADBF76221572C52609D from Apt::Source
docker]): $key_source is deprecated and will be
removed in the next major release. Please use
$source instead.
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.80
seconds
Notice: /Stage[main]/Apt/File[preferences]/
ensure: created
Notice: /Stage[main]/Apt/Apt::Setting[-
conf-update-stamp]/File[/etc/apt/apt.
conf.d/15update-stamp]/content: content changed
‘{md5}b9de0ac9e2c9854b1bb213e362dc4e41’ to
‘{md5}0962d70c4ec78bbfa6f3544ae0c41974’
Notice: /Stage[main]/Docker::Repos/
Apt::Source[docker]/Apt::Key[Add key:
58118E89F3A912897C070ADBF76221572C52609D
from Apt::Source docker]/Apt_key[Add key:
58118E89F3A912897C070ADBF76221572C52609D from
Apt::Source docker]/ensure: created
Notice: /Stage[main]/Docker::Repos/
Apt::Source[docker]/Apt::Pin[docker]/Apt::Set-
ting[pref-docker]/File[/etc/apt/preferences.d/
docker.pref]/ensure: defined content as ‘{md5}
a38a00ab77b0ee306eba00501245d384’
Notice: /Stage[main]/Docker::Repos/
Apt::Source[docker]/Apt::Setting[list-dock-
er]/File[/etc/apt/sources.list.d/docker.list]/
ensure: defined content as ‘{md5}c2bd5681c1f-
d63e3e2630128a815f799’
Notice: /Stage[main]/Apt::Update/Exec[apt_
update]: Triggered ‘refresh’ from 1 events
```

```
Notice: /Stage[main]/Docker::Repos/
Package[cgroup-lite]/ensure: created
Notice: /Stage[main]/Docker::Install/Package[-
linux-image-extra-3.13.0-92-generic]/ensure:
created
Notice: /Stage[main]/Docker::Install/
Package[docker]/ensure: created
Notice: /Stage[main]/Docker::Service/File[/etc/
init.d/docker]/ensure: ensure changed 'file' to
'link'
Notice: /Stage[main]/Docker::Service/File[/
etc/default/docker]/content: content changed
'{md5}ddb6348855959363f63b73853c21f4d0' to
'{md5}1d959149e3fc18aff355ae5a823a5dbf'
Notice: /Stage[main]/Docker::Service/
Service[docker]: Triggered 'refresh' from 2
events
Notice: Applied catalog in 31.10 seconds
```

Managing Docker Images

We're now ready to test by downloading an image. Within the same directory, create a file called `pull_image.pp` with the following content:

```
docker::image { 'ubuntu':
  ensure    => 'present',
  image_tag => 'trusty'
}
```

This downloads an Ubuntu 14.04 image (hence the tag

"trusty").

Validate the code to ensure that there are no syntax errors:

```
puppet parser validate pull_image.pp
```

And then run it to download the image. Again, the output should look similar to the following:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply pull_image.pp
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.23
seconds
Notice: /Stage[main]/Main/Docker::Image[ubun-
tu]/File[/usr/local/bin/update_docker_image.
sh]/ensure: defined content as '{md5}993edb-
514c9469e41aab570e278c04d2'
Notice: /Stage[main]/Main/Docker::Image[ubun-
tu]/Exec[/usr/local/bin/update_docker_image.sh
ubuntu:trusty]/returns: executed successfully
Notice: Applied catalog in 11.12 seconds
```

We should now be able to see the image we downloaded by running docker images:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             SIZE
ubuntu              trusty
f2d8ce9fa988        3 days ago          187.9 MB
```

With that verified, let's delete it. Create a file called `rm_image.pp`, which should have the identify code except that the ensure is now absent:

```
docker::image { 'ubuntu':
  ensure    => 'absent',
  image_tag => 'trusty'
}
```

Once again, verify that there are no errors with:

```
puppet parser validate rm_image.pp
```

And then run it with:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply rm_image.pp
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.21
seconds
Notice: /Stage[main]/Main/Docker::Image[ubuntu]/
Exec[docker rmi ubuntu:trusty]/returns: executed
successfully
Notice: Applied catalog in 0.30 seconds
```

Now if you run docker images, you should that the image we downloaded is no longer present:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
```

Running and Removing Containers

Now that we have verified Docker functionality, let's spin up a container. Create a file in the docker_demo directory called `test_run_container.pp` with the following code:

```
docker::run { 'mycontainertest':
  ensure  => 'present',
  image   => 'ubuntu',
  command => '/bin/sh -c "while true; do echo
test container sleep 1; done"',
}
```

This will run a container called `mycontainertest` using the most recent Ubuntu image (it will download the image if it is not already on the machine); then it will run a basic loop within the image.

After validating that there are no syntax errors with:

```
puppet parser validate test_run_container.pp
```

Go ahead and execute the code:

```
puppet apply test_run_container.pp
```

The output should return with the container in a running state:

```
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.28
```

```
seconds
Notice: /Stage[main]/Main/Docker::Run[-
mycontainertest]/File[/etc/init.d/
docker-mycontainertest]/content: content changed
‘{md5}ec6fcdf1488159431eca235a6773ae0e’ to
‘{md5}4f31d5c215c3f210b472299658e8f76d’
Notice: /Stage[main]/Main/Docker::Run[mycontai
nertest]/Service[docker-mycontainertest]/ensure:
ensure changed ‘stopped’ to ‘running’
Notice: Applied catalog in 0.62 seconds
```

You should be able to verify that the container is running with the `docker ps -a command`:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
a62a52181b57        ubuntu              “/bin/sh
-c ‘while tr”   59 seconds ago      Up 58 second
s                           mycontainertest
```

You should also notice that the container downloaded the Ubuntu image for us:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker images
REPOSITORY            TAG                 IMAGE
ID            CREATED             SIZE
ubuntu                latest
c73a085dc378        6 days ago          127.1 MB
```

Now we will go and remove the container. Create a file called `test_remove_container.pp` and insert the following code:

```
docker::run { 'mycontainertest':
  ensure  => 'absent',
  image   => 'ubuntu',
  command => '/bin/sh -c "while true; do echo
test container sleep 1; done"',
}
```

As you can see, the only difference is that we are now ensuring that the container is "absent" or destroyed.

Again, validate the code with:

```
puppet parser validate test_remove_container.pp
```

And then run it with:

```
puppet apply test_remove_container.pp
```

It should now remove the container, changing the state to "stopped":

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply test_remove_container.pp
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.84
seconds
Notice: /Stage[main]/Main/Docker::Run[mycontain-
ertest]/Service[docker-mycontainertest]/ensure:
ensure changed 'running' to 'stopped'
Notice: Applied catalog in 2.79 seconds
```

The container should no longer be running:

```
root@ ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
```

With that done, we are ready to deploy our custom image.

Deploying and Destroying Custom Docker Containers

A notable use case with Puppet and Docker is to configure the host server to use alternate repositories. This makes it very useful to deploy internal applications that should remain private. This is one way you can do it.

Create a file called `pull_from_private_registry.pp` with this code:

```
docker::registry { ' https://index.docker.io/
v1/:
  username => 'myusername',
  password => 'mypassword',
  email    => 'myusername@example.com',
}

docker::image { 'rilindo/myapacheweb':
  ensure    => 'present',
  image_tag => 'v1',
}
```

What this will do is configure Docker to login to Docker Hub and then download the image we customized and stored in our private repository.

IMPORTANT: As an aside, credentials should not be stored in plaintext within code. We suggest that you should use secrets tools to encrypt your credentials within Puppet. Here is one tool based on hiera.

In any case, once you are done, verify that the there are no syntax errors with:

```
puppet parser validate pull_from_private_image.
pp
```

Run puppet apply against the code:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply pull_from_private_registry.pp
```

The resulting output should look like this:

```
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.22
seconds
Notice: /Stage[main]/Main/Docker::Regis-
try[https://index.docker.io/v1/]/Exec[https://
index.docker.io/v1/ auth]/returns: executed
successfully
Notice: /Stage[main]/Main/Docker::Image[rilindo/
myapacheweb]/Exec[/usr/local/bin/update_docker_
image.sh rilindo/myapacheweb:v1]/returns:
executed successfully
Notice: Applied catalog in 24.03 seconds
```

You should be able to verify that the image is downloaded by running:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker images
REPOSITORY            TAG                 IMAGE
ID            CREATED             SIZE
rilindo/myapacheweb   v1
238570161601        3 days ago          485.4 MB
ubuntu                latest
c73a085dc378        3 days ago          127.1 MB
```

At this point, we will now create a new container from that image and expose access on port 80 with the following code:

```
docker::run { 'mywebserver':
  ensure => 'present',
  image   => 'rilindo/myapacheweb:v1',
  ports  => '80',
  expose => '80',
}
```

Save the code in the file `run_container.pp` and after verifying that there are no syntax errors, run it with:

```
puppet apply run_container.pp
```

The output should look like this:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply run_container.pp
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.24
seconds
Notice: /Stage[main]/Main/Docker::Run[mywebserv-
er]/Service[docker-mywebserver]/ensure: ensure
changed 'stopped' to 'running'
Notice: Applied catalog in 0.29 seconds
```

Run `docker ps -a` to verify that it is running:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker ps -a
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS
NAMES
7d0ac5730696        rilindo/myapacheweb:v1   "/
bin/sh -c 'apachect"   4 seconds ago       Up
3 seconds        0.0.0.0:32770->80/tcp
mywebserver
```

Retrieve the IP of the container with docker inspect:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker inspect mywebserver | grep -i
ipaddress
            “SecondaryIPAddresses”: null,
            “IPAddress”: “172.17.0.2”,
                    “IPAddress”: “172.17.0.2”,
```

And then use curl or another http utility to verify that you can view the custom page

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# curl 172.17.0.2:80
This is a custom index file built during the
image creation
```

With that done, we can now finish by tearing down the container. Create a file called `remove_container.pp` with the following code:

```
docker::run { ‘mywebserver’:
  ensure => ‘absent’,
  image   => ‘rilindo/myapacheweb:v1’,
  ports  => ‘80’,
  expose => ‘80’,
}
```

Once again, verify that there are no syntax errors, then run it. The output should look like this:

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# puppet apply remove_container.pp
Notice: Compiled catalog for ip-10-1-2-3.
monzell.com in environment production in 0.24
```

```
seconds
Notice: /Stage[main]/Main/Docker::Run[mywebserv-
er]/Service[docker-mywebserver]/ensure: ensure
changed ‘running’ to ‘stopped’
Notice: Applied catalog in 0.18 seconds
```

At this point, you should be able to verify that the container has been destroyed.

```
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
root@ip-10-1-2-3:~/src/puppet/modules/docker_
demo#
```

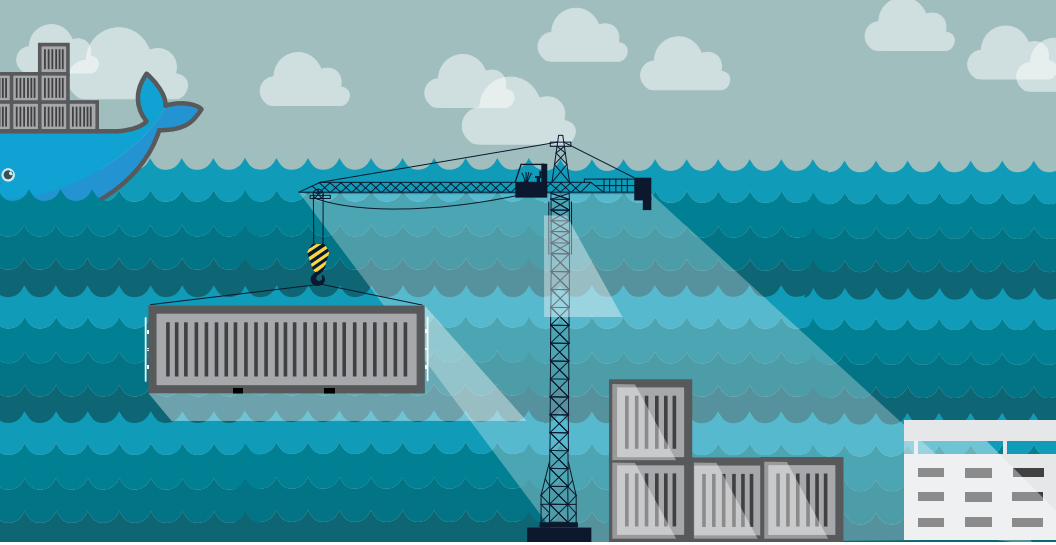
This is how you can manage Docker Containers with Puppet.

Resources

The following Linux Academy resources can be used to assist with following this guide:

- Linux Essentials Certification
- Docker Quick Start
- Docker Deep Dive
- Learning Puppet DevOps Deployment (Puppet Professional Cert)

The Puppet module used in this guide is an approved module by PuppetLabs. Further documentation can be found here.



Setting up LaraDock for Laravel

A step-by-step guide to setting up LaraDock in a Laravel project.

In this guide, you will learn how to set up LaraDock in a Laravel project. LaraDock gives us an easy way to run a Laravel application using Docker in a single command.

Before we dive into the basics of LaraDock, let's gain some context.

What is Laravel?

This guide will not focus on the details of Laravel itself, but you are likely already familiar with it since you're here. The Laravel Philosophy:

> *Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable, creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as authentication, routing, sessions, and caching.*

via Laravel.com

What is Docker?

While this guide's focus is not on the core concepts or use cases of Docker, it's good to have a basic idea of what it

is before we dig in. Docker gives us a nice way to prepare an environment to handle a particular task without being invasive to our main operating system. LaraDock uses Docker and pre-configured containers to do most of its magic, but the power of Docker goes far beyond this example.

According to docker.com:

> *Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.*

For a deep explanation of Docker and its capabilities, check out the Docker Deep Dive course offered by Linux Academy.

What is LaraDock?

Often compared to Laravel Homestead (which uses Vagrant), LaraDock is an open source project that uses Docker to easily prepare the environment you need to run a Laravel application. It's easy to use and can save a developer's valuable time.

For example: If you project needed to use mongoDB and nginx, LaraDock will be able to prepare the necessary containers with the proper configuration using a single command:

```
docker-compose up -d mongo nginx
```

Nice, right?

The LaraDock documentation contrasts its use of Docker with Homestead's use of Vagrant:

> *Vagrant creates Virtual Machines in minutes while Docker creates Virtual Containers in seconds. Instead of providing a full Virtual Machines like you get with Vagrant, Docker provides you lightweight Virtual Containers that share the same kernel and allow to safely execute independent processes.*
>
> *In addition to the speed, Docker gives tons of features that cannot be achieved with Vagrant.*
>
> *Most importantly Docker can run on Development and on Production (same environment everywhere). While Vagrant is designed for Development only, (so you have to re-provision your server on Production every time).*
>
> *Running a virtual Container is much faster than running a full virtual Machine. Thus LaraDock is much faster than Homestead.*

Let's begin

Before we get going, we need to make sure that you have the necessary tools. You will need to have all of these tools installed and ready to use in your terminal, but you most likely already have what you need:

Prerequisites

- `git` - We will use git to clone the LaraDock repository. - Instructions
- `docker` - Docker is required for LaraDock to function. - Instructions
- `composer` - Optional. I will use composer to create a new Laravel project in this guide, so you will need that if you want follow along. If you already have an existing Laravel project, you are good to go. You may also use the Laravel Installer - Instructions

Notice that you do *not* need to manually prepare PHP, databases, or any other typical infrastructure dependencies that a Laravel project might need to have installed in your development environment. LaraDock will handle all of this for us!

For the sake of instruction, I'm going create a new Laravel project to show the setup process. *Even if you already have a Laravel project I would suggest creating a new one along with me to avoid messing anything up while you learn.* If you feel comfortable enough, feel free to use your existing project.

Creating an example Laravel project

Optional, but recommended...

Let's make a fresh Laravel project to play with. I will be using composer for this in the terminal.

1. Navigate to a directory where you want to create the project in terminal. In this example, I will be creating a directory at `~/laravel-projects` and navigating to it:

```
mkdir ~/laravel-projects
cd ~/laravel-projects
```

2. Use `composer` to create a new Laravel project in this directory. I'm going to call my project `example-proj`.

```
composer create-project --prefer-dist laravel/
laravel example-proj
```

You will see a good bit out output while the project is created.

```
Installing laravel/laravel (v5.3.16)
   -     Installing laravel/laravel (v5.3.16)
........
```

You should see "`Application key [base64:...] set successfully`" once it has finished. Run `ls` to make sure you see a new directory for the project (in this case we should see `example-proj`).

Adding LaraDock to Laravel project

Now we can install LaraDock into the project. First, navigate into the project directory and use `git` to install LaraDock. *To avoid conflicts when using LaraDock in multiple projects, let's specify a unique name for its directory in each project.* A good convention is to use "`laradock-PROJECT_NAME`" (so I will use `laradock-example-proj` in this example):

```
cd example-proj
git clone https://github.com/LaraDock/laradock.
git laradock-example-proj
```

Note: If you are installing into an existing Laravel project that

uses `git`, you should use `git submodule add` instead of `git clone` in the snippet above.

You can use `ls` to see the new directory (`laradock-example-proj` in this example) inside of our project folder. We will need to navigate to this folder to use LaraDock. The reason for this should be clear momentarily.

```
cd laradock-example-proj
```

Now we are ready to configure and use LaraDock!

Configuring LaraDock and Laravel

For the sake of example, let's pretend our Laravel project uses `mysql` and `nginx`.

First, we need to tell LaraDock how we want to configure our services (passwords, ports, etc). This is done in a file called `docker-compose.yml`.

Next, we need to tell the Laravel project what those settings were so it can interact with them. This is done in a file called `.env`.

Using the `docker-compose.yml` file

We will be using LaraDock with the `docker-compose` command. This command will adhere to the properties defined in a `docker-compose.yml` file.

The LaraDock developers have prepared the directory that we added in the previous step with a default configuration file that can be modified to suit the project's needs.

Whenever the `docker-compose` command is used in terminal, it will look for the `docker-compose.yml` file in the current working directory. *This is why we need to be inside of the LaraDock directory for the commands to function properly.*

Note: `The docker-compose.yml` file resides inside of LaraDock's directory within our project. In our example project, the configuration file is at `example-proj/lara-dock-example-proj/docker-compose.yml`

We can modify the configuration defined in the `docker-compose.yml` file to fit our needs. Let's open it up in a text editor to poke around. I'll be using the nano editor in terminal:

```
nano docker-compose.yml
```

In practice, you should take the time to configure all of these containers properly, but for this example, let's just look at the MySQL configuration in the `docker-compose.yml` file. Here's the default:

```
MySQL Container ##############################
##########
mysql:
   build: ./mysql
   volumesfrom:
         -     volumesdata
   ports:
         -     “3306:3306”
   environment:
         MYSQLDATABASE: homestead
         MYSQLUSER: homestead
         MYSQLPASSWORD: secret
         MYSQLROOTPASSWORD: root
```

We can modify the these entries to match our desired settings. Later on when we use the docker-compose command to start things up, LaraDock and Docker will set up the mysql container with these settigns. Let's change the database name, user, and password that it will be set up with:

```
MySQL Container ##############################
##########
mysql:
   build: ./mysql
   volumesfrom:
         -     volumesdata
```

```
    ports:
        -     “3306:3306”
    environment:
        MYSQLDATABASE: example-proj
        MYSQLUSER: example-user
        MYSQLPASSWORD: example-pass
        MYSQLROOTPASSWORD: example-root-pass
```

Save the file and exit.

Note: If you are using `nano`, use `control`+`X` and follow the prompt at the bottom of the terminal window. If it fails to save, you may need re-open the file with sudo (`sudo nano docker-compose.yml`) and try again.

Now we need to tell the Laravel project how our services are configured.

Using the `.env` file:

Now that we've set up the configuration for the services, we need to tell Laravel everything it needs to know so it can interact with them. Edit the file called `.env` in the parent directory of the Laravel project (`example-proj/.env` in this example). Since our terminal's current working directory is "one directory deep" into our project, we can get to it with `../.env`

```
nano ../.env
```

You will need to update this information to match what is defined in the `docker-compose.yml` file that we modified in the previous step. Based on our example setup, we want to make sure these entries are set correctly:

```
DBCONNECTION=mysql
DBHOST=127.0.0.1
DBPORT=3306
DBDATABASE=example-proj
DBUSERNAME=example-user
DBPASSWORD=example-pass
```

Save and exit the file.

Starting/Stopping services with LaraDock

Now that we have prepared our configuration files, we are ready to use LaraDock.

First off, ensure that your terminal's working directory is inside of the laradock folder within the project. You can check using the `pwd` command. If not, navigate to it using `cd` in terminal.

Remember that we are pretending that our example Laravel project requires `mysql` and `nginx`.

Let's start them up. We will use the `docker-compose up` command. Let's use the `-d` option to daemonize the services (runs them in the background). We will also specify the services we want to use.

```
docker-compose up -d mysql nginx
```

Note: You need to have the Docker daemon started.

LaraDock will now instruct Docker how to prepare our services. This may take some time while the proper dependencies are obtained and built. After all has completed, you should see something similar to this:

```
Creating laradockexampleproj_volumes_source_1
Creating laradockexampleproj_volumes_data_1
Creating laradockexampleproj_workspace_1
Creating laradockexampleproj_mysql_1
Creating laradockexampleproj_php-fpm_1
Creating laradockexampleproj_nginx_1
```

Guess what! You're ready to use your Laravel project. Open your browser and navigate to `http://localhost/`. You should see the default "Laravel" page from the `example-proj`!

When you're ready to stop the services, head back to Terminal. Be sure you're in the proper directory and run:

```
docker-compose down
```

If you refresh `http://localhost/` in your browser, you should see that the page is no longer available.

LaraDock supports more than just `mysql` and `nginx`. Check out the documentation for more supported services and more commands to make the most out of it.

By Linux Academy

First Edition, April 2017