

GL340
DOCKER
RHEL7

Table of Contents

Chapter 1		Chapter 4	
CONTAINER TECHNOLOGY OVERVIEW	1	CREATING IMAGES WITH DOCKERFILE	1
Application Management Landscape	2	Dockerfile	2
Application Isolation	3	Caching	3
Container Resource Control & Security	5	docker build	4
Container Types	6	Dockerfile Instructions	6
Container Ecosystem	7	ENV and WORKDIR	7
Lab Tasks	8	Running Commands	8
1. Container Concepts LXC	9	Getting Files into the Image	9
2. Container Concepts Systemd	15	Defining Container Executable	10
		Best Practices	11
Chapter 2		Lab Tasks	12
MANAGING CONTAINERS	1	1. Dockerfile Fundamentals	13
Installing Docker	2		
Docker Control Socket	4	Chapter 5	
Creating a New Container	5	DOCKER NETWORKING	1
Listing Containers	6	Overview	2
Viewing Container Operational Details	7	Data-Link Layer Details	3
Running Commands in an Existing Container	8	Network Layer Details	5
Interacting with a Running Container	9	Hostnames and DNS	6
Stopping, Starting, and Removing Containers	10	Local Host <--> Container	7
Lab Tasks	11	Container <--> Container (same node)	8
1. Docker Basics	12	Container <--> Container: Links	9
2. Install Docker via Docker Machine	22	Container <--> Container: Private Network	10
3. Configure a docker container to start at boot.	28	Managing Private Networks	12
		Remote Host <--> Container	13
Chapter 3		Multi-host Networks with Overlay Driver	14
MANAGING IMAGES	1	Lab Tasks	16
Docker Images	2	1. Docker Networking	17
Listing and Removing Images	4	2. Docker Ports and Links	26
Searching for Images	6	3. Multi-host Networks	37
Downloading Images	8		
Committing Changes	9	Chapter 6	
Uploading Images	10	DOCKER VOLUMES	1
Export/Import Images	11	Volume Concepts	2
Save/Load Images	12	Creating and Using Volumes	3
Lab Tasks	13	Managing Volumes (cont.)	4
1. Docker Images	14	Changing Data in Volumes	5
2. Docker Platform Images	24	Removing Volumes	6
		Backing up Volumes	7
		SELinux Considerations	8

Mapping Devices	9
Lab Tasks	10
1. Docker Volumes	11
Chapter 7	
DOCKER COMPOSE/SWARM	1
Concepts	2
Compose CLI	3
Defining a Service Set	4
Docker Swarm	5
Lab Tasks	7
1. Docker Compose	8
2. Docker Swarm	19
Appendix A	
CONTINUOUS INTEGRATION WITH GITLAB, GITLAB CI, AND DOCKER	1
Lab Tasks	2
1. GitLab and GitLab CI Setup	3
2. Unit and Functional Tests	7

Typographic Conventions

The fonts, layout, and typographic conventions of this book have been carefully chosen to increase readability. Please take a moment to familiarize yourself with them.

A Warning and Solution

A common problem with computer training and reference materials is the confusion of the numbers "zero" and "one" with the letters "oh" and "ell". To avoid this confusion, this book uses a fixed-width font that makes each letter and number distinct.

Typefaces Used and Their Meanings

The following typeface conventions have been followed in this book:

`fixed-width normal` ⇒ Used to denote file names and directories. For example, the `/etc/passwd` file or `/etc/sysconfig/directory`. Also used for computer text, particularly command line output.

`fixed-width italic` ⇒ Indicates that a substitution is required. For example, the string `stationX` is commonly used to indicate that the student is expected to replace `X` with his or her own station number, such as `station3`.

`fixed-width bold` ⇒ Used to set apart commands. For example, the `sed` command. Also used to indicate input a user might type on the command line. For example, `ssh -X station3`.

`fixed-width bold italic` ⇒ Used when a substitution is required within a command or user input. For example, `ssh -X stationX`.

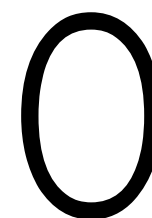
`fixed-width underlined` ⇒ Used to denote URLs. For example, `http://www.gurulabs.com/`.

`variable-width bold` ⇒ Used within labs to indicate a required student action that is not typed on the command line.

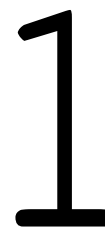
Occasional variations from these conventions occur to increase clarity. This is most apparent in the labs where bold text is only used to indicate commands the student must enter or actions the student must perform.



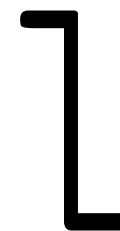
The number
"zero".



The letter
"oh".



The number
"one".



The letter
"ell".

Typographic Conventions

Terms and Definitions

The following format is used to introduce and define a series of terms:

- deprecate** ⇒ To indicate that something is considered obsolete, with the intent of future removal.
- frob** ⇒ To manipulate or adjust, typically for fun, as opposed to tweak.
- grok** ⇒ To understand. Connotes intimate and exhaustive knowledge.
- hork** ⇒ To break, generally beyond hope of repair.
- hosed** ⇒ A metaphor referring to a Cray that crashed after the disconnection of coolant hoses. Upon correction, users were assured the system was rehosed.
- mung (or munge)** ⇒ Mash Until No Good: to modify a file, often irreversibly.
- troll** ⇒ To bait, or provoke, an argument, often targeted towards the newbie. Also used to refer to a person that regularly trolls.
- twiddle** ⇒ To make small, often aimless, changes. Similar to frob.

When discussing a command, this same format is also used to show and describe a list of common or important command options. For example, the following **ssh** options:

- X** ⇒ Enables X11 forwarding. In older versions of OpenSSH that do not include **-Y**, this enables trusted X11 forwarding. In newer versions of OpenSSH, this enables a more secure, limited type of forwarding.
- Y** ⇒ Enables trusted X11 forwarding. Although less secure, trusted forwarding may be required for compatibility with certain programs.

Representing Keyboard Keystrokes

When it is necessary to press a series of keys, the series of keystrokes will be represented without a space between each key. For example, the following means to press the "j" key three times: **j j j**

When it is necessary to press keys at the same time, the combination will be represented with a plus between each key. For example, the following means to press the "ctrl," "alt," and "backspace" keys at the same time:

Ctrl + Alt + Backspace. Uppercase letters are treated the same: **Shift + A**

Line Wrapping

Occasionally content that should be on a single line, such as command line input or URLs, must be broken across multiple lines in order to fit on the page. When this is the case, a special symbol is used to indicate to the reader what has happened. When copying the content, the line breaks should not be included. For example, the following hypothetical PAM configuration should only take two actual lines:

```
password required /lib/security/pam_cracklib.so retry=3,
    type= minlen=12 dcredit=2 ucredit=2 lcredit=0 ocredit=2
password required /lib/security/pam_unix.so use_authtok
```

Representing File Edits

File edits are represented using a consistent layout similar to the unified **diff** format. When a line should be added, it is shown in bold with a plus sign to the left. When a line should be deleted, it is shown struck out with a minus sign to the left. When a line should be modified, it is shown twice. The old version of the line is shown struck out with a minus sign to the left. The new version of the line is shown below the old version, bold and with a plus sign to the left. Unmodified lines are often included to provide context for the edit. For example, the following describes modification of an existing line and addition of a new line to the OpenSSH server configuration file:

File: /etc/ssh/sshd_config	
	#LoginGraceTime 2m
-	#PermitRootLogin yes
+	PermitRootLogin no
+	AllowUsers sjansen
	#StrictModes yes

Note that the standard file edit representation may not be used when it is important that the edit be performed using a specific editor or method. In these rare cases, the editor specific actions will be given instead.

Lab Conventions

Lab Task Headers

Every lab task begins with three standard informational headers: "Objectives," "Requirements," and "Relevance". Some tasks also include a "Notices" section. Each section has a distinct purpose.

Objectives ⇒ An outline of what will be accomplished in the lab task.

Requirements ⇒ A list of requirements for the task. For example, whether it must be performed in the graphical environment, or whether multiple computers are needed for the lab task.

Relevance ⇒ A brief example of how concepts presented in the lab task might be applied in the real world.

Notices ⇒ Special information or warnings **needed** to successfully complete the lab task. For example, unusual prerequisites or common sources of difficulty.

Command Prompts

Though different shells, and distributions, have different prompt characters, examples will use a \$ prompt for commands to be run as a normal user (like guru or visitor), and commands with a # prompt should be run as the root user. For example:

```
$ whoami
guru
$ su -
Password: password
# whoami
root
```

Occasionally the prompt will contain additional information. For example, when portions of a lab task should be performed on two different stations (always of the same distribution), the prompt will be expanded to:

```
stationX$ whoami
guru
stationX$ ssh root@stationY
root@stationY's password: password
stationY# whoami
root
```

Variable Data Substitutions

In some lab tasks, students are required to replace portions of commands with variable data. Variable substitution are represented using italic fonts. For example, *X* and *Y*.

Substitutions are used most often in lab tasks requiring more than one computer. For example, if a student on station4 were working with a student on station2, the lab task would refer to station*X* and station*Y*

```
stationX$ ssh root@stationY
```

and each would be responsible for interpreting the *X* and *Y* as 4 and 2.

```
station4$ ssh root@station2
```

Truncated Command Examples

Command output is occasionally omitted or truncated in examples. There are two type of omissions: complete or partial.

Sometimes the existence of a command's output, and not its content, is all that matters. Other times, a command's output is too variable to reliably represent. In both cases, when a command should produce output, but an example of that output is not provided, the following format is used:

```
$ cat /etc/passwd
. . . output omitted . . .
```

In general, at least a partial output example is included after commands. When example output has been trimmed to include only certain lines, the following format is used:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
. . . snip . . .
clints:x:500:500:Clint Savage:/home/clints:/bin/zsh
. . . snip . . .
```

Lab Conventions

Distribution Specific Information

This courseware is designed to support multiple Linux distributions. When there are differences between supported distributions, each version is labeled with the appropriate base strings:

R ⇒ Red Hat Enterprise Linux (RHEL)
S ⇒ SUSE Linux Enterprise Server (SLES)
U ⇒ Ubuntu

The specific supported version is appended to the base distribution strings, so for Red Hat Enterprise Linux version 6 the complete string is: R6.

Certain lab tasks are designed to be completed on only a sub-set of the supported Linux distributions. If the distribution you are using is not shown in the list of supported distributions for the lab task, then you should skip that task.

Certain lab steps are only to be performed on a sub-set of the supported Linux distributions. In this case, the step will start with a standardized string that indicates which distributions the step should be performed on. When completing lab tasks, skip any steps that do not list your chosen distribution. For example:

1) ^[R4] *This step should only be performed on RHEL4.*
Because of a bug in RHEL4's Japanese fonts...

Sometimes commands or command output is distribution specific. In these cases, the matching distribution string will be shown to the left of the command or output. For example:

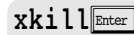
```
$ grep -i linux /etc/*-release | cut -d: -f2
[R6] Red Hat Enterprise Linux Server release 6.0 (Santiago)
[S11] SUSE Linux Enterprise Server 11 (i586)
```

Action Lists

Some lab steps consist of a list of conceptually related actions. A description of each action and its effect is shown to the right or under the action. Alternating actions are shaded to aid readability. For example, the following action list describes one possible way to launch and use **xkill** to kill a graphical application:



Open the "Run Application" dialog.



Launch **xkill**. The cursor should change, usually to a skull and crossbones.

Click on a window of the application to kill.

Indicate which process to kill by clicking on it. All of the application's windows should disappear.

Callouts

Occasionally lab steps will feature a shaded line that extends to a note in the right margin. This note, referred to as a "callout," is used to provide additional commentary. This commentary is never necessary to complete the lab successfully and could in theory be ignored. However, callouts do provide valuable information such as insight into why a particular command or option is being used, the meaning of less obvious command output, and tips or tricks such as alternate ways of accomplishing the task at hand.

```
[S10] $ sux - _____
      Password: password
      # xclock
```

- On SLES10, the **sux** command copies the MIT-MAGIC-COOKIE-1 so that graphical applications can be run after switching to another user account. The SLES10 **su** command did not do this.

Content

Application Management Landscape	2
Application Isolation	3
Container Resource Control & Security	5
Container Types	6
Container Ecosystem	7
Lab Tasks	8
1. Container Concepts LXC	9
2. Container Concepts Systemd	15

Chapter

1

CONTAINER TECHNOLOGY OVERVIEW

Applications Management Challenges

A modern server can run many application services (daemons) at the same time. One Linux system can serve as a mail server, database server, web server, file server, etc. However, this non-isolated approach isn't considered best practice for several reasons. Firstly, the more daemons running on the same server increases the attack surface area. With the daemons all running on the same operating system, a successful attack on one daemon can lead to compromise of the other daemons. Techniques such as `chroot(2)` and SELinux can help to prevent an attacker from moving laterally from one daemon to another.

Upgrades can be more difficult with multiple daemons on the same operating system. Suppose one web application requires a particular version of Python and then an update to a web application requires a newer version of Python that the first web application is not compatible with. This situation can crop up when running 3rd party web application frameworks that don't ship with a specific Enterprise Linux distribution.

Deploying an application often times requires tweaks and modifications to the operating system, installing application data and configuring the applications configuration files. Once an application is installed on one Linux system it is no trivial matter to migrate the application to another Linux system. Automation and orchestration frameworks such as Puppet, SALT, and Chef can help by abstracting and centrally managing application deployments.

Application Management Landscape

Approaches to server and application management have evolved

- Full featured server installs vs Just enough Operating System (JeOS)
- Many applications per server vs single application per server
- Physical servers vs virtual servers

Many different technologies address specific pain points

- Security, deployment, isolation, resource management

Linux Containers

- Goal: comprehensive, secure, lightweight application isolation and deployment
- Leverages modern features in kernel

Without configured resource limits, applications can consume more CPU, memory, or I/O than is desirable. If multiple applications are competing for a resource, the Linux kernel does its best to prevent monopolization and keep resource allocation as fair as possible. Sometimes resource allocation should purposely not be fair. For example, a database server should allow the database processes to monopolize the I/O.

Containers

Linux containers aim to solve many of the application deployment challenges in a comprehensive fashion. Linux containers bring together many modern Linux features to provide a very lightweight application isolation with image based application packing and delivery.

Docker provides a framework for creating, deploying, and managing containers. Competing frameworks exist.

Application Isolation

Pre-Container approaches to isolation

- One server per application (physical or virtual)
cons: expensive/heavyweight
- Change root (`chroot(2)`) — added to UNIX in 1979
changes apparent root directory for a process
cons: only isolates filesystem, limited security
- SELinux — enforces proper/allowed application behavior

Linux container features

- Good isolation of global system resources via namespaces(7)
filesystems, hostname, network, PID, IPC and UID/GIDs
- Single kernel = no virtualization overhead

Traditional Isolation Techniques

Some application developers have integrated support for the `chroot(2)` system call, so that when the application starts, it isolates itself to a specific directory on the filesystem and uses that directory as its apparent root directory. Notable examples include the DNS server **named**, the network time server **ntpd**, and the Postfix email server. Many Linux distributions have configured these applications to use `chroot` by default. One drawback of `chroot(2)` is that an attacker with root privileges can escape it.

In many ways, SELinux provides an updated `chroot(2)` mechanism that is much broader in scope. It focuses specifically on the security aspect of isolation, and enforces a mandatory "principal of least privilege". Applications can only see, modify and use the parts of the operating system and data that they need to do their job.

Using virtual machines, with one virtual machine per application, has been the most commonly deployed implementation of application isolation for many years. The drawback of virtualization is the overhead of emulating a PC architecture, including virtual PCI buses, PCI devices such as storage controllers and network cards, and then running another kernel and full operating system on top of the emulated PC architecture. Although a lot of work has been done to reduce the overhead of virtualization including special support within CPU chips for things like nested page tables and single root IO virtualization, the overhead of virtualization can never be eliminated.

Linux Namespaces

The Linux kernel contains global resources that are shared amongst all the processes running a system. Over time the Linux kernel has slowly gained the ability to segment more of these data structures. The abstraction of these resources is called a namespace. Processes within a namespace will appear to have their own isolated instance of the global resource; exactly as if an application was running in its own virtual machine but without the overhead of virtualization. Processes in different namespaces can use global resources, possibly even with the same id or name, and not conflict with each other on the container host. Currently, the available namespaces are:

IPC ⇒ System V interprocess communication resources such as POSIX message queues, shared memory segments and semaphores.

Network ⇒ A physical or virtual network interfaces, full networking stacks and firewall rules

Mount ⇒ Filesystem mount points

PID ⇒ Process IDs so that each container can have its own **init** (aka PID 1).

UTS ⇒ Hostname and NIS domain name

User ⇒ User and group IDs allow for a separate root user in each container

Although many technologies contribute to modern container implementations, namespaces are often considered the primary defining characteristic of a Linux container.

Examples of Namespace Use

By default, processes inherit the namespaces of their parent. Alternatively, when creating a child process through the `clone(2)` system call, a process can use the `CLONE_NEW*` flags to have one or more private namespaces created for the child process. A process can also use the `unshare(2)` system call to have a new namespace created and join it, or the `setns(2)` call to join an existing namespace.

The **unshare** command is a wrapper around the `unshare(2)` call, and the **nsenter** command wraps the `setns(2)` call. Using these programs, any program can be placed into alternate namespaces. Some other userspace commands also have an awareness of namespaces, a notable example is the **ip** command used to configure networking. The following examples show the use of these commands:

Private namespaces with unshare

Create a container running a shell in private IPC and PID namespaces:

```
# unshare -ip -f --mount-proc /bin/bash
# ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY      START   TIME COMMAND
root    1  0.0  0.1 116528 3312 pts/3    15:19   0:00 /bin/bash
root   29  0.0  0.0 123372 1376 pts/3    15:20   0:00 ps aux
# ipcs
----- Message Queues -----
key          msqid        owner         perms        used-bytes   messages

----- Shared Memory Segments -----
key          shmid        owner         perms        bytes       nattch     status

----- Semaphore Arrays -----
key          semid        owner         perms        nsems
```

Network namespaces with ip netns and nsenter

Create a new network namespace (`dmz1`), and run a process in that namespace that will return a process details listing when someone connects to the loopback on port 80:

```
# ip netns add dmz1
# ip netns exec dmz1 nc -l 127.0.0.1 80 <<<$(ps aux) &
[1] 22173
```

Launch a shell into the existing network namespace that PID 22173 is in:

```
# nsenter -t 22173 -n /bin/bash
# lsof -i
COMMAND  PID USER  FD  TYPE  DEVICE  NODE NAME
nc       22173 root   3u  IPv4 1644993  TCP localhost:http (LISTEN)
# nc 127.0.0.1 80
. . . output omitted . . .
```

User namespace with unshare

Run a shell with apparent UID of root that actually maps to an unprivileged user within the global namespace:

```
$ id -un
guru
$ unshare --user --map-root-user /bin/bash
# id -un
root
# cat /proc/$$/uid_map
          0          500          1
# touch file
# ls -l file
-rw-r--r--. 1 root root 0 Oct  1 13:08 file
# exit
$ ls -l file
-rw-r--r--. 1 guru guru 0 Oct  1 13:08 file
```

Container Resource Control & Security

System Resource Management via Linux Control Groups (cgroups)

- Can allocate and control CPU time, system memory, I/O bandwidth, etc

SELinux sVirt

- Isolates VMs or Containers from each other and host
- Enabled by default
- See `docker_selinux(8)`

POSIX Capabilities

- Docker runs as root but drops unneeded privileged capabilities by default

Files within the container are labeled by default as follows:

svirt_sandbox_file_t ⇒ default for all files within a container image.

docker_var_lib_t ⇒ default type for internal volumes

sysfs_t, proc_*_t, sysctl_*_t, etc. ⇒ virtual filesystems mounted within the container have specific types related to their security needs.

***** ⇒ files within external volumes keep their assigned types.

Containers and SELinux Multi-Category Security Separation

Files and processes for containers are further separated from one another via the assignment of MCS labels. By default a unique label is generated on container start, and assigned to that container's processes and files. The MCS label is formed from a pair of category numbers. For example, the complete label on a file within a container might be: `system_u:object_r:svirt_sandbox_file_t:s0:c241,c344` with a corresponding process for that container having a complete label of: `system_u:system_r:svirt_lxc_net_t:s0:c338,c578`.

Note that within the container it will appear as if SELinux is disabled even though it can be confirmed that SELinux is in enforcing mode on the host.

Control Groups

The Linux kernel 2.6.24 added a new feature called "control groups" (cgroups) that was primarily created by engineers at Google. The use of cgroups allows for labeling, control, isolation, accounting, prioritization, and resource limiting of groups of processes. With modern Linux systems, cgroups are managed with systemd service, scope, and slice units. Every child process inherits the cgroup of its parent and can't escape that cgroup unless it is privileged. View cgroups with **systemd-cgls** either in their entirety or for an individual service.

When running Docker containers, proportional memory, CPU, and I/O weights can be specified as parameters for the container. The constraints only take effect if there is contention. A container with constraints can use resources beyond the constraints if nothing else needs them. For more advanced cgroup functionality, a persistent cgroup can be defined in a systemd unit file and then referenced as the cgroup parent in a Docker container parameters.

Containers and SELinux Type Enforcement

By default, containers are assigned the `svirt_lxc_net_t` SELinux type. This type is allowed to read and execute from certain types (all of `/usr` and most of `/etc`), but denied access to most other types (such as content in `/var`, `/home`, `/root`, etc). The `svirt_lxc_net_t` type can only write to the `svirt_sandbox_file_t` and `docker_var_lib_t` types.

Container Types

One of the innovations of the Docker container system is packaging applications into self-contained images that are separate from the Docker host. This way applications can be deployed onto any Docker host or host that implement the Open Container Specification.

A container is based on an image that contains multiple layers. At the base is a platform image, all changes are made in the top-most writable layer.

Container Types

Image-based Containers — Most common production usage

- Application packaged with JeOS runtime
 - JeOS can be based on any Linux distribution
- Portability
- Multiple image layers

Host Containers

- Lightweight application sandboxes
- Same userspace as the host
 - Security updates applied to the host apply to the containers

Container Ecosystem

Open Container Initiative

- Industry Standard for Containers — based on Docker

Minimal Container Hosts

- CoreOS
- Red Hat's Project Atomic
- Canonical's Snappy Core

Orchestration

- Kubernetes
- Docker Compose

Open Container Initiative

The Open Container Initiative (OCI) is specification that isn't tied to any particular vendor, client, host or orchestration stack. The features of the specification are (From the OSI FAQ):

Composable ⇒ All tools for downloading, installing, and running containers should be well integrated, but independent and composable. Container formats and runtime should not be bound to clients, to higher level frameworks, etc.

Portable ⇒ The runtime standard should be usable across different hardware, operating systems, and cloud environments.

Secure ⇒ Isolation should be pluggable, and the cryptographic primitives for strong trust, image auditing and application identity should be solid

Decentralized ⇒ Discovery of container images should be simple and facilitate a federated namespace and distributed retrieval.

Open ⇒ The format and runtime should be well-specified and developed by a community. We want independent implementations of tools to be able to run the same container consistently.

Minimalist ⇒ The spec should aim to do a few things well, be minimal and stable, and enable innovation and experimentation above and around it

Backwards compatible ⇒ Given the broad adoption of the current Docker container format (500M container downloads to date), the new standard should strive be as backward compatible as possible with that format

Lab 1

Estimated Time: 35 minutes

Task 1: Container Concepts LXC

Page: 1-9 Time: 20 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Task 2: Container Concepts systemd

Page: 1-15 Time: 15 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Objectives

- ☞ Create and administer a simple container using virsh and the libvirt-lxc driver
- ☞ Use systemd tools to examine a running container.
- ☞ Explore the use of namespaces to isolate processes.

Requirements

- 🖥 (1 station) 🖥 (classroom server)


Relevance

Libvirt and systemd both have container support and provide a good environment to explore the basic concepts of containers without involving Docker.

Notices

- ☞ The libvirt-lxc backend is deprecated as of RHEL7.1
- ☞ The native systemd container capabilities are expanding rapidly and newer systemd versions have features very similar to Docker. See the following man pages for more details:
<http://www.freedesktop.org/software/systemd/man/machinectl.html>
<http://www.freedesktop.org/software/systemd/man/systemd-nspawn.html>

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Install the libvirt-lxc libvirt backend:

```
# yum install -y libvirt-daemon-lxc.x86_64  
. . . snip . . .  
Installed:  
  libvirt-daemon-lxc.x86_64 0:1.2.8-16.el7_1.2  
  
Dependency Installed:  
  libvirt-daemon-driver-lxc.x86_64 0:1.2.8-16.el7_1.2  
  
Complete!
```

Lab 1

Task 1

Container Concepts LXC

Estimated Time: 20 minutes

- 3) Configure the virsh command to use the lxc backend by default and verify it connects:

```
# virsh uri
qemu:///system

# export LIBVIRT_DEFAULT_URI=lxc:///

# virsh uri
error: failed to connect to the hypervisor
error: no valid connection
error: no connection driver available for lxc:///
error: Failed to reconnect to the hypervisor
# systemctl restart libvirtd
# virsh uri
lxc:///
```

- Determine the current default URI that libvirt is connecting to.
- Use the lxc backend by default to avoid typing `virsh -c lxc:///` each time.
- libvirtd doesn't see the additional backend drivers until it is restarted.

- 4) Create a simple XML file with the following content:

```
File: /root/test-lxc.xml
+ <domain type='lxc'>
+   <name>test-lxc</name>
+   <memory>25600</memory>
+   <os>
+     <type>exe</type>
+     <init>/bin/sh</init>
+   </os>
+   <devices>
+     <console type='pty' />
+   </devices>
+ </domain>
```

Note that the memory defined above will become a limit enforced by cgroups.

- 5) Define a new machine from the XML, and start the container:

```
# virsh define test-lxc.xml
Domain test-lxc defined from test-lxc.xml
# virsh list --all
```

```

Id      Name                                     State
-----
-       test-lxc                                shut off
# virsh start test-lxc
Domain test-lxc started
# virsh list
Id      Name                                     State
-----
6013    test-lxc                                running

```

- 6) Connect to the shell running in the container and explore the isolating effect of namespaces:

```

# virsh console test-lxc
Connected to domain test-lxc
Escape character is ^]
# pwd
/
# ls
bin  dev  home  lib  media  mnt  opt  root  sbin  sys  usr
boot etc  labfiles  lib64  misc  net  proc  run  srv  tmp  var
# ls /dev
console  full  ptmx  random  stdin  tty  urandom
fd       null  pts   stderr  stdout  ttyl  zero
# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root          1      0  0 16:31 pts/0        00:00:00 /bin/sh
root         23      1  0 16:33 pts/0        00:00:00 ps -ef
# hostname test-lxc
# hostname
test-lxc
# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 10.100.0.X  netmask 255.255.255.0  broadcast 10.100.0.255
    inet6 fe80::5054:ff:fe02:X  prefixlen 64  scopeid 0x20<link>



. . . snip . . .
# free

```

- mount namespaces can present a different filesystem(s). Default for libvirt-lxc backend is to share filesystems of host system.
- /dev is minimally populated with some base devices and those specifically defined for that domain (such as the console device).
- pid namespace isolates processes
- uts namespace isolates things like hostname changes.
- This will not be seen outside of the container.
- libvirt-lxc shares all network by default (but can isolate within a namespace if desired).
- Note the unusual negative used memory.

```
# head -n3 /proc/meminfo
```

```
MemTotal:      25600 kB
MemFree:       24788 kB
MemAvailable:  310476 kB
```

```
#  + 
```

An alternative way to view the memory limit is to run the following from the host (while the container is still running):

```
# grep memory_limit /sys/fs/cgroup/memory/machine.slice/machine-lxc\x2dtest\x2dlxc.scope/memory.stat
hierarchical_memory_limit 26214400
```

- /proc reports the container total memory as "Memtotal", and the host system total memory as "MemAvailable". This confuses many tools that do calculations with these numbers.
- This escape exits the virsh console. Typing exit would instead terminate the shell which would cause the container to stop.

Bonus: Using machinectl and further exploring namespaces

- 7) libvirtd registered the container with the systemd container services. Use the native systemd commands to examine the container:

```
# machinectl list
```

```
MACHINE                               CONTAINER SERVICE
lxc-test-lxc                          container libvirt-lxc
```

```
1 machines listed.
```

```
# machinectl status lxc-test-lxc
```

```
lxc-test-lxc(79daf6b4787d4dc89d4f031f73a3a2f4)
  Since: Wed 2015-06-03 16:40:19 MDT; 2min 31s ago
  Leader: 6225
  Service: libvirt-lxc; class container
  Unit: machine-lxc\x2dtest\x2dlxc.scope
       | -6226 /usr/libexec/libvirt_lxc --name test-lxc
  --console 22 --security=selinux --handshake 25 --background
       ` -6227 /bin/sh
```

Take note of the PID number for the Borne shell from your output:

Result: _____

- 8) Examine the namespaces in effect for the containerized shell, and your current shell:

```
# readlink /proc/PID_of_shell_from_step_7/ns/*
ipc:[4026532213]
mnt:[4026532211]
```

```

net:[4026531956]
pid:[4026532214]
uts:[4026532212]
# readlink /proc/$$/ns/*
ipc:[4026531839]
mnt:[4026531840]
net:[4026531956]
pid:[4026531836]
uts:[4026531838]

```

Note which namespaces are different (ipc, mnt, pid, and uts) and which are the same (net).

- 9) Discover how many namespaces are currently in use by processes and how many processes are using each namespace:

```

# ls -li /proc/*/ns/* | awk -F/ '{print $1, $5}' | sort | uniq -c
140 4026531836 pid
140 4026531838 uts
140 4026531839 ipc
136 4026531840 mnt
1 4026531856 mnt
140 4026531956 net
1 4026532114 net
1 4026532164 mnt
1 4026532165 mnt
1 4026532210 mnt
1 4026532211 mnt
1 4026532212 uts
1 4026532213 ipc
1 4026532214 pid

```

Take note of the inode number associated with the single process running in its own pid namespace (4026532214 in the sample output shown):

Result: _____

- 10) Track down the PID number of all processes using that pid namespace and use the ps command to view them (should be the containerized shell):

```

# find -L /proc/*/ns -inum inode_num_from_previous_step

```

```
/proc/6227/ns/pid
# ps e -p PID_from_previous_command_output
  PID TTY          STAT       TIME COMMAND
 6227 pts/0      Ss+        0:00 /bin/sh PATH=/bin:/sbin TERM=linux container=lxc-libvirt,
        container_uuid=79daf6b4-787d-4dc8-9d4f-031f7
```

Cleanup

11) Remove the test-lxc container:

```
# virsh destroy test-lxc
Domain test-lxc destroyed

# virsh undefine test-lxc
Domain test-lxc has been undefined

# virsh list --all
  Id      Name                                     State
-----
```

Note that if these cleanup steps are performed in a new shell, then **export LIBVIRT_DEFAULT_URI=lxc:///** before running the virsh commands.

Objectives

- 🔗 Create and administer a simple container using systemd-nspawn
- 🔗 Use systemd tools to manage a running container.

Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance

Notices

- 🔗 The native systemd container capabilities are expanding rapidly and newer systemd versions have features very similar to Docker. See the following man pages for more details:
<http://www.freedesktop.org/software/systemd/man/machinectl.html>
<http://www.freedesktop.org/software/systemd/man/systemd-nspawn.html>

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) The kernel audit facility is not currently supported when running systemd containers. Disable it by adding the following parameter to the existing line, and then rebooting:

File: /etc/default/grub
→ GRUB_CMDLINE_LINUX="video=800x600 crashkernel=auto rd.lvm.lv=vg0/swap ↵ rd.lvm.lv=vg0/root rhgb quiet audit=0 "

```
# grub2-mkconfig > /boot/grub2/grub.cfg  
Generating grub configuration file ...  
Found linux image: /boot/vmlinuz-3.10.0-229.1.2.el7.x86_64  
Found initrd image: /boot/initramfs-3.10.0-229.1.2.el7.x86_64.img  
Found linux image: /boot/vmlinuz-0-rescue-b5d83f3494c840b9b8d7f6d62a2b559d  
Found initrd image: /boot/initramfs-0-rescue-b5d83f3494c840b9b8d7f6d62a2b559d.img  
done  
# reboot
```

Lab 1

Task 2

Container Concepts Systemd

Estimated Time: 15 minutes

- 3) Install a minimal R7 system into the /srv/c1 directory using yum:

```
# mkdir /var/lib/machines/c1
# yum -y --releasever=7 --nogpg --installroot=/var/lib/machines/c1 --disablerepo='*' --enablerepo=base_
  install systemd passwd yum redhat-release-server vim-minimal
. . . snip . . .
tzdata.noarch 0:2015g-1.el7          ustr.x86_64 0:1.0.4-16.el7
util-linux.x86_64 0:2.23.2-26.el7    xz.x86_64 0:5.1.2-12alpha.el7
xz-libs.x86_64 0:5.1.2-12alpha.el7  yum-metadata-parser.x86_64 0:1.1.4-10.el7
zlib.x86_64 0:1.2.7-15.el7
```

Complete!

- 4) For simplicity, disable SELinux for the remainder of this exercise:

```
# setenforce 0
```

- 5) Launch a shell within a container rooted to the /var/lib/machines/c1 directory and set an initial root password:

```
# systemd-nspawn -D /var/lib/machines/c1
Spawning namespace container on /var/lib/machines/c1 (console is /dev/pts/2).
Init process in the container running as PID 2855.
-bash-4.2# passwd
Changing password for user root.
New password: makeitso 
BAD PASSWORD: The password fails the dictionary check - it is based on a dictionary word
Retype new password: makeitso 
passwd: all authentication tokens updated successfully.
-bash-4.2# exit
#
```

- 6) Boot the container and login:

```
# systemd-nspawn -D /var/lib/machines/c1/ -b
Spawning container c1 on /var/lib/machines/c1.
Press ^] three times within ls to kill container.
systemd 219 running in system mode. (+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSE
TUP +GCRYPT +GNUTLS +ACL +XZ -LZ4 -SECCOMP +BLKID +ELFUTILS +KMOD +IDN)
```




```
Detected virtualization systemd-nspawn.  
Detected architecture x86_64.
```

```
Welcome to Red Hat Enterprise Linux Server 7.2 (Maipo)!
```

```
Cannot add dependency job for unit display-manager.service, ignoring: Unit display-manager.service,  
failed to load: No such file or directory.  
[ OK ] Reached target Remote File Systems.  
[ OK ] Created slice Root Slice.  
. . . snip . . .  
[ OK ] Reached target Multi-User System.  
[ OK ] Reached target Graphical Interface.
```

```
Red Hat Enterprise Linux Server 7.2 (Maipo)  
Kernel 3.10.0-327.4.4.el7.x86_64 on an x86_64
```

```
c1 login: root  
Password: makeitso   
-bash-4.2#
```

Leave this terminal logged in and perform the next steps in another terminal.

- 7) From another terminal, examine the container using the `systemd machinectl` command:

```
[2]# machinectl list  
MACHINE                               CONTAINER SERVICE  
c1                                     container nspawn  
  
1 machines listed.  
[2]# machinectl status c1  
c1  
    Since: Fri 2016-03-11 22:34:46 MST; 1min 37s ago  
    Leader: 13535 (systemd)  
    Service: nspawn; class container  
    Root: /var/lib/machines/c1  
    Address: 10.100.0.1  
            192.168.122.1  
            fe80::52:ff:fe17:1  
    OS: Red Hat Enterprise Linux Server 7.2 (Maipo)  
    Unit: machine-c1.scope  
        |-13535 /usr/lib/systemd/systemd  
        |-user.slice
```

```

|  \-user-0.slice
|    \-session-31.scope
|      |-13587 login -- root
|      \-13658 -bash
|-system.slice
|   \-dbus.service
|     \-13581 /bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-a
|   \-systemd-logind.service
|     \-13580 /usr/lib/systemd/systemd-logind
|   \-systemd-journald.service
|     \-13547 /usr/lib/systemd/systemd-journald

```

Mar 11 22:34:46 station1.example.com systemd[1]: Starting Container c1.



8) Shut down the container and verify:

```

[2]# machinectl poweroff c1
[2]# systemctl status systemd-machined.service
systemd-machined.service - Virtual Machine and Container Registration Service
   Loaded: loaded (/usr/lib/systemd/system/systemd-machined.service; static; vendor preset: disabled)
   Active: inactive (dead)
     Docs: man:systemd-machined.service(8)
           http://www.freedesktop.org/wiki/Software/systemd/machined

```

```

Mar 11 22:25:11 station1.example.com systemd-machined[11763]: New machine lxc-test-lxc.
Mar 11 22:27:54 station1.example.com systemd-machined[11763]: Machine lxc-test-lxc terminated.
Mar 11 22:33:33 station1.example.com systemd[1]: Starting Virtual Machine and Container Registration .....
Mar 11 22:33:33 station1.example.com systemd[1]: Started Virtual Machine and Container Registration S...ce.
Mar 11 22:34:21 station1.example.com systemd-machined[13139]: New machine c1.
Mar 11 22:34:35 station1.example.com systemd-machined[13139]: Machine c1 terminated.
Mar 11 22:34:46 station1.example.com systemd-machined[13139]: New machine c1.
Mar 11 22:42:13 station1.example.com systemd-machined[13139]: Machine c1 terminated.

```

Bonus

9) If Internet access is available, or the Instructor can provide a copy of a Fedora

Cloud Base disk image such as:

https://dl.fedoraproject.org/pub/fedora/linux/releases/21/Cloud/Images/x86_64/Fedora-Cloud-Base-20141203-21.x86_64.raw.xz

10) After downloading the image to your system, extract it and then add an account:

```
# unxz Fedora-Cloud-Base-20141203-21.x86_64.raw.xz
# systemd-nspawn -i Fedora-Cloud-Base-20141203-21.x86_64.raw
Spawning container Fedora-Cloud-Base-20141203-21.x86_64.raw on /root/Fedora-Cloud-Base-20141203-21.x86_64.raw.
Press ^] three times within 1s to kill container.
[root@Fedora-Cloud-Base-20141203-21 ~]#
# useradd guru
# passwd guru
Changing password for user guru.
New password: work 
BAD PASSWORD: The password is shorter than 8 characters
Retype new password: work 
passwd: all authentication tokens updated successfully.
# exit
logout
Container Fedora-Cloud-Base-20141203-21.x86_64.raw exited successfully.
```

11) Boot the image and login with the added account:

```
# systemd-nspawn -i Fedora-Cloud-Base-20141203-21.x86_64.raw -b -M fedora21
Spawning container fedora21 on /root/Fedora-Cloud-Base-20141203-21.x86_64.raw.
Press ^] three times within 1s to kill container.
systemd 217 running in system mode. (+PAM +AUDIT +SELINUX +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +G
Detected virtualization 'systemd-nspawn'.
Detected architecture 'x86-64'.


Welcome to Fedora 21 (Twenty One)!

Set hostname to <localhost.localdomain>.





Running in a container, ignoring fstab device entry for /dev/disk/by-uuid/c3dbe26e-e200-496e-af8e-d3071afela29.
. . . snip . . .
[21614.879720] cloud-init[240]: ci-info: |    0    |    0.0.0.0    | 10.100.0.254 |    0.0.0.0    |    eth0    |    UG    |
[21614.879842] cloud-init[240]: ci-info: |    1    |   10.100.0.0   |    0.0.0.0   | 255.255.255.0 |    eth0    |    U    |
[21614.879976] cloud-init[240]: ci-info: |    2    | 192.168.122.0  |    0.0.0.0   | 255.255.255.0 |   virbr0   |    U    |
[21614.880273] cloud-init[240]: ci-info: +-----+-----+-----+-----+-----+-----+
Fedora release 21 (Twenty One)
Kernel 3.10.0-327.4.4.el7.x86_64 on an x86_64 (console)

station1 login: guru
```

```

Password: work 
Last login: Sat Mar 12 02:02:37 on pts/0
[guru@station1 ~]$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 02:08 ?        00:00:00 /usr/lib/systemd/systemd
root          18         1  0 02:08 ?        00:00:00 /usr/lib/systemd/systemd-journald
root          35         1  0 02:08 ?        00:00:00 /usr/lib/systemd/systemd-logind
. . . output omitted . . .
$ exit
logout

Fedora release 21 (Twenty One)
Kernel 3.10.0-327.4.4.el7.x86_64 on an x86_64 (console)

station1 login: +++
Container fedora21 terminated by signal KILL.

```

Cleanup

12) Remove the container filesystem and restore SELinux enforcing mode:

```

# rm -rf /var/lib/machines/c1
# rm Fedora-Cloud-Base-20141203-21.x86_64.raw
# setenforce 1

```

Content

Installing Docker	2
Docker Control Socket	4
Creating a New Container	5
Listing Containers	6
Viewing Container Operational Details	7
Running Commands in an Existing Container	8
Interacting with a Running Container	9
Stopping, Starting, and Removing Containers	10
Lab Tasks	11
1. Docker Basics	12
2. Install Docker via Docker Machine	22
3. Configure a docker container to start at boot.	28

Chapter

2

MANAGING CONTAINERS

Installing Docker

Use packages provided by distro

- RHEL, Fedora, CentOS, Oracle Linux, SLES, Debian, Ubuntu, etc.

Use sources from Docker

- `curl -sSL https://get.docker.com/ | sh`

Use docker-machine

- Drivers for: Amazon EC2, Azure, Digital Ocean, Exoscale, Google Cloud, Openstack, Rackspace, Softlayer, VirtualBox, VMware Cloud, VMWare Vsphere

Installing Docker

A growing list of Linux distributions include Docker packages in their official supported repositories. Generally, installing Docker is as simple as installing the provided package via the native package system (YUM, APT, etc.); for example:

```
# yum install -y docker-engine
Resolving Dependencies
--> Running transaction check
---> Package docker-engine.x86_64 0:1.10.3-1.el7.centos will be installed
. . . snip . . .
Installed:
  docker-engine.x86_64 0:1.10.3-1.el7.centos
```

```
Dependency Installed:
  docker-engine-selinux.noarch 0:1.10.3-1.el7.centos
```

Complete!

The main Docker repository often has a version newer than the various distro repositories. To install from the Docker repo directly, use the shell script served by the `get.docker.com` host; for example:

```
# curl -sSL https://get.docker.com/ | sh
+ sh -c 'sleep 3; yum -y -q install docker-engine'
warning: /var/cache/yum/x86_64/7Server/docker-main-repo/packages/docker-engine-selinux-1.10.3-1.el7.centos.noarch.rpm: Header V4 RSA
Public key for docker-engine-selinux-1.10.3-1.el7.centos.noarch.rpm is not installed
Importing GPG key 0x2C52609D:
  Userid      : "Docker Release Tool (releasedocker) <docker@docker.com>"
  Fingerprint: 5811 8e89 f3a9 1289 7c07 0adb f762 2157 2c52 609d
  From        : https://yum.dockerproject.org/gpg
```

```
. . . output omitted . . .
```

```
# systemctl start docker
```

```
# docker -v
```

```
Docker version 1.10.3, build 20f81dd
```

Cloud or VM Installs with docker-machine

The **docker-machine** command can be installed onto a host and then used to deploy and manage Docker daemon instances across a large number of platforms; for example:

```
# docker-machine create --driver virtualbox dev
```

```
Creating CA: /home/root/.docker/machine/certs/ca.pem
```

```
Creating client certificate: /home/root/.docker/machine/certs/cert.pem
```

```
Image cache does not exist, creating it at /home/root/.docker/machine/cache...
```

```
No default boot2docker iso found locally, downloading the latest release...
```

```
Downloading https://github.com/boot2docker/boot2docker/releases/download/v1.6.2/↵  
boot2docker.iso to /home/root/.docker/machine/cache/boot2docker.iso...
```

```
Creating VirtualBox VM...
```

```
Creating SSH key...
```

```
Starting VirtualBox VM...
```

```
Starting VM...
```

```
# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM
dev		virtualbox	Running	tcp://10.100.0.3:2376	

```
# eval "$(docker-machine env dev)"
```

```
# docker info
```

```
. . . output omitted . . .
```

Docker Client to Daemon via UNIX Socket

By default, the Docker daemon creates the `/var/run/docker.sock` control socket and sets it owned by `root:docker` mode `660` so that only the root user or group can interact with Docker. To allow a different group access, modify the socket options specified in the systemd socket definition. For example:

```
# cp /lib/systemd/system/docker.socket \
    /etc/systemd/system/docker.socket
```

File: /etc/systemd/system/docker.socket	
	[Socket]
	ListenStream=/var/run/docker.sock
	SocketMode=0660
	SocketUser=root
→	SocketGroup=dockercontainer

```
# groupadd container
# usermod -G container guru
# systemctl daemon-reload
# systemctl restart docker.socket
# ls -l /var/run/docker.sock
srw-rw----. 1 root container 0 Jul 31 11:04 /var/run/docker.sock
```

Using Docker via Network Socket

Instead of creating a local UNIX socket, the Docker daemon can bind to a specified TCP port allowing remote hosts access. Simply launch

Docker Control Socket

Docker client to daemon control socket

- no fine grained ACL, all or nothing!

Default `/var/run/docker.sock`

- `-G` daemon option to set group

-H daemon option to set alternate socket

- `-H tcp://IP:port`
- `$DOCKER_HOST=`
- Can secure with TLS certs

the daemon using the `-H IP:port` option and then connect using the same option from the client; for example:

File: /etc/systemd/system/docker.service	
→	ExecStart=/usr/bin/docker daemon -H 10.100.0.1:2375

```
[host1]# systemctl restart docker
```

```
[host2]$ export DOCKER_HOST="tcp://10.100.0.1:2375"
```

```
[host2]$ docker info
```

```
. . . snip . . .
Operating System: Red Hat Enterprise Linux Server 7.2 (Maipo)
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 740.2 MiB
Name: host1.example.com
ID: J3U4:TZW3:6GLX:3VHK:WUQW:U4W2:QKMU:D35P:UUEI:U3L3:H4NJ:WTMS
```

Security Warning

Access to the Docker socket must be protected as it is trivial to obtain root level access to the host system if arbitrary containers can be run. Treat access to the socket as the equivalent of giving root access. If listening on a network socket, firewall rules should be used to limit access to authorized individuals.

Creating a New Container

docker run [*OPTIONS*] *image* [*COMMAND*] [*ARG...*]

- image downloaded if not found locally
/var/lib/docker/
- default command and args generally defined by image

<http://docs.docker.com/reference/run/> and in the `docker-run(1)` manual page.

Most images include a default command and args that are used if one is not specified when the container is run. Depending on how the image is built, arguments appended to the end of the run command maybe interpreted either as *command arg ...*, or just *arg ...* (passed to an internally specified command).

docker run Examples

To create a detached container called `web1` from the Nginx image:

```
# docker run -d --name web1 nginx
c4d8b61d14d746339b38fd915a1e29a03f72f4e44aa76284070effe67035
```

To create a temporary container from the busybox image and run the `id` command as UID 500 and GID 1000:

```
# docker run -u 500:1000 --rm busybox id
uid=500 gid=1000(default)
```

To create an Ubuntu container using the latest image and open an interactive shell attached to the container:

```
# docker run -ti ubuntu
root@c7d0ea45fcc9:/#
```

Creating and Starting a Container

New containers are created with the **docker run** command. The only required parameter is the name of the image (filesystem) that the container uses. If the image is not already on the Docker host, then it is downloaded from the configured registry (Docker.io by default). The run command supports many options in part because it allows overriding any options contained within the image. Most of the options are used in more advanced cases and are discussed in later chapters. Examples of a few common options include:

- d|--detach** ⇒ Run container in background. Container ID is printed to terminal. Appropriate for containers that invoke long running daemons.
- ti|--tty --interactive** ⇒ Allocate a pseudo-TTY and keep STDIN open and attached. Most commonly used when container runs an interactive process like a shell.
- name=** ⇒ Assigns a container name that can be used in addition to the auto-assigned name and ID. Makes it easier to refer to the container in other Docker commands. As an alternative, consider storing the ID in a named variable: `name_var=$(docker run -d image)`
- rm** ⇒ Removes container after it exits. Use when launching temporary containers (containers are never removed by default).
- u|--user user:group** ⇒ User and group identity to use within container. If not specified, identity will usually be root (can be specified by image).

The full list of options is described at

Listing Containers

docker ps

- displays only running containers by default
- **-a** → list all containers
- **-l** → only last container
- **-q** → only print IDs

Listing Containers

The **docker ps** command will show the list of containers on the connect Docker host. By default, only running containers are shown. To see all containers (even stopped ones), use the **-a** option. To list just container IDs, use the **-q** option. To list just the last container, use the **-l** option. For example:

```
# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
f3dbedddb763   nginx:latest   "nginx -g 'daemon of    2 days ago    Up 3 minutes   0.0.0.0:32768->80/tcp    web2
c6bda13fe3e8   nginx:latest   "nginx -g 'daemon of    2 days ago    Up 17 minutes   80/tcp, 443/tcp        web1
# docker ps -q
f3dbedddb763
c6bda13fe3e8
```

To store the ID of the last created container in a variable:

```
# CONTAINER_ID=$(docker ps -ql)
```

To stop any running containers and then delete all containers:

```
# docker stop $(docker ps -q)
f3dbedddb763
c6bda13fe3e8
# docker rm $(docker ps -qa)
fdf2509e8380
daf4672ab635
8b0b0b66665e
f3dbedddb763
c6bda13fe3e8
```

Viewing Container Operational Details

`docker top` → process listing for container
`docker stats` → mem, CPU, and net I/O

Viewing Container Process Info

The `docker top` command will generate a `ps` like process listing for the specified container, and does not require any special commands to be installed within the container image itself. The following example output shows listing the processes in a simple webserver container:

```
# docker top web1
UID      PID      PPID     C   STIME   TTY   TIME      CMD
root     26380    13741    0   19:28   ?     00:00:00  nginx: master process nginx -g daemon off;
104      26442    26380    0   19:28   ?     00:00:00  nginx: worker process
```

Viewing Container Resource Usage

Through CGroups, Docker can both track and limit the system resources used by containers. the `docker stats` command will show the current CPU, memory, and network usage for the specified containers; for example:

```
# docker stats web1 web2
CONTAINER   CPU %       MEM USAGE/LIMIT   MEM %       NET I/O
web1        0.00%       7.23 MiB/741.4 MiB  0.98%       1.713 KiB/870 B
web2        0.00%       6.957 MiB/741.4 MiB  0.94%       648 B/648 B
```

Running Commands in an Existing Container

docker exec → Run command within a running container

- **-ti** → interactive session
- **-d** → command runs in background

Running Commands in an Existing Container

When a container is first launched, an initial process is specified either as part of the `docker run` command, or within the image as a `CMD` or `ENTRYPOINT`. This initial command may start a single process, or a collection of processes. Configuration, maintenance, or troubleshooting of processes within the container often requires running additional tasks within the container. The `docker exec` command will start a new process within an existing container. The specified command and args are passed to `/bin/sh -c`, so shell processing occurs. This means that variable expansion, command substitution, redirection and piping, etc. can all be used. Use of wildcards can be tricky due to the multiple levels of escaping.

The following examples show running some common troubleshooting commands within a container:

```
# docker exec web1 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=c4d8b61d14d7
container_uuid=c4d8b61d-14d7-4633-9b38-fd915a1e29a0
NGINX_VERSION=1.9.3-1~jessie
HOME=/root
# docker exec web1 ip addr show dev eth0 | grep inet
    inet 172.17.0.3/16 scope global eth0
    inet6 fe80::42:acff:fe11:3/64 scope link
# docker exec web1 /bin/sh -c 'ls -lZ /etc/pass*'
-rw-r--r--. 1 root root system_u:object_r:svirt_sandbox_file_t:s0:c378,c704 1251 Jul 18 00:20 /etc/passwd
-rw-----. 1 root root system_u:object_r:svirt_sandbox_file_t:s0:c378,c704 1238 Jul 18 00:20 /etc/passwd-
```

For an interactive shell session:

```
# docker exec -ti web1 /bin/bash
root@c4d8b61d14d7:/#
```

Interacting with a Running Container

docker attach → **attach a terminal to STD(IN|OUT|ERR) of container**

- `Ctrl+P` `Ctrl+Q` → Detach but continue running container.
- `Ctrl+C` → Send kill signals to processes and detach; container stops.

docker logs → **display accumulated logs from STD(OUT|ERR)**

Interacting with Terminal of Running Container

The `docker attach` command connects the local terminal to the input and output of the container. Multiple sessions can be attached to the same container allowing for Screen like multi-user functionality.

Viewing Container Logs

The preferred way to run daemons within a container is to run them in the foreground. When run in this way, most daemons will output vital log information to `STDOUT|ERR`. This output is collected by Docker and stored in the `/var/lib/docker/containers/container_id/container_id-json.log` along with a date/time stamp. The `docker logs` command will display the full output of the corresponding container's log file to the screen, and supports the following options:

-f|--follow ⇒ continue reading log output for new data (like `tail -f`)
-t|--timestamps ⇒ Show date/time stamps with log output
--tail num ⇒ output the specified number of line from end of log

```
# docker logs web1
172.17.42.1 - - [28/Jul/2015:19:11:36 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
172.17.42.1 - - [28/Jul/2015:19:14:41 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
172.17.42.1 - - [28/Jul/2015:19:14:54 +0000] "POST / HTTP/1.1" 405 172 "-" "curl/7.29.0" "-"
```

Stopping, Starting, and Removing Containers

docker stop → **Stop container; SIGTERM then SIGKILL**

docker kill → **Kill container; SIGKILL**

- **-s** | **--signal** send alternate signal

docker start → **Start a previously defined container**

- **-a** → attach STD(OUT|ERR)
- **-i** → attach STDIN; interactive use

docker rm → **Remove container**

- **-f** | **--force** → remove even if running
- **-v** | **--volumes** → remove associated volumes

Stopping Containers

The `docker stop` command will send SIGTERM to the processes running in a container. After an optionally specified time (default 10 seconds), it will send SIGKILL to any remaining processes. When the initial process of a container (usually pid 1) terminates, the container stops. For example, to stop the `web1` and `web2` containers (allowing the processes up to 30 seconds to respond to the SIGTERM signal):

```
$ docker stop -t 30 web1 web2
```

The following would suspend the processes in the `db` container by sending SIGSTOP:

```
$ docker kill -s SIGSTOP db
```

Starting Containers

To start a container that was previously created and either terminated naturally or was manually stopped use the `docker start` command. For processes that are not listening for input, but have output, use the `-a` option to attach to the STD(OUT|ERR). For processes that accept interactive input, use the `-i` option to attach the local terminal STDIN to the container STDIN.

Removing Containers

By default, containers are persistent and will still be defined when they are manually stopped or terminate naturally. This keeps a record of the container, and allows restarting the container if needed. If a container is truly no longer needed, then it should be deleted to recover the space associated with storing its images, logs, and volumes. Use the `docker rm` command to remove a container. If the container is still running, then the `-f` option must be used to remove the container. If the `-v` option is used then any volumes associated with the container will also be removed assuming they are not referenced by other containers.

The following example shows using command nesting to get a list of all containers and remove them (and their associated volumes), even if they are running:

```
$ docker rm -fv $(docker ps -qa)
```

Delete all containers that are not currently running:

```
[bash]$ docker rm $(comm -3 <(docker ps -qa|sort) > <(docker ps -q|sort))
```

Lab 2

Estimated Time: 60 minutes

Task 1: Docker Basics

Page: 2-12 Time: 30 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Task 2: Install Docker via Docker Machine

Page: 2-22 Time: 15 minutes

Requirements: 🖥️🖥️ (2 stations) 🖨️ (classroom server)

Task 3: Configure a docker container to start at boot.

Page: 2-28 Time: 15 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Lab 2

Task 1

Docker Basics

Estimated Time: 30 minutes

Objectives

- 🔧 Install the Docker client and server
- 🔧 Configure Docker to use a local registry
- 🔧 Configure Docker to use devicemapper for storage
- 🔧 Download an image and launch simple containers
- 🔧 Manage containers and images
- 🔧 Examine the changes made to a host system by Docker when running containers


Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance

Install Docker Manually

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Install the Docker engine on your first assigned lab host:

```
# yum install -y docker-engine  
. . . snip . . .  
Installed:  
  docker-engine.x86_64 0:1.10.3-1.el7.centos  
  
Dependency Installed:  
  docker-engine-selinux.noarch 0:1.10.3-1.el7.centos  
  
Complete!
```

- 3) Examine the systemd unit file used by Docker to discover how it is configured and run:

```
# cat /usr/lib/systemd/system/docker.service  
[Unit]  
Description=Docker Application Container Engine  
Documentation=https://docs.docker.com
```



```

After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
ExecStart=/usr/bin/docker daemon -H fd://
MountFlags=slave
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target

```

- 4) Create a copy of the unit file and edit it to reference an external environment file:

```
# cp /usr/lib/systemd/system/docker.service /etc/systemd/system/docker.service
```

File: /etc/systemd/system/docker.service	
	[Service]
	Type=notify
+	EnvironmentFile=/etc/sysconfig/docker
→	ExecStart=/usr/bin/docker daemon -H fd:// \$OPTIONS

- 5) Edit the main Docker config adding these lines so that it uses the registry on the classroom server:

File: /etc/sysconfig/docker	
+	OPTIONS='-H fd:// --registry-mirror http://server1:5000 --insecure-registry 10.100.0.0/24'

The insecure option is to avoid having to install the TLS certificate for now.

- 6) Start the Docker service and configure it to start on boot:

```

# systemctl enable docker
ln -s '/usr/lib/systemd/system/docker.service' '/etc/systemd/system/multi-user.target.wants/docker.service'
# systemctl start docker
# systemctl status docker
docker.service - Docker Application Container Engine

```

```

Loaded: loaded (/etc/lib/systemd/system/docker.service; enabled)
Active: active (running) since Thu 2015-03-30 16:59:42 MDT; 18s ago
  Docs: http://docs.docker.com
Main PID: 4664 (docker)
  CGroup: /system.slice/docker.service
          └─4664 11354 /usr/bin/docker daemon -H fd:// --registry-mirror http://server1:5000 --insecure-registry 10.100.

```

If you see errors reported, then recheck your configs and restart.

7) Examine the detailed version info for this Docker host instance:

```

# docker version
Client:
 Version:      1.10.3
 API version:  1.22
 Go version:   gol.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:39:25 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.3
 API version:  1.22
 Go version:   gol.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 15:39:25 2016
 OS/Arch:      linux/amd64

```

8) Download a simple Docker image from the classroom server:

```

# docker pull server1:5000/busybox
Using default tag: latest
Pulling repository server1:5000/busybox
47bcc53f74dc: Pull complete
1834950e52ce: Pull complete
Status: Downloaded newer image for server1:5000/busybox:latest
# docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
server1:5000/busybox	latest	298a170cb181	12 days ago	1.113 MB

- 9) Tag the image with an alternate name that is easier to use:

```
# docker tag server1:5000/busybox busybox
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
busybox	latest	298a170cb181	12 days ago	1.113 MB
server1:5000/busybox	latest	298a170cb181	12 days ago	1.113 MB

- 10) Create a new container from the image and run the echo command within the container:

```
# docker run busybox /bin/echo "Hello"
Hello
# docker ps
. . . output omitted . . .
```

- STDOUT from processes in the container are sent to the terminal by default.
- No output is shown because the container terminated immediately after running the command
- -a shows recent containers (even stopped)

```
# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
28dcc6a4d179	busybox:latest	"/bin/echo Hello"	17 seconds ago	Exited (0)	16 seconds ago	stoic_heisenberg

Container names are automatically generated if one is not explicitly assigned when the container is launched. Names can be used instead of container IDs when referring to the container later.

- 11) Start a new container (based on the same image) that runs a long running process and also detaches from the terminal (runs in background):

```
# docker run -d busybox /bin/sh -c 'while true; do echo Hello; sleep 2; done'
c732093ba0812dfa8e0c0c6401f62c7dc85633d9389aa489bdc513e0240d75e
```

- This is the full container ID.

```
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c732093ba081	busybox:latest	"/bin/sh -c 'while t	9 seconds ago	Up 9 seconds		

- 12) Docker captures the STD(OUT|ERR) of processes in disconnected containers. View the output that has accumulated for the disconnected container (use the container ID obtained from the previous docker ps output):

```
# docker logs c732093ba081
Hello
Hello
```

```
Hello
. . . snip . . .
```

- 13) A local terminal can be reattached to interact with STD(IN|OUT|ERR) if needed.
Attach to the still running container:

```
# docker attach c7
Hello
Hello
```

- Remember that container IDs can be abbreviated to as little as one letter as long as they are still unique.

- 14) Detach from the container again by running the following in another terminal window as the root user:

```
# pkill -f -9 'docker attach'
```

Return to the original terminal and verify that the process was killed:

```
. . . snip . . .
Hello
Hello
Killed
#
```

- 15) Assign a name to the container and then stop it:

```
# docker rename c732093ba081 test
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
c732093ba081	busybox:latest	"/bin/sh -c 'while t	11 minutes ago	Up 11 minutes	

```
# docker stop test
test
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
--------------	-------	---------	---------	--------	-------

- 16) Identity the container ID of the first container (not test) and remove it:

```
# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	
c732093ba081	busybox:latest	"/bin/sh -c 'while t	13 minutes ago	Exited (137) 12 seconds ago	. . . snip
28dcc6a4d179	busybox:latest	"/bin/echo Hello"	15 minutes ago	Exited (0) 15 minutes ago	. . . snip

```
# docker rm 28dcc6a4d179
28dcc6a4d179
```

- 17) Start the test container back up:

```
# docker start test
test
```

- 18) Execute a shell, running inside of the container and attach that shell to the terminal:

```
# hostname
stationX.example.com
# docker exec -ti test /bin/sh
/ # hostname
c732093ba081
```

- 19) Explore the environment within the container to see the effect of the namespaces:

```
/ # ps -ef
PID   USER     COMMAND
   1 root    /bin/sh -c while true; do echo Hello; sleep 2; done
   87 root    /bin/sh
   92 root    sleep 2
   93 root    ps -ef
/ # ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
/ # exit
```

- 20) Examine the virtual Ethernet interface and bridge created automatically by Docker to network the container:

```
# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
```

```

    link/ether 02:52:00:13:01:03 brd ff:ff:ff:ff:ff:ff
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT
    link/ether 52:54:00:3e:fa:22 brd ff:ff:ff:ff:ff:ff
4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 state DOWN mode DEFAULT qlen 500
    link/ether 52:54:00:3e:fa:22 brd ff:ff:ff:ff:ff:ff
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT
    link/ether 02:42:9f:74:64:db brd ff:ff:ff:ff:ff:ff
15: veth5d55869@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT
    link/ether b6:5f:9a:b6:2c:9c brd ff:ff:ff:ff:ff:ff link-netnsid 0
# brctl show docker0
    bridge name      bridge id                STP enabled    interfaces
    docker0          8000.56847afe9799        no              veth5d55869

```

- 21) Examine (from within the container) the writable filesystem layer and associated device created for the container:

```

# docker exec test df -Ph | grep docker
/dev/mapper/docker-253:3-6314090-11dcable9bee...4bdb0d  xfs  10.0G 33.8M 10.0G  0% /

```

There are many storage related options that can be passed to the Docker daemon to control the size and type of the associated backing store.

- 22) Examine (from the host system) the device mapper device used by Docker:

```

# dmsetup table | grep docker.*pool
docker-253:3-6314090-pool: 0 209715200 thin-pool 7:1 7:0 128 32768 1 skip_block_zeroing
# grep loop /proc/partitions
    7          0 104857600 loop0
    7          1   2097152 loop1
# losetup
NAME          SIZELIMIT OFFSET AUTOCLEAR RO BACK-FILE
/dev/loop0    0          0          1 0 /var/lib/docker/devicemapper/devicemapper/data
/dev/loop1    0          0          1 0 /var/lib/docker/devicemapper/devicemapper/metadata
# ls -lsh /var/lib/docker/devicemapper/devicemapper/*
28M -rw-----. 1 root root 100G Mar 30 17:52 /var/lib/docker/devicemapper/devicemapper/data
624K -rw-----. 1 root root 2.0G Mar 30 17:50 /var/lib/docker/devicemapper/devicemapper/metadata

```

Using thinly provisioned device mapper devices built on loopback devices over sparse files is the default, but not appropriate for real world deployments due to inefficiencies.

Configuring devicemapper Storage

23) Stop Docker and remove the associated directories:

```
# systemctl stop docker
# rm -rf /var/lib/docker/*
```

24) Resize the /var filesystem so that it has enough space to hold later Docker images:

```
# df -h /var
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/vg0-var      2.0G  157M  1.9G   8% /var
# fsadm -l resize /dev/vg0/var 5G
Size of logical volume vg0/var changed from 2.00 GiB (512 extents) to 5.00 GiB (1280 extents).
Logical volume var successfully resized.
meta-data=/dev/mapper/vg0-var  isize=256    agcount=4, agsize=131072 blks
=                               sectsz=512   attr=2, projid32bit=1
=                               crc=0      finobt=0
data      =                               bsize=4096 blocks=524288, imaxpct=25
=                               sunit=0    swidth=0 blks
naming    =version 2                   bsize=4096 ascii-ci=0 ftype=0
log       =internal                   bsize=4096 blocks=2560, version=2
=                               sectsz=512  sunit=0 blks, lazy-count=1
realtime  =none                       extsz=4096  blocks=0, rtextents=0
data blocks changed from 524288 to 1310720
# df -h /var
Filesystem                Size  Used Avail Use% Mounted on
/dev/mapper/vg0-var      5.0G  233M  4.8G   5% /var
```

25) Create an LVM thin pool within the existing volume group:

```
# lvcreate -T -L 15G -n dockerpool vg0
Logical volume "dockerpool" created.
```

26) Modify the daemon options to use the thinpool for storage (and to make it easier to read):

File: /etc/sysconfig/docker

```
- OPTIONS='-H fd:// --registry-mirror http://server1:5000 --insecure-registry 10.100.0.0/24'
+ OPTIONS='-H fd:// \
+   --registry-mirror http://server1:5000 \
+   --insecure-registry 10.100.0.0/24 \
+   --storage-driver=devicemapper \
+   --storage-opt=dm.thinpooldev=/dev/mapper/vg0-dockerpool'
```

27) Start the daemon back up and create a simple container:

```
# systemctl start docker
# docker run -dti --name test server1:5000/busybox
Unable to find image 'server1:5000/busybox:latest' locally
latest: Pulling from busybox
385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:6757d4b17cd75742fc3b1fc1a8d02b45b637f2ac913ee9669f5c2aed0c9b26ba
Status: Downloaded newer image for server1:5000/busybox:latest
f527a582f50dc0081d802d94a223654ecda93d6995ce7829af68becf44a9cf4c
```

28) Examine the devicemapper devices for both the thinpool and the thinly provisioned storage space (backed by the pool) for the launched container:

```
# dmsetup ls | grep dockerpool
vg0-dockerpool_tmeta (253:4)
vg0-dockerpool (253:6)
vg0-dockerpool_tdata (253:5)
# dmsetup ls | grep ^docker
docker-253:3-8388736-90230ce9e3cfc1dfc68301896ec2367324a69264eab188679eded16015b9ea07 (253:7)
```

When Docker is using the devicemapper storage backend, it uses an LVM thin pool (which is built on top of kernel device mapper thin pools. Visible in the device mapper table, there will be a block device for each container that is running that has an ID matching the container name (in this case 838876...), and a pool device backed by data and meta-data pool devices.

For more details regarding thin-pool devices, see the associated kernel and LVM documentation:

<https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt>,
man lvmthin(7).

- 29) Within the LVM thin pool the individual container filesystems are composed. By default, each container gets an apparent 10G (with space only really allocated when needed allowing for over subscription within the thin-pool LV):

```
# docker exec test df -Ph | grep docker
/dev/mapper/docker-253:3-8388736-90230ce9e3cfc1dfc68301896ec2367324a69264eab188679eded16015b9ea07      10.0G      33.8M
```

- 30) Remove the running container:

```
# docker rm test
Error response from daemon: Conflict, You cannot remove a running container. Stop the container before,
attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
# docker rm -f test
test
```

- 31) Notice that the image persists even if all containers associated with it have been deleted:

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
busybox              latest             298a170cb181       12 days ago        1.113 MB
server1:5000/busybox latest             298a170cb181       12 days ago        1.113 MB
```

- 32) Try to remove the image:

```
# docker rmi busybox
Untagged: busybox:latest
```

Since other tags reference the same image layers, only the tag is removed.

- 33) Remove the image:

```
# docker rmi server1:5000/busybox
Untagged: server1:5000/busybox:latest
Deleted: sha256:298a170cb181629fa5c0c996203c94f6c4cde84c7b7d159753b8109df5494392
Deleted: sha256:f6075681a244e9df4ab126bce921292673c9f37f71b20f6be1dd3bb99b4fdd72
Deleted: sha256:1834950e52ce4d5a88a1bbd131c537f4d0e56d10ff0dd69e66be3b7dfa9df7e6
```

Since no other tags reference those image layers, and no containers are using them, the image layers are deleted.

Objectives

- 🔗 Install the Docker Engine to your second lab node.

Requirements

- 🖥️ (2 stations) 🖥️ (classroom server)

Relevance

When hosting a larger number of containers, it is helpful to have an automated method of installing, configuring, and using the Docker engine on a host. Docker Machine supports many backend drivers allowing you to easily provision Docker onto a wide variety of Virtual, and Cloud hosting providers.

Notices

- 🔗 In this lab task, the first lab system assigned to you is referred to as node1 or stationX. The second lab system assigned to you is referred to as node2 or stationY.

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) From your first assigned lab host (node1), create an SSH key and install it as trusted onto your second host:


```
[node1]# ssh-keygen  
Generating public/private rsa key pair.  
Enter file in which to save the key (/root/.ssh/id_rsa):   
Enter passphrase (empty for no passphrase):   
Enter same passphrase again:   
Your identification has been saved in /root/.ssh/id_rsa.  
Your public key has been saved in /root/.ssh/id_rsa.pub.  
The key fingerprint is:  
e1:9b:cf:9c:58:8f:67:17:c3:b1:35:fb:01:94:75:3a root@stationX.example.com  
. . . snip . . .  
[node1]# ssh-copy-id stationY  
The authenticity of host 'stationY (10.100.0.Y)' can't be established.  
ECDSA key fingerprint is 11:9d:dd:80:7f:69:96:67:3d:e2:53:54:be:96:5b:39.  
Are you sure you want to continue connecting (yes/no)? yes
```

Lab 2

Task 2

Install Docker via Docker Machine

Estimated Time: 15 minutes

```
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
root@stationY's password: makeitso 
```

```
Number of key(s) added: 1
```

```
. . . snip . . .
```

```
[node1]# ssh stationY hostname -i
```

```
10.100.0.Y
```

- Verify the newly installed key works by running a command on node2

3) Install the docker-machine binary onto your system:

```
[node1]# curl -o /usr/bin/docker-machine http://server1/docker-machine
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100 34.5M	100 34.5M	0 0	385M 0	--:--:--	--:--:--	--:--:--	387M

```
[node1]# chmod a+x /usr/bin/docker-machine
```

4) Docker Machine invokes the install script from the Docker website by default. Create a simple service that simply installs Docker from the local yum repo instead and serve that on a network accessible URL:

```
[node1]# echo "yum install -y docker-engine" | nc -l 10.100.0.X 8000 &
[1] 15679
```

5) Use Docker Machine and the generic driver (SSH) to install and configure Docker onto your second lab host (node2). Be sure to replace X with your first node address, and Y with your second node address:

```
[node1]# docker-machine create -d generic --generic-ip-address 10.100.0.Y --engine-install-url  
http://stationX:8000/ stationY.example.com
```

```
Running pre-create checks...
```

```
Creating machine...
```

```
(stationY.example.com) No SSH key specified. Connecting to this machine now and in the future will require the  
ssh agent to contain the appropriate key.
```

```
Waiting for machine to be running, this may take a few minutes...
```

```
Detecting operating system of created instance...
```

```
Waiting for SSH to be available...
```

```
Detecting the provisioner...
```

```
Provisioning with redhat...
```

```
GET / HTTP/1.1
```

```
User-Agent: curl/7.29.0
Host: stationX:8000
Accept: */*
```

```
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine-
  env stationY.example.com
[1]+  Done                  echo "yum install -y docker-engine" | nc -l 10.100.0.X 8000
```

• This may take a minute due to entropy depletion.... Be patient.

6) Verify that the node2 system is now listed:

```
# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
stationY.example.com	-	generic	Running	tcp://10.100.0.Y:2376		v1.10.3	

7) Examine the configuration that was created for the Docker instance installed on your second node:

```
[node1]# docker-machine config stationY.example.com
--tlsverify
--tlscacert="/root/.docker/machine/certs/ca.pem"
--tlscert="/root/.docker/machine/certs/cert.pem"
--tlskey="/root/.docker/machine/certs/key.pem"
-H=tcp://10.100.0.Y:2376
[node1]# ssh 10.100.0.Y 'grep ExecStart /etc/systemd/system/docker.service'
ExecStart=/usr/bin/docker daemon -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver devicemapper-
  --tlsverify --tlscacert /etc/docker/ca.pem --tlscert /etc/docker/server.pem --tlskey /etc/docker/server-key.pem-
  --label provider=generic
```

Note that the second node Docker engine was configured to listen on the network and to require TLS for communication.

8) Import the necessary environment variables to point the docker command to your second node:

```
# eval $(docker-machine env stationY.example.com)
```

```
# env | grep DOCKER
```

```
DOCKER_HOST=tcp://10.100.0.Y:2376
```

```
DOCKER_MACHINE_NAME=stationY.example.com
```

```
DOCKER_TLS_VERIFY=1
```

```
DOCKER_CERT_PATH=/root/.docker/machine/machines/stationY.example.com
```

• Examine the variables that were set

- 9) Verify that the docker command is now accessing the second node:

```
# docker-machine active
```

```
stationY.example.com
```

```
# docker info 2>/dev/null | grep ^Name
```

```
Name: stationY.example.com
```

- 10) Attempt to launch a container on node2 to verify that it is configured correctly:

```
# docker run --name webtest -d server1:5000/nginx
```

```
Unable to find image 'server1:5000/nginx:latest' locally
```

```
docker: Error response from daemon: unable to ping registry endpoint https://server1:5000/v0/
```

```
v2 ping attempt failed with error: Get https://server1:5000/v2/: EOF
```

```
v1 ping attempt failed with error: Get https://server1:5000/v1/_ping: EOF.
```

```
See 'docker run --help'.
```

Fails because the node is not Internet connected, and not configured for any other image registries.

- 11) Configure node2 to use the classroom server as a registry:

File: /root/.docker/machine/machines/stationY.example.com/config.json

```
... snip ...
```

```
"HostOptions": {
```

```
  "Driver": "",
```

```
  "Memory": 0,
```

```
  "Disk": 0,
```

```
  "EngineOptions": {
```

```
... snip ...
```

```
→   "InsecureRegistry": ["10.100.0.0/24"],
```

```
... snip ...
```

```
→   "RegistryMirror": ["http://server1:5000"],
```

12) Re-deploy with the new config and restart:

```
[node1]# docker-machine provision stationY.example.com
Waiting for SSH to be available...
Detecting the provisioner...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
[node1]# docker-machine ssh stationY.example.com
[node2]# systemctl daemon-reload
[node2]# systemctl restart docker
[node2]# exit
```

For most Docker Machine drivers, running `docker-machine restart node_name` would be the correct way to do this. The generic SSH driver currently does not support this method, thus the manual restart of the docker service.

13) Test your node2 configuration by launching a simple web server container and connecting to it from node1:

```
[node1]# docker run --name webtest -dp 80:80 server1:5000/nginx
Unable to find image 'server1:5000/nginx:latest' locally
Pulling repository server1:5000/nginx
af4b3d7d5401: Pull complete
917c0fc99b35: Pull complete
. . . snip . . .
29da7f3ca6ddf9375020ef893c033514798da3fa3fc6de05d57174679ee59e7a
[node1]# curl stationY
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
. . . snip . . .
```

Cleanup

14) Remove the webtest container and related image:

```
[node1]# docker rm -f webtest
```

```
webtest
[nodel]# docker rmi server1:5000/nginx
Untagged: server1:5000/nginx:latest
Deleted: sha256:50836e9a7005ec9036a2f846dcd4561fd21f708b02cda80f0b6d508dfae6515
Deleted: sha256:2bd2cdbf8a04fa01956fdf075059c4b2bae0203787cb8a5eaa7dd084a23f0724
. . . snip . . .
```

- 15) Unset the variables exported by the earlier eval command so that the docker command points to the local system again:

```
# unset -v $(env | awk -F= '/DOCKER/ {print $1}')
# docker info 2>/dev/null | grep ^Name
Name: stationX.example.com
```

Objectives

- ☞ Configure a docker container to restart if it fails
- ☞ Configure a docker container to auto-start at boot

Requirements

- 🖥 (1 station) 🖥 (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Run a simple ubuntu container:

```
# docker run --name c1 -d server1:5000/ubuntu /bin/sleep 86400  
Unable to find image 'ubuntu:latest' locally  
Trying to pull repository server1:5000/ubuntu ...  
09b55647f2b8: Download complete  
6071b4945dcf: Download complete  
. . . snip . . .  
2556113a7f655b5e52db815e831f2536eald0974bfcb15aa4718b8adec4f57a1
```

- 3) Confirm the ubuntu container is running:

```
# docker ps  
CONTAINER ID          IMAGE          COMMAND          CREATED          STATUS          PORTS          NAMES  
2556113a7f65          ubuntu        "bin/sleep 86400" 53 seconds ago  Up 51 seconds  c1
```

- 4) Kill the container via the docker command to see if it automatically restarts:

```
# docker kill c1  
c1  
# docker ps -q
```

Containers that are manually stopped or killed via docker commands are never auto-restarted.

Lab 2 Task 3

Configure a docker container to start at boot.

Estimated Time: 15 minutes

- 5) Restart the container and then kill the primary process within the container to see if the container is automatically restarted:

```
# docker start c1
c1
# docker top c1 -o pid
PID
10089
# kill -9 10089
# docker ps -q
```

The container did not get auto-restarted. However, this is governed by the policy assigned to the container which is explored in the next steps.

- 6) Inspect the restart policy for the container and start a new container with a different policy:

```
# docker inspect -f '{{.HostConfig.RestartPolicy}}' c1
{no 0}
# docker rm c1
c1
# docker run --name c1 --restart=always -d server1:5000/ubuntu /bin/sleep 86400
376156d2f073594c78b7202b5f220b2ecd454c336bf22b254565d1b1121a4f31
# docker inspect -f '{{.HostConfig.RestartPolicy}}' c1
{always 0}
```

- 7) Kill the primary process within the container to see if the container is automatically restarted:

```
# docker top c1 -o pid
PID
10395
# kill -9 10395
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
376156d2f073	server1:5000/ubuntu	"/bin/sleep 86400"	About a minute ago	Up 5 seconds	

```
# docker top c1 -o pid
PID
10422
```

• Note new PID because it is a new container.

The container did get auto-restarted because of the assigned policy.

8) Reboot the system:

```
# systemctl reboot
```

9) Log back into the host.

10) See if the container is running after a reboot:

```
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
376156d2f073        server1:5000/ubuntu  "/bin/sleep 86400"  7 minutes ago       Up About a minute
```

The container restarted at boot because of the assigned restart policy:

Cleanup

11) Remove the container:

```
# docker rm -f c1
c1
```

Bonus: Auto-start Container on Boot via Systemd

12) Create a new systemd unit file to control a simple Ubuntu container:

```
File: /etc/systemd/system/ubuntu-container.service
+ [Unit]
+ Description=Simple Ubuntu container
+ Requires=docker.service
+ After=docker.service
+
+ [Service]
+ ExecStartPre=/bin/docker run --name c1 -dti server1:5000/ubuntu
+ ExecStart=/bin/docker start -a c1
+ ExecStop=/bin/docker stop c1
+
+ [Install]
+ WantedBy=multi-user.target
```

13) Notice the unit is disabled:

```
# systemctl status ubuntu-container
ubuntu-sleep.service - An example which starts an Ubuntu container at boot.
   Loaded: loaded (/etc/systemd/system/ubuntu-container.service; disabled)
   Active: inactive (dead)
```

14) Enable and start the unit:

```
# systemctl enable ubuntu-container
ln -s '/etc/systemd/system/ubuntu-container.service' '/etc/systemd/system/multi-user.target.wants/ubuntu-container.service'
# systemctl start ubuntu-container
```

If any errors are reported, check the unit file created above for typos before proceeding.

15) Confirm the container started:

```
# systemctl is-active ubuntu-container.service
active
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ecd8cfdcf4e	server1:5000/ubuntu	"/bin/bash"	10 seconds ago	Up 9 seconds		

16) Reboot the system:

```
# systemctl reboot
```

17) Confirm the unit started at boot:

```
# systemctl status ubuntu-container
o ubuntu-container.service - Simple Ubuntu container
   Loaded: loaded (/etc/systemd/system/ubuntu-container.service; enabled; vendor preset: disabled)
   Active: active (running) since Tue 2016-04-05 15:00:16 MDT; 2min 16s ago
   Process: 1484 ExecStartPre=/bin/docker run --name c1 -dti server1:5000/ubuntu (code=exited, status=125)
   Main PID: 1516 (docker)
   CGroup: /system.slice/ubuntu-container.service
           └─1516 /bin/docker start -a c1
```

18) Attach to the running container:

```
# docker attach c1
Enter
root@492e9afe4cb4:/#
root@492e9afe4cb4:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.4 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.4 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@492e9afe4cb4:/# [Ctrl]+[P][Ctrl]+[Q]
```

Cleanup

19) Stop and disable the systemd unit, and remove the unit file:

```
# systemctl stop ubuntu-container
# systemctl disable ubuntu-container
rm '/etc/systemd/system/multi-user.target.wants/ubuntu-container.service'
# rm -f /etc/systemd/system/ubuntu-container.service
```

20) Remove the container and image:

```
# docker rm c1
c1
# docker rmi server1:5000/ubuntu
. . . output omitted . . .
```

Content

Docker Images	2
Listing and Removing Images	4
Searching for Images	6
Downloading Images	8
Committing Changes	9
Uploading Images	10
Export/Import Images	11
Save/Load Images	12
Lab Tasks	13
1. Docker Images	14
2. Docker Platform Images	24

Chapter

3

MANAGING IMAGES

Docker Images

The filesystem that will be used by a container

- built by progressively layering tar images

The set of instructions for running containers

- contained in *layer_id*/json

Examine image's metadata with

- `docker inspect image`
- `docker history image`

What is an Image?

Docker images are the foundation for running containers. Images contain two types of things:

1. A collection of layers that are combined to form the filesystem that will be seen within the container.
2. A collection of configuration options that provide the defaults for any containers launched using that image.

The configuration data for an image (contained in a JSON array), provides the information Docker needs to determine the order that

filesystem layers are assembled in. Each layer has a name which is a hash and includes a reference to a parent layer (by hash number). The final base layer lacks the reference to a parent.

When a container is started, the desired image is referenced and a union of the various layers leading from that layer back to the base are mounted into a namespace and handed to the container.

When a container is in operation, a writable layer is created under `/var/lib/docker/tmp/` and mounted as the final layer to store any temporary changes that accumulate.

Inspecting Images

The metadata associated with each image can be viewed using the **docker inspect** command. The **-f** option allows passing of a Go template to select and format the output; for example:

```
# docker inspect nginx:latest
[{"Architecture": "amd64",
  "Author": "NGINX Docker Maintainers \"docker-maint@nginx.com\"",
  "Comment": "",
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
```

```

    "nginx",
    "-g",
    "daemon off;"
],
... output omitted ...
# docker inspect -f 'Image was created on: {{ .Created }}' nginx
Image was created on: 2015-07-14T00:04:53.560160038Z
# docker inspect -f '{{ .Config.Env }}' nginx
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin NGINX_VERSION=1.9.2-1-jessie]

```

The operations committed in each layer of the image can be viewed using the **docker history** command. By default, the CREATED BY column is truncated. To get a full listing include the **--no-trunc** option. The following example shows listing the commit history of the python image:

```

# docker history python
IMAGE                CREATED              CREATED BY                                      SIZE
0f1edeb099a9         6 weeks ago         /bin/sh -c #(nop) CMD ["python3"]             0 B
a6a93f1e673b         6 weeks ago         /bin/sh -c cd /usr/local/bin                   && ln -s easy_i      0 B
2506cb28a791         6 weeks ago         /bin/sh -c set -x                               && mkdir -p /usr/src/pytho  79.41 MB
5e225c0a4afe         6 weeks ago         /bin/sh -c #(nop) ENV PYTHON_PIP_VERSION=7.0.  0 B
ddea991fd03f         8 weeks ago         /bin/sh -c #(nop) ENV PYTHON_VERSION=3.4.3    0 B
52226e2a71d7         8 weeks ago         /bin/sh -c gpg --keyserver ha.pool.sks-keyser 12.61 kB
3009dde7a8a5         8 weeks ago         /bin/sh -c #(nop) ENV LANG=C.UTF-8            0 B
24b3549417de         8 weeks ago         /bin/sh -c apt-get purge -y python.*          1.069 MB
c9e3effdd23a         8 weeks ago         /bin/sh -c apt-get update && apt-get install   385.7 MB
a2703ed272d7         8 weeks ago         /bin/sh -c apt-get update && apt-get install   122.3 MB
7a3871ba15f8         8 weeks ago         /bin/sh -c apt-get update && apt-get install   44.33 MB
df2a0347c9d0         8 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]           0 B
39bb80489af7         8 weeks ago         /bin/sh -c #(nop) ADD file:5de08c81c24812789a 125.1 MB

```

Listing and Removing Images

`docker images` → **list images**

- Only lists top parent images by default
- `-a` to list all layers

`docker rmi image` → **remove images**

- `-f` remove image even if in use
- `-no-prune` do not delete untagged parents

Listing Images

The images already downloaded to the Docker host being used can be listed as follows:

`docker images`

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
server1/postgres	latest	c0ee64808866	7 days ago	265.5 MB
server1/mysql	latest	a5e5891111da	7 days ago	283.5 MB
server1/python	2.7	0d1c644f790b	12 days ago	672.9 MB
server1/elasticsearch	latest	bc6061565360	4 weeks ago	514.7 MB
server1/redis	latest	0ecdcla8a4c9	4 weeks ago	111 MB
server1/nginx	latest	319d2015d149	4 weeks ago	132.8 MB
server1/debian	jessie	bf84c1d84a8f	4 weeks ago	125.1 MB
server1/ubuntu	15.10	906f2d7d1b8b	4 weeks ago	133.6 MB

To see all the layers in an image, use the `-a` option as follows:

`docker images -a`

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
<none>	<none>	61f2d0dc0350	43 hours ago	132.8 MB
server1/nginx	latest	ce293cb4a3c9	43 hours ago	132.8 MB
<none>	<none>	4b1798a9cafa	43 hours ago	132.8 MB
<none>	<none>	2b12d66de277	43 hours ago	132.8 MB
<none>	<none>	51ed40dd3671	43 hours ago	132.8 MB
<none>	<none>	7340fb61bdef	43 hours ago	132.8 MB
<none>	<none>	97df1ddba09e	44 hours ago	125.2 MB
<none>	<none>	438d75921414	44 hours ago	125.2 MB
<none>	<none>	5dd2638d10a1	44 hours ago	125.2 MB
<none>	<none>	aface2a79f55	44 hours ago	125.2 MB

<none>	<none>	9a61b6b1315e	2 days ago	125.2 MB
<none>	<none>	902b87aaaec9	2 days ago	125.2 MB

Removing Images

Images can be removed with the **docker rmi** commands. If the specified image is tagged with another name, then the tag name used in the **docker rmi** command will be removed, but the image will remain. If no other tags use that image, then the image file itself will be removed. If the image is used by a defined container, then a warning is issued instead. To delete images even if they are in use, use the **-f** option. When an image is removed, parent images that are not referenced by another image are also removed:

```
# docker rmi docker.io/bcroft/app1
Untagged: docker.io/bcroft/app1:latest
# docker rmi app_dev
Error response from daemon: Conflict, cannot delete a48057b84954 because the container 9bc0022e0b24 is using it, use -f to force
FATA[0000] Error: failed to remove one or more images
# docker rmi -f app_dev
Untagged: app_dev:latest
Deleted: a48057b8495407198704514ed217da607cfe955fde93944bbd9e08e11f854627
Deleted: b3f21859ded1ee392ab604de36836690fe751bdf08a68e91989a296bc3417ea3
```

Searching for Images

Find image

- Web search: <https://registry.hub.docker.com/>
- Index CLI search: **docker search *image***

Finding Images

Docker images are generally stored in a Docker Registry. A registry can be run on the local network or even local host. When the **docker** command is run, it will connect to the Docker daemon specified by the **-H** option if used, or the local Docker UNIX socket. The daemon in turn will have one or more registries configured. Registries store one or more repositories and can optionally have an associated Index service that can provide the ability to search for the images and their related metadata.

The official Docker registry can be accessed at: <https://registry.hub.docker.com/>. It supports a web search interface and is currently the best way to search for images.

Searching via CLI

The **docker search** command can be used to search the configured registry. The search string provided can optionally include the repository name to further qualify the search. It also allows filtering by number of stars as shown in the following example:

```
# docker search library
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
ubuntu	Ubuntu is a Debian-based Linux operating s...	3660	[OK]	
nginx	Official build of Nginx.	2632	[OK]	
centos	The official build of CentOS.	2134	[OK]	
mysql	MySQL is a widely used, open-source relati...	2122	[OK]	
node	Node.js is a JavaScript-based platform for...	2006	[OK]	
redis	Redis is an open source key-value store th...	1942	[OK]	
postgres	The PostgreSQL object-relational database ...	1909	[OK]	
mongo	MongoDB document databases provide high av...	1694	[OK]	
jenkins	Official Jenkins Docker image	1306	[OK]	

```
# docker search -s 3 ssh
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
docker.io/jdeathe/centos-ssh	CentOS-6 6.6 x86_64 / EPEL Repo. / OpenSSH...	3		[OK]

If more details are needed, the Registry REST API can be used. For example, to get a list of all tags associated with the ubuntu image you could run:

```
# curl https://registry.hub.docker.com/v1/repositories/ubuntu/tags
```

```
[{"layer": "d2a0ecff", "name": "latest"}, {"layer": "3db9c44f", "name": "10.04"},  
  {"layer": "6d021018", "name": "12.04"}, {"layer": "6d021018", "name": "12.04.5"},  
  {"layer": "c5881f11", "name": "12.10"}, {"layer": "463ff6be", "name": "13.04"}, . . . snip . . .
```

Many tools exist that could parse and format the JSON in easier to read ways. For example:

```
# curl https://registry.hub.docker.com/v1/repositories/ubuntu/tags | python -mjson.tool
```

% Total	% Received	% Xferd	Average	Speed	Time	Time	Time	Current
			Dload	Upload	Total	Spent	Left	Speed
100	2502	0	2502	0	0	313	0	--:--:-- 0:00:07 --:--:-- 574

```
[  
  {  
    "layer": "d2a0ecff",  
    "name": "latest"  
  },  
  {  
    "layer": "3db9c44f",  
    "name": "10.04"  
  },  
  {  
    "layer": "6d021018",  
    "name": "12.04"  
  },  
  . . . output omitted . . .
```

Downloading Images

```
docker pull image
```

- `-a|--all-tags=true`

Downloading Images

Once the desired image has been identified, it can be downloaded using the **docker pull *image*** command. At a minimum, a bare image name can be used in which case it will try to connect to the main Docker Registry and pull the latest official library image by that name. Optionally, the registry host, repository name, image name, and tag can all be specified if desired. As an example, assuming that tag version 22 is the same as tag latest, all of the following commands would download the same image:

```
# docker pull fedora
# docker pull fedora:latest
# docker pull fedora:22
# docker pull registry.hub.docker.com/fedora:22
```

To download tag version v2 of the image named `project_red` from the `bcroft` repository from the registry running on `server1` port 5000, use:

```
# docker pull server1.example.com:5000/bcroft/project_red:v2
```

All tags for an image can be pulled by adding the `-a` option when executing the pull. Be careful with this option as some repositories have a large number of images version, and corresponding tags.

Committing Changes

docker commit *container* [*REPO[:TAG]*]

- changes to filesystem automatically included
- **-c** → change container metadata
- **-a** *author*
- **-m** *commit message*

docker diff *container*

- lists accumulated changes to container's filesystem

Committing Filesystem Changes

Uncommitted changes to a container's filesystem are lost when the container is removed. If desired, changes can be saved to a new image layer that lists the current container top layer as its parent. This would allow future containers to be started with the new layer and its associated changes, or with the older original layer. After making changes to the filesystem, commit the changes by running **docker commit** as shown in the following example:

```
# docker commit -a "bcroft <bryan@example.com>" \  
-m "Add borken service" mega_app:latest
```

Committing Metadata Changes

To make changes to the metadata, use the **-c** | **--change** option and pass the desired Dockerfile style commands. For example, to create a new layer that updates the default command that is run and also defines a new environment variable you might run:

```
# docker commit -c 'CMD ["python", "app2.py"]' -c 'ENV ver=2'
```

Listing Changes to Container Filesystem

To list all the accumulated changes to a container's filesystem, use the **docker diff** command as shown in the following example. The letter C indicates that the file or path data or metadata is changed. The letter A indicates a new file:

```
# docker diff web1  
C /usr  
C /usr/bin  
A /usr/bin/curl  
A /usr/bin/c_rehash  
. . . snip . . .
```

Using the Official Docker Hub

A free public registry is run by the Docker project. After creating an account, you can authenticate via the **docker login** command and upload images to repositories that you have created. The free account only allows a single private repository to be created, but does not limit the use of public repositories. The following example shows authenticating using the interactive mode instead of passing the information via options:

```
# docker login
Username: bcroft
Password: *****
Email: bcroft@example.com
WARNING: login credentials saved in /root/.dockercfg.
Login Succeeded
```

The stored auth credential are used to authenticate to the remote registry for push/pull requests:

```
File: ~/.dockercfg
{
    "https://index.docker.io/v1/": {
        "auth": "YmNyY3Z0OnB3b3YyMDE1RA==",
        "email": "bcroft@example.com"
    }
}
```

The cached credentials can be deleted as follows:

Uploading Images

```
docker login
  • -e|--email=
  • -p|--password=
  • -u|--username=
~/.dockercfg
docker push
```

```
# docker logout
Remove login credentials for https://index.docker.io/v1/
```

Uploading Images

To upload an image, first tag it with the remote registry/repository information, then issue the push:

```
# docker tag app_dev docker.io/bcroft/app_prod
# docker images
REPOSITORY          TAG      IMAGE ID      CREATED      V SIZE
app_dev              latest   a48057b84954  4 min ago   2.43 MB
docker.io/bcroft/app_prod latest   a48057b84954  4 min ago   2.43 MB
# docker push bcroft/app_prod
```

```
Do you really want to push to public registry? [y/n]: y
The push refers to a repository [docker.io/bcroft/] (len: 1)
a48057b84954: Image already exists
b3f21859ded1: Image successfully pushed
8c2e06607696: Image successfully pushed
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image successfully pushed
Digest: sha256:c6dc9a2ac8a34923a2404df82506761362460056→
d86cdbb367b4812ef9a5d3a6
```

Export/Import Images

docker export *container*

- generate consolidated tar archive from specified container's filesystem layers
- STDOUT by default, **-o** *filename*

docker import *image.tar*

- **-c** | **--change=** apply Dockerfile instruction(s) on import

Exporting a Container's Filesystem

In addition to pushing and pulling images, images can be transported from one Docker host to another by exporting them into a tar archive and then importing them again on the destination Docker host. The **docker export** command requires a container id (not image name) and then outputs the consolidated layers to STDOUT as a tar archive. The mounted temp layer for the container that has accumulated any runtime changes is expected to be ephemeral data only and is not included in the export. Additionally, data volumes will not be included in the export. The following example shows exporting a container's filesystem and then examining the resulting tar archive:

```
# docker export container42 > container42.tar
# tar tvf container42.tar
-rwxr-xr-x 0/0          0 2015-07-16 13:03 .dockerenv
-rwxr-xr-x 0/0          0 2015-07-16 13:03 .dockerinit
drwxr-xr-x 0/0          0 2015-07-13 11:28 bin/
-rwxr-xr-x 0/0      1029624 2014-11-12 16:08 bin/bash
-rwxr-xr-x 0/0       51912 2015-03-14 09:47 bin/cat
-rwxr-xr-x 0/0       14560 2014-09-08 01:01 bin/chacl
-rwxr-xr-x 0/0       60072 2015-03-14 09:47 bin/chgrp
. . . output omitted . . .
```

Importing a Filesystem Image

Tar archives created with **docker export**, or filesystem images created via other methods, can be imported into Docker using the **docker import** command. Note that since the export process collapses all layers into a single archive, the intermediate layers and associated metadata history are lost. Subsequently, it is not the preferred method for backing up images. The following example shows importing an image from a tar archive and assigning a tag:

```
# docker import - myapp:latest <container42.tar
adf4f848c700914b954bd4dac651ad23e96015a00d2a6f7df8439876da491ce4
# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        VIRTUAL SI
myapp         latest   adf4f848c700   39 seconds ago 131.2 MB
```

Save/Load Images

docker save *image*

- save image to tar archive
- STDOUT by default, **-o *filename***

docker load *image*

- load image from tar archive
- STDIN by default, **-i *filename***

Saving an Image

The **docker save** command will save the specified image including all its individual layers and associated metadata. This is an excellent way to backup an image. The following example shows saving an image into a compressed archive and then examining its contents:

```
# docker save agentapp:v47 | gzip agent47.tgz
# tar tvzf agent47.tgz
drwxr-xr-x 0/0          0 2015-07-16 13:30 2b12d66de2777efbc3fcb84ac615f358573137f1e7e6d31078414a9376b8019a/
-rw-r--r-- 0/0          3 2015-07-16 13:30 2b12d66de2777efbc3fcb84ac615f358573137f1e7e6d31078414a9376b8019a/VERSION
-rw-r--r-- 0/0        1620 2015-07-16 13:30 2b12d66de2777efbc3fcb84ac615f358573137f1e7e6d31078414a9376b8019a/json
-rw-r--r-- 0/0       3072 2015-07-16 13:30 2b12d66de2777efbc3fcb84ac615f358573137f1e7e6d31078414a9376b8019a/layer.tar
drwxr-xr-x 0/0          0 2015-07-16 13:30 438d75921414c48c09defacd519b2339fd5c06a1192562b40e68465fd06df017/
-rw-r--r-- 0/0          3 2015-07-16 13:30 438d75921414c48c09defacd519b2339fd5c06a1192562b40e68465fd06df017/VERSION
-rw-r--r-- 0/0       1615 2015-07-16 13:30 438d75921414c48c09defacd519b2339fd5c06a1192562b40e68465fd06df017/json
-rw-r--r-- 0/0       1024 2015-07-16 13:30 438d75921414c48c09defacd519b2339fd5c06a1192562b40e68465fd06df017/layer.tar
. . . output omitted . . .
```

Loading an Image

The **docker load** command will load the specified image (contained in an optionally compressed tar archive) into the Docker image store. The archive is expected to contain a `repositories` file that defines tags, and one or more directories named by digest values which contain `layer.tar` and `json` layer metadata files. The following example shows loading a compressed image:

```
# docker load -i agent47.tgz
```


Lab 3

Estimated Time: 40 minutes

Task 1: Docker Images

Page: 3-14 Time: 35 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Task 2: Docker Platform Images

Page: 3-24 Time: 5 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Objectives


- ☞ Pull and push images.
- ☞ Make and commit a change to an image.
- ☞ Export, change, and import a container's filesystem.
- ☞ Save, examine, and load a docker image.

Requirements

- 🖥 (1 station) 🖥 (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Verify that your system has the Docker daemon running, and has no images:

```
# docker images  
REPOSITORY          TAG          IMAGE ID          CREATED          VIRTUAL SIZE
```

- 3) Download the latest Ubuntu image from the classroom registry:

```
# docker pull server1:5000/ubuntu  
Trying to pull repository server1:5000/ubuntu ...  
09b55647f2b8: Download complete  
6071b4945dcf: Download complete  
5bff21ba5409: Download complete  
e5855facec0b: Download complete  
8251da35e7a7: Download complete  
6b977291cadf: Download complete  
baea68532906: Download complete  
Status: Downloaded newer image for server1:5000/ubuntu:latest  
# docker images  
REPOSITORY          TAG          IMAGE ID          CREATED          VIRTUAL SIZE  
server1:5000/ubuntu  latest      09b55647f2b8     11 days ago     188.3 MB
```

Lab 3 Task 1 Docker Images

Estimated Time: 35 minutes

- Since no tag is specified, the default of "latest" will be used.

- 4) View all the layers that compose the ubuntu:latest image:

```
# docker history server1:5000/ubuntu
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
ae8ceea7a06f	8 months ago	/bin/sh -c #(nop) COPY file:36b0236214b470b52	228 B	
<missing>	8 months ago	/bin/sh -c mv /etc/apt/sources.list /etc/apt/	1.895 kB	
<missing>	8 months ago	/bin/sh -c #(nop) MAINTAINER Bryan Croft <bcr	0 B	
<missing>	8 months ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0 B	
<missing>	8 months ago	/bin/sh -c sed -i 's/^#\s*\s*(deb.*universe\)\$/'	1.895 kB	
<missing>	8 months ago	/bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic	194.5 kB	
<missing>	8 months ago	/bin/sh -c #(nop) ADD file:8c057200590b935e07	188.1 MB	

- 5) Tag the image with a simpler name for convenience in launching containers:

```
# docker tag server1:5000/ubuntu ubuntu
```

- 6) Launch a container using the image and verify that the curl command is not currently in the image:

```
# docker run --name c1 -ti ubuntu
/#
/# curl
bash: curl: command not found
```

- 7) Install the curl package into the container:

```
/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package curl
/# apt-get update
. . . snip . . .
Get:5 http://server1.example.com binary/ Packages [14.8 kB]
Get:6 http://server1.example.com trusty/main amd64 Packages [515 kB]
Fetched 873 kB in 1s (801 kB/s)
Reading package lists... Done
/# apt-get install -y --force-yes curl
Reading package lists... Done
```

```

Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  ca-certificates krb5-locales libasn1-8-heimdal libcurl3 libgssapi-krb5-2
. . . snip . . .
Processing triggers for libc-bin (2.19-0ubuntu6.6) ...
Processing triggers for ca-certificates (20130906ubuntu2) ...
Updating certificates in /etc/ssl/certs... 164 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.

```

- 8) Minimize the changes that will be stored in the new image layer by deleting unneeded info, and then exit:

```

/# apt-get clean
/# history -c
/# exit
exit

```

- 9) Have Docker report on the changes to the container's filesystem:

```

# docker diff c1
C /usr
C /usr/bin
A /usr/bin/curl
A /usr/bin/c_rehash
. . . snip . . .
# docker diff c1
C /usr
C /usr/local
C /usr/local/share
A /usr/local/share/ca-certificates
. . . snip . . .

```

- Note that the diff output order is not consistent between runs.

- 10) Commit the changes into a new layer and tag the new image as mydev:v1:

```

# docker commit -a "StudentX <student@example.com>" -m "Add curl" c1 mydev:v1
sha256:a781c5b4f31ff763b361234df230abe18375097116cbe4fbc3db9a106303e404

```

- 11) Launch a new container that uses the modified image and verify that it has the curl command:

```
# docker run --name c2 -ti mydev:v1
/# which curl
/usr/bin/curl
/# curl -v
curl 7.35.0 (x86_64-pc-linux-gnu) libcurl/7.35.0 OpenSSL/1.0.1f zlib/1.2.8 libidn/1.28 librtmp/2.3
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtmp rtsp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IDN IPv6 Largefile NTLM NTLM_WB SSL libz TLS-SRP
/# exit
exit
```

- 12) Tag the new image so that it can be pushed back up to the registry:

```
# docker tag mydev:v1 server1:5000/studentX/mydev:v1
```

- 13) Push the image to the classroom registry, and confirm:

```
# docker push server1:5000/studentX/mydev:v1
The push refers to a repository [server1:5000/studentX1/mydev] (len: 1)
Sending image list
Pushing repository server1:5000/studentX1/mydev (1 tags)
48e51c772484: Layer already exists
f86c02b0df1a: Layer already exists
030a3fe9584f: Layer already exists
5f70bf18a086: Layer already exists
2c1bd54c37a8: Layer already exists
3873458c48b8: Layer already exists
0ea724891d39: Pushed
v1: digest: sha256:05b30740b97bf97c80daalce260d968777ba10d1ba16e1e55c23e93c6c82a462 size: 1960
# curl server1:5000/v2/_catalog
{"repositories":["busybox","nginx","studentX/mydev","ubuntu",...]}
```

• Lower layers need not be pushed because they already exist within the remote registry.

- 14) Create another variant of the image that uses the newly installed Curl program as the entrypoint, so that args can be passed to it, and tag it as v2:

```
# docker commit -a 'StudentX <student@example.com>' \
```

```
> -c 'ENTRYPOINT ["/usr/bin/curl"]' c2 server1:5000/studentX/mydev:v2
sha256:a57f67de34ae6725f4fbb8082a07c387d44381bf99dd137e82aaf1c876fc40f5
```

- 15) Test the new image to verify that the entrypoint works:

```
# docker run --rm server1:5000/studentX/mydev:v2 -v
curl 7.35.0 (x86_64-pc-linux-gnu) libcurl/7.35.0 OpenSSL/1.0.1f zlib/1.2.8 libidn/1.28 librtmp/2.3
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtmp rtsp smtp smtps telnet tftp
Features: AsynchDNS GSS-Negotiate IDN IPv6 Largefile NTLM NTLM_WB SSL libz TLS-SRP
```

- 16) Push the v2 tagged image up to the remote registry into the previously created repository:

```
# docker push server1:5000/studentX/mydev:v2
... output omitted ...
0ea724891d39: Layer already exists
v2: digest: sha256:11ca0157eb83ed95ac13b427a97e0fee01a89de4c9d8926884a715e6437a4ee8 size: 2167
```

- 17) Use the Curl binary in the new image to connect to the classroom registry and list all variants (tags) for your image via the Registry REST API:

```
# docker run --rm server1:5000/studentX/mydev:v2 \
> -sS http://server1:5000/v2/studentX/mydev/tags/list
{"name":"studentX/mydev","tags":["v1","v2"]}
```

Exporting and Importing Images

- 18) Export the filesystem image of the previously created container and unpack it into directory for examination:

```
# mkdir /tmp/image-export
# cd /tmp/image-export
# docker export c2 | tar x
```

- 19) Verify that this is the modified image that contains the curl command:

```
# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
# find . -name curl
./usr/bin/curl
```

```
./usr/share/doc/curl
```

- 20) Make another change to the filesystem by adding a new file to the root:

```
# echo "v1.1 change" > readme
```

- 21) Archive (tar) the filesystem and import it back into Docker tagged as v1.1:

```
# tar c . | docker import - mydev:v1.1
sha256:681cc0a222065ca984b9682eb407c2ad98ce00df7b010f3c72f7e70f25826615
# docker images | grep ^[Rm]
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mydev	v1.1	681cc0a22206	31 seconds ago	175.5 MB
mydev	v1	a781c5b4f31f	40 minutes ago	200.7 MB

The mydev:v1.1 is a new image consisting of all the previous layers collapsed. Although this generally results in a smaller image, the commit history is now gone, and the image won't be able to share layers with the images it was originally built on. This way of creating a modified image is rarely desirable and not considered best practice.

- 22) Verify that the imported image contains the additional file by launching a new container and trying to access the file:

```
# docker run --rm mydev:v1.1 /bin/cat readme
v1.1 change
```

Saving and Loading Images

- 23) Save the modified mydev:v1 image layers so they can be examined:

```
# mkdir /tmp/image-save
# cd /tmp/image-save
# docker save mydev:v1 | tar x
```

- 24) Compare the structure of the saved image layers with the layers stored by the local Docker process:

```
# ls -F
```

```

038018d99ee757a2556a5128c7f528803ecd10eac1753ee8f2f47e02b48b09bc/
083f56e714ec1eaa29ecelc64af36c6bb6ef63c9ef2c6597edb7cb1388d54f31/
5b69eeda9041268b37a106ccf5354ef2c6632bf2fd216cbe2181e83558c2f586/
5e096974a66d7ad8c336111513948db1b304cac8be317a79dedb08ae5ba3ee19/
623e3c7fbb2c6b71406d0ecc87fc2e7e2cc8d9e4991b5c3a63c3df1c13a6ff4b/
6cafb7bc591d8e441a48bf51c56cda96afbd19b8e2b57a01fb56a6908d143341/
dbcd063980209fe1e5a4938ee6fe7592a3e836ee01c3d1dae598169e0d6664df/
ebc52fba86e3984d8ca6911b9eee4ad3fe7952e84bed57f6ee3ec99eade5a5b3.json
eca838e93dc15acf6a979eb99961a5656edade094bf34ba61f139e16a2103a86/
manifest.json
repositories
# cat manifest.json | python -mjson.tool
[
    {
        "Config": "ebc52fba86e3984d8ca6911b9eee4ad3fe7952e84bed57f6ee3ec99eade5a5b3.json",
        "Layers": [
            "038018d99ee757a2556a5128c7f528803ecd10eac1753ee8f2f47e02b48b09bc/layer.tar",
            "5b69eeda9041268b37a106ccf5354ef2c6632bf2fd216cbe2181e83558c2f586/layer.tar",
            "eca838e93dc15acf6a979eb99961a5656edade094bf34ba61f139e16a2103a86/layer.tar",
            "dbcd063980209fe1e5a4938ee6fe7592a3e836ee01c3d1dae598169e0d6664df/layer.tar",
            "623e3c7fbb2c6b71406d0ecc87fc2e7e2cc8d9e4991b5c3a63c3df1c13a6ff4b/layer.tar",
            "5e096974a66d7ad8c336111513948db1b304cac8be317a79dedb08ae5ba3ee19/layer.tar",
            "083f56e714ec1eaa29ecelc64af36c6bb6ef63c9ef2c6597edb7cb1388d54f31/layer.tar",
            "6cafb7bc591d8e441a48bf51c56cda96afbd19b8e2b57a01fb56a6908d143341/layer.tar"
        ],
        "RepoTags": [
            "mydev:v1"
        ]
    }
]
# sha256sum 038018d99e...2b48b09bc/layer.tar • Use the first layer ID from the manifest output.
0ea724891d39006f7c9d4704f7307191e2fae896604911f478a1ae09ae7813f3 038018d99e...2b48b09bc/layer.tar
# ls -F /var/lib/docker/image/devicemapper/layerdb/sha256/
0ea724891d39006f7c9d4704f7307191e2fae896604911f478a1ae09ae7813f3/ • Matching image layers stored by docker
1a9dcce1578874aa8bae8f886bb980ce07d3ca4a3b73f2878562259dc5394eb9/
1aeffc90c7f0a72e8c2cc6fa7e4a3ad78318f51e44a1e644e57486c96acbc99/
. . . output omitted . . .
# cat ebc52fba86e3984...9eade5a5b3.json | python -mjson.tool • JSON file is the config from the manifest.
. . . snip . . .
    "rootfs": {
        "diff_ids": [

```



```


        "sha256:0ea724891d39006f7c9d4704f7307191e2fae896604911f478a1ae09ae7813f3",
        "sha256:3873458c48b8bc033deb7642793140d2f9ddbdf6cef44649f53a05a18f383c97",
        "sha256:2c1bd54c37a8c71aa3c0c6161204362c630775df014ed7825285a15556e8929c",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
        "sha256:030a3fe9584fb61f8e5f7a8f548eab71645ee5af4f7980d1372cf33e0aa5b8f1",
        "sha256:f86c02b0df1a0f1f7d43a05a3908bf55c19ba65366d3bebe66bc4a7939335369",
        "sha256:48e51c7724844f1d7ad59628f11df9b6135d1345b44a97b359be665a2ae98808"
    ],
    "type": "layers"
#

```

Prior to 1.10, Docker stored the corresponding image layers in `/var/lib/docker/graph/layer_id_name`. 1.10 moved to using content addressable image IDs that are a SHA256 hash of the actual image's content and now stores those layers in the `/var/lib/docker/image/devicemapper/layerdb/sha256/` path by default.

- 25) Determine which layer is the top of the stack (the last commit to the image) and change to the corresponding directory:

```

# cat repositories
{"mydev":{"v1":{"4c496cee7f8588c1bbd8d439402ef1349124e9224adb7520eacd80bb24cba51e"}}}
# cd 4c496cee7f8588c1bbd8d439402ef1349124e9224adb7520eacd80bb24cba51e/ — •USE SHELL COMPLETION 

```

- 26) Examine the files which define this layer and verify that it is your commit containing the curl command:

```

# ls
json  layer.tar  VERSION
# sed 's/,/\n/g' json
{"id":"98c86fd202ec8d4a6a50d917a1f69fcec83d1297543a03735d398f34a641a29e"
"parent":"981b0f18b5d6aaf9766057db958a6ce63d51680a6113033688c24ce5aa104a78"
. . . snip . . .
"docker_version":"1.10.3"
"author":"StudentX \u003cstudent@example.com\u003e"
. . . snip . . .
# tar tvf layer.tar | grep bin/curl
-rwxr-xr-x 0/0      154328 2016-04-14 12:03 usr/bin/curl

```

- 27) Repackage the layers into a compressed file that could be distributed and used by Docker:

```
# cd /tmp/image-save/
# tar czf /tmp/mydev.v1.tgz .
# ls -lh /tmp/mydev.v1.tgz
-rw-r--r--. 1 root root 68M Apr 14 13:10 /tmp/mydev.v1.tgz
```

- 28) Remove all of the images:

```
# docker rmi -f $(docker images -q) 2>/dev/null
. . . output omitted . . .
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
------------	-----	----------	---------	--------------

- 29) Load the mydev:v1 image (and all of its layers) back into Docker storage from the local compressed archive:

```
# docker load -i /tmp/mydev.v1.tgz
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
mydev	v1	4c496cee7f85	42 minutes ago	209.1 MB

- 30) Pull the original ubuntu image from the remote registry:

```
# docker pull server1:5000/ubuntu
Using default tag: latest
latest: Pulling from ubuntu
dca023eb4c26: Already exists
95ba0485b693: Already exists
1aae2409782a: Already exists
a3ed95caeb02: Already exists
4a9d3a6c5fce: Already exists
a8a57b5d803f: Already exists
Digest: sha256:f05b37037cf4457cda6b11eb4c6f4fdcf2aa8ba688d819d7de7bf9249ec3c86b
Status: Downloaded newer image for server1:5000/ubuntu:latest
```

Notice that the "download" was basically instantaneous. This is because that actual filesystem layers were already in local Docker storage as part of the mydev

image. Effectively the only change was the base image for ubuntu was tagged and given a repository name.

31) Restore the v2 variant by pulling from the remote registry:

```
# docker pull server1:5000/studentX/mydev:v2
v2: Pulling from studentX/mydev
dca023eb4c26: Already exists
95ba0485b693: Already exists
1aae2409782a: Already exists
a3ed95caeb02: Already exists
4a9d3a6c5fce: Already exists
a8a57b5d803f: Already exists
cfa4c2ac93b5: Already exists
4277ba035d4d: Pull complete
Digest: sha256:11ca0157eb83ed95ac13b427a97e0fee01a89de4c9d8926884a715e6437a4ee8
Status: Downloaded newer image for server1:5000/studentX/mydev:v2
```

32) Verify that all three images again are listed in the local Docker images store:

```
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
server1:5000/studentX/mydev	v2	3e8a810affc1	56 minutes ago	200.7 MB
mydev	v1	1e2af1c32183	2 hours ago	200.7 MB
server1:5000/ubuntu	latest	09b55647f2b8	6 weeks ago	188.3 MB

Cleanup

33) Remove the unneeded directories and files in /tmp:

```
# cd; rm -rf /tmp/image-{save,export}
# rm /tmp/mydev.v1.tgz
```

Objectives

- 🔗 Create a basic platform container image.

Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance


Lab 3

Task 2

Docker Platform Images

Estimated Time: 5 minutes

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Install a minimal R7 system into the /srv/R7 directory using yum:

```
# mkdir /srv/R7  
# yum -y --releasever=7 --nogpg --installroot=/srv/R7 install passwd yum redhat-release-server vim-minimal  
. . . snip . . .  
systemd-libs.x86_64 0:208-20.el7_1.2          tzdata.noarch 0:2015a-1.el7  
ustr.x86_64 0:1.0.4-16.el7                   util-linux.x86_64 0:2.23.2-21.el7  
xz-libs.x86_64 0:5.1.2-9alpha.el7            yum-metadata-parser.x86_64 0:1.1.4-10.el7  
zlib.x86_64 0:1.2.7-13.el7
```

Complete!

- 3) Create a new Docker platform image based on the directory structure:

```
# cd /srv/R7  
# tar c -C /srv/R7 . | docker import - r7base  
a71791b75a46a9b1c1fa9f0b32402387dc6ba43066f37b68529a5d19222ccd4a
```

- 4) Test the new image by launching a container:

```
# docker run -ti --rm r7base /bin/bash  
/# ls  
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var  
/# cat /etc/redhat-release  
Red Hat Enterprise Linux Server release 7.2 (Maipo)  
/# exit
```

5) Create a compressed archive of the image to be used in a later lab:

```
# tar czf /labfiles/R7.tgz -C /srv/R7 /srv/R7
tar: Removing leading `/' from member names
tar: Removing leading `/' from hard link targets
```


Content

Dockerfile	2
Caching	3
docker build	4
Dockerfile Instructions	6
ENV and WORKDIR	7
Running Commands	8
Getting Files into the Image	9
Defining Container Executable	10
Best Practices	11
Lab Tasks	12
1. Dockerfile Fundamentals	13

Chapter

4

CREATING IMAGES WITH DOCKERFILE

Dockerfile

Text file with instructions to build an image

Processed by **docker build**

Results in one or more layers

- images can share layers

Dockerfile

The Docker Hub publishes many useful images that can be downloaded and used. However, the power of Docker is that you can build your own images that match your specific needs. New images can be built in a number of different ways:

1. Using **docker commit** to make an image that includes changes manually made within a container.
2. Using **docker import** to import a tar file that contains a set of files and directories that constitute the desired image.
3. Using **docker build** to create an image based on the instructions within the specified Dockerfile.

This third method, using a Dockerfile, is the most common in most environments. A Dockerfile is a simple text file that contains the build instructions to be processed by **docker build**. Each line within the file starts with an instruction (by convention in all upper case), and is followed by arguments appropriate to that instruction. The full reference for the Dockerfile syntax is found at: <https://docs.docker.com/reference/builder/>

Example Dockerfile

An example of a Dockerfile that might be used to create a simple application image, and that shows several best practices:

File: Dockerfile

```
FROM ubuntu:15.04
MAINTAINER Bryan Croft <bcroft@example.com>
RUN apt-get update && \
    apt-get -y install python
COPY requirements.txt /tmp
RUN pip install /tmp/requirements.txt
COPY . /code
EXPOSE 80
CMD ["python", "/code/app.py"]
```

Image Layers

When building images, each instruction from the Dockerfile that is being processed generally results in a new layer. Through careful planning, and ordering of instructions within the Dockerfile's in use, lower layers can be shared between many different final images. This can result in significant disk space savings for Docker hosts storing many images.

Caching

Each layer produced by an instruction is cached

- can greatly accelerate subsequent builds
- can result in a cached layer being used when a new build is needed instead

Cache invalidations caused by

- changed instruction
- referenced file checksum changed
- previous layer cache invalid

2. Changes to a file that is being referenced by an ADD or COPY instruction. This includes changes to both data and meta-data both of which will result in a different checksum for that file.
3. Any previous layer's cache being invalidated. Since the layers are stacked, changes to a lower level layer will always result in all higher layers being rebuilt.

Overriding Cache Invalidation

For the most part, the automatic methods of invalidating the cache work well, however there are a few times when you may wish to override the behavior. If a RUN instruction references a command that is non-idempotent, the normal checksum methods cannot detect that the layer should be rebuilt when the result of running the command has changed. This can result in subsequent layers being corrupt. The solution is to use **docker build --no-cache** to force cache invalidation.

By default, the base image is not checked to see if it has been updated. Remember that if the base layer is referenced without a tag, the tag of latest is assumed. This tag is often updated within the image registry to point to new base images. To force a pull when building, use: **docker build --pull=true**.

Caching

To speed up the building of image layers, a cache is maintained and checked whenever a new build is initiated. The output of the **docker build** command indicates for each layer if it is being built, or taken from cache as seen in the following sample output where steps 1 and 2 are using cache and step 3 is not:

```
# docker build -t example:v1 .
Step 0 : FROM scratch
--->
Step 1 : MAINTAINER "Bryan Croft"
---> Using cache
---> bf3f4e51d185
Step 2 : COPY file1 /
---> Using cache
---> 6b81efff9642
Step 3 : COPY file2 /
---> 7e78b5fab91a
Removing intermediate container 5a4acb0fb8a9
Successfully built 7e78b5fab91a
```

Cache Invalidations

Several things can cause the cache for a particular image layer to become invalidated (resulting in that layer being rebuilt from scratch):

1. Changing command statement within the Dockerfile. A checksum is created for each line of text within the file, and any change will result in cache invalidation.

docker build

Build directory

- everything passed to Docker daemon
- .dockerignore

-t tag

--no-cache

Using intermediate containers

- --rm=false
- --force-rm

Building Images with docker build

The following basic steps are used to build new images:

1. Create a directory to hold all files related to that image.
2. Create the Dockerfile that defines the build instructions.
3. Run **docker build .** from within the build dir.

When creating and populating a build directory, remember that all files contained within that directory (or subdirectories) will be sent to the Docker daemon as part of the build context (even if they are not used within the image). Files that are not essential for the build operation (especially large files) should not be included in the directory. Remember that the Docker daemon will not always reside on the local host, so large build contexts will potentially be passed across the network.

.dockerignore

To limit the files sent to the daemon, create a .dockerignore file within the build directory. This file can then contain Go filepath.Match globs describing the files to be ignored. Ignored files can include the Dockerfile and the .dockerignore file itself.

See

<https://docs.docker.com/reference/builder/#dockerignore-file> for additional details.

Tags

Most invocations will specify a tag. For example: **docker build -t ssh .** If a tag is not specified, the image is still built, and will show a tag of <none> in the **docker images** output. This image can always be assigned a tag at a later date:

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
<none>	<none>	8d87dd510df5	9 seconds ago	251.4 MB

docker tag 8d87dd510df5 ssh

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	V SIZE
ssh	latest	8d87dd510df5	About a minute ago	251.4 MB

Using Intermediate Images and Containers

When building images, if any instruction results in a non zero return code, the build will fail. Any layers that did build successfully will remain, and one effective method of troubleshooting is to start a container using the previous layer's image and then run the command manually (perhaps with changes to produce more verbose output) to determine the source of the problem:

```
# docker build -t borken:v1 .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
----> d2a0ecffe6fa
... snip ...
Step 6 : RUN sed 's/enabled/disabled/' -i /etc/thing.conf
----> Using cache
----> c91999ebec3e
Step 7 : RUN apt-get -qq install borken 2>/dev/null
----> Running in e7332456eb92
INFO[0002] The command [/bin/sh -c apt-get -y install borken] returned a non-zero code: 100
# docker run -ti c91999ebec3e /bin/bash
root@0db2507034ee:/# apt-get -y install borken
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package borken
```

After a successful build, all the intermediate containers used to process Dockerfile instructions are discarded. If it is useful (generally for troubleshooting), you can suppress this with the `--rm=false` option.

When a build fails, the intermediate containers are not removed by default. To force removal, use the `--force-rm` option.

Dockerfile Instructions

<https://docs.docker.com/reference/builder/>

FROM *image* → **what base image to build on**

- hub.docker.com official library
- scratch

MAINTAINER → **who is responsible**

LABEL → **extra metadata**

Specifying a Base Image

Most images start with a base image, often an official image from Docker Hub. Specifying the base image is done with the FROM instruction. Thousands of images are published via official Docker registry. The official registry features an index and corresponding web interface that allows easy searching for base images. When possible, use official base images which tend to be size optimized, well maintained, and otherwise follow best practices. It is also important to review the Dockerfile associated with the image and other documentation that is published by the image's author.

If an empty base image is desired, then use the scratch image which is interpreted as a key word and forces the use of an empty base image.

When specifying a base image, remember that if no tag or digest is given, then Docker will assume a tag of latest when the image is pulled. This tag can be a moving target, and a tag should always be explicitly listed if a particular version of the image is required for correct operation.

Defining the Author

The MAINTAINER instruction allows specifying information to identify the author of the image. This is embedded into the resulting image and can be queried by **docker inspect**. The structure of the value is not defined, but is commonly a name and email address.

Assigning Additional Metadata

Arbitrary metadata can be embedded into an image in the form of simple key/value pairs. These are declared with the LABEL instruction. Each LABEL instruction results in a new image layer. Best practice is to chain labels on the same line to reduce image layers.

Example Dockerfile

The following example Dockerfile shows the correct use of these instructions:

File: Dockerfile

```
FROM ubuntu:14.04
```

```
MAINTAINER Bryan Croft <bcroft@example.com>
```

```
LABEL subversion="3.14" description="Rule the world"
```

ENV and WORKDIR

ENV → **set an environment variable in containers based on this image**

WORKDIR → **sets a working directory for other instructions**

The ENV Instruction

The ENV instruction allows environment variables to be defined (and exported) within containers based on this image. When launching containers, additional variables can be set, or variables defined by ENV instructions can be overridden when launching containers as follows:

```
# docker inspect -f '{{index .Config.Env 1}}' example
var1=default
# docker run --rm --env=var1=foo --env=var2=bar example env
var1=foo
var2=bar
. . . output omitted . . .
```

ENV can set multiple variables on the same line (resulting in a single layer) when used with the ENV *key=value ...* form; for example:

File: Dockerfile
ENV var1=val1 var2="value two" \ var3=val\ 3

The WORKDIR Instruction

Simply sets the working directory that is used by RUN, CMD, ENTRYPOINT, COPY, and ADD instructions. Can include variables defined within the Dockerfile; for example:

File: Dockerfile
ENV DATASET=/data1 WORKDIR \$DATASET/ RUN cp * /var/lib/data/app1

The WORKDIR instruction can be overridden when a container is launched by running **docker run --workdir=*dir***.

Running Commands

RUN → **commands to run within container**

New layer created and commands run within a temporary container

If successful, layer is committed and used for next step

Cache and non-idempotent commands

Running Commands Within a Container

The RUN instruction allows for further customization of the base image by automating the execution of commands. For each RUN encountered, a temporary container is started, and the listed commands are run resulting in the commit of a new layer. The output of the build process shows the commands being run, and their output. If a command exits with an error, then the build operation terminates, and Docker shows the error encountered.

The RUN instruction can take the following two forms:

exec form: `RUN ["command", "arg1", "arg2", "..."]` ⇒ This allows running commands in images that lack a shell.

shell form: `RUN commands` ⇒ Passes the command(s) as an argument to `/bin/sh -c` allowing for shell expansion, redirects, etc.

Optimizing Number of Image Layers

When using RUN instructions, seek for a balance in the number of layers generated. Commands should be group logically and ordered in a sequence to allow for lower layers to be reused by other images. When grouping commands, use shell line continuation to improve readability; for example:

File: Dockerfile

```
RUN yum update && yum install -y \  
    vim-enhanced \  
    clusterssh \  
    openssh  
RUN echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
```

Cache Consideration

When ordering RUN instructions, remember that changing the line will result in that layer and all later layers being rebuilt (instead of using the cache). If practical, put commonly edited RUN instructions near the end of the Dockerfile to minimize the number of layers that must be rebuilt.

Most commands result in the exact same action every time they are run. For commands that are not deterministic, and may result in a different result, remember that Docker has no way of detecting this since it only uses a checksum of the instruction line. The most common example of this is updating the package list (ex. `yum update`, or `apt-get update`). To force a rebuild, either use the `docker build --no-cache`, or update a previous line to force the change; for example:

File: Dockerfile

```
ENV update_date="Wed Jul 15 11:06:33 MDT 2015"  
RUN yum update && yum install -y
```

Getting Files into the Image

COPY → **copies files or directories into container**

ADD → **expands local or remote files into container**

- source can be a tar archive (optionally compressed with gzip, bzip2, or xz)
- source can be URL such as <http://server1.example.com/conf/sshd.conf>

The COPY Instruction

The COPY instruction is the preferred way to get files from the local system into an image. It takes two basic forms:

simple text form: **COPY <src>... <dest>** ⇒

JSON array form: **COPY ["<src>", ... "<dest>"]** ⇒ Allows copying where a path or filename contains whitespace.

The source files must be within the build directory. Remember that the Docker daemon doing the build may be on a remote host and only has access to the context it was sent when the build started.

If the source is a directory then its contents are copied to the destination directory, including preservation of metadata.

If the source is multiple files, then the destination must end in a / (indicating it is a directory).

If the source is a list of files, then the destination files created in the image will be owned by **root:root**, but will retain their original mode. If the file ownership is critical, it can be modified by a later RUN directive, or the ADD instruction can be used instead to extract a tar archive.

The ADD Instruction

There is some overlap between the COPY and ADD instructions. Generally, COPY should be used if possible, reserving ADD for cases where its special feature are needed. The ADD instruction takes two basic forms:

simple text form: **ADD src ... dest** ⇒

JSON array form: **ADD ["src", ... "dest"]** ⇒ Allows copying where a path or filename contains whitespace.

The ADD instruction can be used to extract compressed tar archives into an image. It can also be used to retrieve remote file referenced by URL and copy them into the image. If more sophisticated download options are needed such as authentication or proxy support, then install a command like **wget** or **curl** into the image and execute it with the RUN instruction instead.

The following example show using the ADD instruction to create a new image from a local tar file and then download an application and data into it:

File: Dockerfile

```
FROM scratch
ADD rootfs.tar.xz /
ADD http://example.com/proj1/app1 /code
ADD http://example.com/proj1/data1.gz /data
```

Defining Container Executable

CMD → **primarily defines default command for image**

- exec form
- shell form
- parameter form

ENTRYPOINT → **treat container as a command (allows passing of arguments)**

- exec form
- shell form

Defining Defaults for Container Execution

The **CMD** instruction is primarily used to define the default command that is executed when a container is run without specifying a command. When used in this way, it can take two forms:

exec form: **CMD** ["*command*", "*arg1*", "*arg2*", "..."] ⇒ When the container is launched, the specified binary is run via `exec(3)` with args as specified in the JSON array.

shell form: **CMD** *command arg1 arg2 ...* ⇒ The specified command and args are passed as arguments to `/bin/sh -c`.

The exec form should be used when possible; for example: **CMD** ["*apache2*", "-DFOREGROUND"]. It avoids the requirement of a shell in the image and the overhead of the shell and its initial processing of the command line. On the other hand, it is sometimes valuable to have shell processing: variable expansion, wildcard processing, shell redirections, etc. For these instances, the shell form can be a useful shortcut; for example: **CMD** *echo "Hello from \$HOSTNAME"*

parameter form: **CMD** ["*arg1*", "*arg2*"] ⇒ Used in conjunction with an **ENTRYPOINT** to define the default additional args that are passed if none are provided from the command line on container execution.

Treating a Container Like an Executable

The **ENTRYPOINT** instruction specifies the command and optionally args to execute when a container is called. Args passed from the CLI on container execution are appended to the list of args specified by **ENTRYPOINT**. As with the **CMD** instruction, an exec and shell form exist:

exec form: **ENTRYPOINT** ["*command*", "*arg1*", "*arg2*", "..."] ⇒ Specified command and args (plus any CLI args) will be run when container is started. If no args are passed from the CLI then args defined by **CMD** (args form) will be used.

shell form: **ENTRYPOINT** *command arg1 arg2 ...* ⇒ Specified command and args will be passed as args to `/bin/sh -c` when container is run. Any additional args listed on the CLI will be ignored.

If the shell form is used, then the final process that is launched is not PID 1 and subsequently signals sent to the container will not be seen by the process. A significant result of this is that when `docker stop` is run on the container, the process will not get the `SIGTERM` and will be forcibly killed via `SIGKILL` after the normal timeout has lapsed.

Best Practices

Keep images minimal

- one process per container, use linking
- no need for SSH, Vim, or even /bin/sh in every image

Examine official library Dockerfiles

- <https://registry.hub.docker.com/search?q=library>

Best Practices

Containers are designed to be ephemeral. Avoid creating images that require significant setup or manual configuration. Keep the content of an image to a minimum. Full OS images are rarely needed and not the primary use case for Docker. Even basic tools like a text editor can often be omitted from images. Content that must be modifiable can be placed in a volume and either edited from the host system, or create an editor image and launch it using the `--volumes-from` option.

The Docker Hub includes many official images that are well cared for. The dockerfile's used to build the images can be examined for ideas and a better understanding of best practices. It is recommended that you examine the associated Dockerfile and any published documentation for any base images you use.

Maintaining the official image repository is a community effort. You can submit new images for inclusion, or suggest updates to existing images. See https://docs.docker.com/docker-hub/official_repos/ for details.

Lab 4

Estimated Time: 45 minutes

Task 1: Dockerfile Fundamentals

Page: 4-13 Time: 45 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Objectives


- 🔗 Create simple Dockerfiles and use them to build images
- 🔗 Understand the use of the cache and the effect of commands on image layers
- 🔗 Make effective use of the .dockerignore file
- 🔗 Understand the CMD and ENTRYPOINT Dockerfile directives
- 🔗 Use the RUN directive to execute commands while building images

Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Remove all containers and images to ensure the lab starts at a known state:

```
# docker rm -f $(docker ps -aq)  
... output omitted ...  
# docker rmi -f $(docker images -q)  
... output omitted ...
```

Cache and Layers

- 3) Create an empty build directory to contain the Dockerfile:

```
# mkdir ~/build  
# cd ~/build/
```

- 4) Create a simple Dockerfile that will use an empty base image (scratch), define a maintainer (insert your own name), and copy a single file into the image:

File: Dockerfile	
+	FROM scratch
+	MAINTAINER "your_name"
+	COPY file1 /

Lab 4

Task 1

Dockerfile Fundamentals

Estimated Time: 45 minutes

5) Create the referenced file and built the image using default options:

```
# echo data1 > file1
# docker build .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
--->
Step 1 : MAINTAINER "your_name"
---> Running in 4e7eeld745d0
---> bf3f4e51d185
Removing intermediate container 4e7eeld745d0
Step 2 : COPY file1 /
---> 6b81efff9642
Removing intermediate container ccc346f2fa95
Successfully built 6b81efff9642
```

For each line within the Dockerfile, a temporary container is created, the directive is processed and the resulting changes to the image are committed.

6) Examine the images that were created:

```
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
<none>	<none>	926d6f79ed13	4 seconds ago	6B

7) Build the image again, this time assigning a tag:

```
# docker build -t image:v1 .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
--->
Step 1 : MAINTAINER "your_name"
---> Using cache ----- • Cached image used.
---> bf3f4e51d185
Step 2 : COPY file1 /
---> Using cache ----- • Cached image used.
---> 6b81efff9642
Successfully built 6b81efff9642
```

```
# docker images
REPOSITORY      TAG          IMAGE ID          CREATED          VIRTUAL SIZE
image           v1          926d6f79ed13     4 minutes ago   6B
```

Notice that the cached images are used instead of having to regenerate those layers.

- 8) Modify the Dockerfile adding another directive:

```
# echo "COPY file2 /" >> Dockerfile
# echo "data2" > file2
```

- 9) Build the image again using a new tag:

```
# docker build -t image:v2 .
. . . snip . . .
Step 2 : COPY file1 /
---> Using cache
---> 6b81efff9642
Step 3 : COPY file2 /
---> 7e78b5fab91a
Removing intermediate container 5a4acb0fb8a9
Successfully built 7e78b5fab91a
```

- Still using cache up to this layer.
- Building additional layer for the first time, so no cache present.

- 10) Examine the internal structure of the new image listing what layers it contains:

```
# docker save image:v2 | tar tv | awk '/layer/ {print $6}'
7e78b5fab91a27d5c3efe0132c2d1d9d8b9ded5d22a34d85a68cb1d3d297d411/layer.tar
bf3f4e51d185dfff828564135deca41583fff487ab9bc6725ea801c353d7cec1/layer.tar
# docker history image:v2
. . . output omitted . . .
```

The two layers correspond to the two file additions as separate layers.

- 11) Modify the Dockerfile so that the files are added in a single copy:

File: Dockerfile	
	FROM scratch
	MAINTAINER "your_name"
+	COPY file* /
-	COPY file1 /
-	COPY file2 /

- 12) Build the image using a new tag, and verify that it results in one less layer being used:

```
# docker build -t image:v3 .
Sending build context to Docker daemon 5.12 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
--->
Step 1 : MAINTAINER "your_name"
---> Using cache
---> bf3f4e51d185
Step 2 : COPY file* /
---> cf91e195b5a0
Removing intermediate container ee7a3012e569
Successfully built cf91e195b5a0
# docker history image:v3
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
79e7d8523e09	19 seconds ago	/bin/sh -c #(nop) COPY multi:a617247862e02b62	12 B	
5b9e1babe583	4 minutes ago	/bin/sh -c #(nop) MAINTAINER "Bryan"	0 B	

• The change to this line invalidates further use of the cache.

Build Context and .dockerignore

- 13) Create a 1G file in the build root:

```
# dd if=/dev/zero of=~/build/bigfile count=1000 bs=1M
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 6.17367 s, 170 MB/s
```

- 14) Rebuild the image using the same tag:

```
# docker build -t image:v3 .
```

```
Sending build context to Docker daemon 1.049GB
Sending build context to Docker daemon
. . . output omitted . . .
```

• Notice the time spent uploading.

The full build context is uploaded to the Docker daemon (including bigfile) even though it is not referenced by the Dockerfile or needed. This could take even longer if the Docker daemon was on a remote host. Best practice is to avoid putting anything in the build root that is not actually needed.

- 15) As an alternative, configure Docker to ignore the bigfile to speed up builds:

```
# echo "bigfile" > .dockerignore
# docker build -t image:v3 .
Sending build context to Docker daemon 5.12 kB
Sending build context to Docker daemon
. . . output omitted . . .
```

- 16) Remove the files that are no longer needed:

```
# rm -f bigfile .dockerignore Dockerfile
```

CMD Directives

- 17) Create the following two simple scripts in the build root:

File: script1	
+	echo "script1"

File: script2	
+	#!/bin/sh
+	echo "script2"

Notice that the first script lacks the magic interpreter line.

- 18) Set both executable and try running them:

```
# chmod a+x script*
# ./script1
script1
# ./script2
```

script2

From a shell on the host, the magic line is not essential in this case. If not specified, the script is interpreted by the current shell.

- 19) Create a new Dockerfile with the following contents:

File: Dockerfile	
+	FROM server1:5000/busybox
+	MAINTAINER "your_name"
+	COPY script* /bin/

- 20) Build the image using a new tag:

```
# docker build -t image:v4 .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM busybox
Trying to pull repository server1:5000/busybox ...
8c2e06607696: Download complete
cf2616975b4a: Download complete
6ce2e90b0bc7: Download complete
Status: Downloaded newer image for server1:5000/busybox:latest
---> 8c2e06607696
Step 1 : MAINTAINER "your_name"
---> Running in 03b30727436d
---> b57ef4227e6e
Removing intermediate container 03b30727436d
Step 2 : COPY script* /bin/
---> 96d1fcb19e90
Removing intermediate container d559952d7452
Successfully built cbfc3ffea5c
```

- 21) Try running a container and executing the scripts:

```
# docker run --rm image:v4 script1
exec format error
FATA[0000] Error response from daemon: Cannot start container 9811549d284d35bcd70963ec35304532bf4a5a3525ed91a67652654462a6ad2d: [8] System error: exec format error
```

- Fails because without the magic (!) being specified on the first script line, execution via the default entrypoint can't determine the interpreter to use.


```
# docker run --rm image:v4 script2
script2
```

- Works fine if interpreter is specified.

22) Define a default command for the container by adding the following:

File: Dockerfile	
	FROM server1:5000/busybox
	MAINTAINER "your_name"
	COPY script* /bin/
+	CMD ["/bin/sh", "-c", "/bin/script1"]

23) Rebuild the image and test by launching a container without specifying a command:

```
# docker build -t image:v4 .
. . . snip . . .
Step 3 : CMD /bin/sh -c /bin/script1
---> Running in 7faa8bb6ef03
---> 5cf591b20ff2
Removing intermediate container 7faa8bb6ef03
Successfully built 5cf591b20ff2
# docker run --rm image:v4
script1
# docker run --rm image:v4 /bin/script2
script2
```

- Now runs as default command.

- When an alternate command is specified it overrides the CMD instruction.

ENV, ENTRYPOINT, and CMD Directives

24) Create another script in the build root and set it executable:

File: script3	
+	#!/bin/sh
+	echo "from environment: \$VAR1"
+	echo "first arg: \$1"
+	echo "second arg: \$2"

```
# chmod a+x script3
```

25) Edit the Dockerfile so that it has the following content:

File: Dockerfile

```
FROM server1:5000/busybox
MAINTAINER "your_name"
COPY script* /bin/
- CMD ["/bin/sh", "-c", "/bin/script1"]
+ ENV VAR1 foo
+ ENTRYPOINT ["/bin/script3"]
```

26) Build the image using a new tag:

```
# docker build -t image:v5 .
. . . snip . . .
Step 3 : ENV VAR1 foo
---> Running in 64ee53a64ac6
---> 1d815c523838
Removing intermediate container 64ee53a64ac6
Step 4 : ENTRYPOINT /bin/script3
---> Running in 5cac282c9460
---> d107500800d1
Removing intermediate container 5cac282c9460
Successfully built d107500800d1
```

27) Run a container using the image and passing arguments:

```
# docker run --rm image:v5 bar baz
from environment: foo
first arg: bar
second arg: baz
```

Note how the ENTRYPOINT let's you treat the container like an executable and pass arguments.

28) Edit the Dockerfile and add a CMD instruction to provide default arguments:

File: Dockerfile	
	FROM server1:5000/busybox
	MAINTAINER "your_name"
	COPY script* /bin/
	ENV VAR1 foo
	ENTRYPOINT ["/bin/script3"]
+	CMD ["bar", "baz"]

When used with an ENTRYPOINT, the CMD instruction defines default args instead of the default command to execute.

29) Rebuild the image and try running without passing any arguments this time:

```
# docker build -t image:v5 .
... output omitted ...
# docker run --rm image:v5
from environment: foo
first arg: bar
second arg: baz
```

• args provided by the CMD JSON array

30) Run another container this time overriding the arguments by passing one on execution:

```
# docker run --rm image:v5 qux
from environment: foo
first arg: qux
second arg:
```

• Passing any args prevents any of the internally specified ones from being used.

31) Instantiate a container, overriding the value of VAR1:

```
# docker run --rm -e VAR1=flurp image:v5
from environment: flurp
first arg: bar
second arg: baz
```

• Passed variable is used instead of one contained within image.

32) Show that the other environment variables are unaffected during container execution by overriding the entrypoint and executing the env command inside the container:

```
# docker run --rm -e VAR1=flurp --entrypoint=/bin/sh image:v5 -c env
container_uuid=1ccf50da-3507-6bfe-fde5-fbb6390df906
HOSTNAME=1ccf50da3507
SHLV=1
HOME=/root
VAR1=flurp
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

RUN Directive

- 33) Remove the need to change permissions on the scripts before building the image by moving that step into the image build script:

File: Dockerfile	
	FROM server1:5000/busybox
	MAINTAINER "your_name"
	COPY script* /bin/
	ENV VAR1 foo
+	RUN chmod a+x /bin/script*
	ENTRYPOINT ["/bin/script3"]
	CMD ["bar", "baz"]

- 34) Verify that your new build instruction works:

```
# cd /root/build/
# chmod a-x script*
# docker build -t image:v6 .
... snip ...
Step 4 : RUN chmod a+x /bin/script*
---> Running in 2be3e0d23ae7
---> 0cc13117a146
Removing intermediate container 2be3e0d23ae7
... output omitted ...
# docker run --rm image:v6
from environment: foo
first arg: bar
second arg: baz
```

• Remove execute permission to test RUN directive.

- 35) Create a completely new Dockerfile in your build directory with the following content:

```
File: /root/build/Dockerfile
+ FROM server1:5000/ubuntu:latest
+ MAINTAINER "your_name"
+ RUN apt-get update && \
+     apt-get install -y --force-yes curl && \
+     rm -rf /var/lib/apt/lists/*
+ RUN curl http://server1.example.com/index.html > /tmp/doc
```

Note how commands can be chained in a single RUN directive. This results in them being processed in a single image layer.

- 36) Build a new image and examine the output to see the specified commands running:

```
# docker build -t image:v7 .
Sending build context to Docker daemon 7.168 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
Trying to pull repository server1:5000/ubuntu ...
09b55647f2b8: Download complete
6071b4945dcf: Download complete
5bff21ba5409: Download complete
e5855facec0b: Download complete
8251da35e7a7: Download complete
6b977291cadf: Download complete
baea68532906: Download complete
Status: Downloaded newer image for server1:5000/ubuntu:latest
---> 09b55647f2b8
Step 1 : MAINTAINER "your_name"
---> Running in 15972d816ff0
---> 010cde86f527
Removing intermediate container 15972d816ff0
Step 2 : RUN apt-get update && apt-get install -y --force-yes curl && rm -rf /var/lib/apt/lists/*
---> Running in b2a790656961
... snip ...
Setting up curl (7.38.0-4+deb8u2) ...
Setting up libsasl2-modules:amd64 (2.1.26.dfsg1-13) ...
```

```

Processing triggers for libc-bin (2.19-18) ...
Processing triggers for ca-certificates (20141019) ...
Updating certificates in /etc/ssl/certs... 173 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.
---> c0f36195bcd5
Removing intermediate container b2a790656961
Step 3 : RUN curl http://server1.example.com/index.html > /tmp/doc
---> Running in ee8a8f2ca473
   % Total % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  29 100    29    0     0    737      0 --:--:-- --:--:-- --:--:-- 743
---> 8d6fad54cfe8
Removing intermediate container ee8a8f2ca473
Successfully built 8d6fad54cfe8

```

37) Run the container and verify that the content was downloaded into the container:

```

# docker run --rm image:v7 cat /tmp/doc
<html>server1 docroot</html>

```

Content

Overview	2
Data-Link Layer Details	3
Network Layer Details	5
Hostnames and DNS	6
Local Host <--> Container	7
Container <--> Container (same node)	8
Container <--> Container: Links	9
Container <--> Container: Private Network	10
Managing Private Networks	12
Remote Host <--> Container	13
Multi-host Networks with Overlay Driver	14
Lab Tasks	16
1. Docker Networking	17
2. Docker Ports and Links	26
3. Multi-host Networks	37

Chapter

5

DOCKER NETWORKING

Overview

docker0 **bridge**

veth **interface pairs**

IPv4 and IPv6 supported

- routing
- NAT

/etc/sysconfig/docker-network

- \$DOCKER_NETWORK_OPTIONS

Docker Default Networking

Docker has a flexible networking system. By default, it will place containers in their own isolated network namespace and then connect them to the host system (and each other) through a combination of virtual interfaces and a bridge. Nearly every aspect of Docker's network configuration can be changed or disabled (allowing for highly custom network configurations to be manually defined).

The central point of connectivity for containers at the data-link layer is a Linux bridge (called docker0 by default). When a new container is started, a virtual Ethernet (veth) interface pair is created. This pair of interfaces behaves like a tunnel where traffic sent in one interface is received on the other interface. Docker places one member of this interface pair into the container namespace where it becomes eth0 for that container. The other member of the pair is added to the docker0 bridge.

The following example output shows the state of the bridge when two containers are running:

```
# brctl show docker0
bridge name    bridge id        STP enabled    interfaces
docker0        8000.56847afe9799  no             veth2623067
```

At the network layer, Docker dynamically assigns IPv4 and/or IPv6 addresses to the container interfaces. Basic IP routing provides connectivity between containers and the host system. For connectivity to remote systems with IPv4, Docker can automatically expose specified ports through NAT rules. IPv6 enabled containers

with a global address can simply route directly.

Passing Network Options to the Docker Daemon

Some of the network options must be specified when the Docker daemon is launched. Other container specific options are defined when images are built, or containers are started. Although the Docker daemon can be started by hand, on a typical Docker host system, the startup will be integrated with the normal system startup scripts.

On RHEL 7, Docker uses a simple systemd unit file that references environment from the following files:

/etc/sysconfig/docker{,network,storage}. Define network related daemon options as values of the DOCKER_NETWORK_OPTIONS variable.

Data-Link Layer Details

-b|--bridge= → **Alternate bridge**

- OpenvSwitch, weave, flannel, etc.

--mtu= → **Maximum transmittable unit**

--mac-address= → **VETH MAC addresses**

--net= → **Network namespace**

Using an Alternate Switch

By default, data-link layer connectivity for Docker containers is provided by a Linux bridge (docker0) as previously described. This works well for connecting a set of containers that reside on a single host. If a more complex or feature rich solution is needed, Docker's auto-creation of a bridge can be disabled and another piece of software can manage the connectivity. To have Docker use an existing bridge, start the daemon with the **--bridge="alternate_bridge_name"** option.

Connecting containers that reside on multiple hosts can be accomplished by creating tunnels between the hosts. One popular solution is to use OpenvSwitch to create a switch on each Docker host, and then connect them into a mesh with GRE tunnels.

Other multi-host networking solutions include flannel, which runs an additions daemon (flanneld on each Docker host. Hosts are then connected via a back-end encapsulation and routing method (by default simple UDP encapsulation). Weave features include peer message exchanges that allow for rapid adaptation to network topology changes (such as failed links), crypto for data, multicast support, and connections to non-containersized applications.

Starting with Docker 1.9, multi-host networks are natively supported through the **docker network create -d overlay** command.

Maximum Transmittable Unit - MTU

Auto detection of PMTU can be unreliable (firewalls blocking ICMP

fragmentation needed messages, etc.) Tunneling packets to remote Docker hosts will also add additional packet overhead which can result in additional fragmentation and possible packet loss. Calculate the MTU based on the encapsulation method used. For example, if using OvS GRE tunnels, you would have 1500 bytes, minus 14 bytes for Ethernet header, minus 20 bytes for IP header, minus 4 bytes for GRE header yielding an interface MTU of 1462 bytes. Set MTU by passing the **--mtu=** option to the Docker daemon.

MAC Address Range

By default, Docker will assign MAC addresses from the following range: 02:42:ac:11:00:00 → 02:42:ac:11:ff:ff. Docker selects a MAC based on the IP address that will be assigned to the container's interface. MACs use the prefix of 02:42:ac:11 and then the final two octets of the IP address are converted to hex and mapped to the final four hex digits. For example, the IP of 172.17.30.42 would map to a MAC of 02:42:ac:11:1e:2a. To assign a specific MAC addresses, pass the **--mac-address=** option when creating a container.

Network Namespace Considerations

When creating a container, the `--net=` option can be passed to determine what network namespace the container will use. This option accepts five modes:

bridge \Rightarrow container placed in a private network and UTS namespace.

A VETH interface pair will be created, one interface added to the container namespace, and the other to the `docker0` bridge. This is the default.

none \Rightarrow container placed in a private network and UTS namespace.

No VETH interfaces are created, but a loopback interface will exist. Any other network interfaces will need to be created manually.

host \Rightarrow container remains in the root network and UTS namespaces.

No VETH interfaces are created, but all host interfaces are accessible. Container will use the host's `/etc/{hosts,resolv.conf}` files instead of the typical bind mounted private files.

container:*container_name* \Rightarrow container placed in the same shared network and UTS namespace as the specified container. No new VETH interfaces are created, but any interfaces in the other container are seen.

network_{name,id} \Rightarrow container connected to the specified user defined network and to the `docker_gwbridge`.

Network Layer Details

--bip= → **IP assigned to Docker bridge**
--fixed-cidr= → **Range used for containers**
--ipv6=true|false → **Link-local IPv6 address for containers**
--fixed-cidr-v6= → **Globally routable IPv6 network prefix**

Assigning IPv4 Addresses

By default, Docker assigns the bridge and containers IPs from the 172.17.0.0/16 netblock. If this conflicts with addresses already in use, another block can be used by used. The bridge IP is set with the --bip Docker daemon option, and the range of addresses allocated to local containers is set with the --fixed-cidr= option. For example, suppose that the local docker0 bridge is connected via tunnels to remote Docker hosts to form a larger virtual data-link. In this case the bridge IP's associated subnet mask might be a larger block while the pool of IPs for local container use might be a subset of that; for example:

```
File: /etc/sysconfig/docker-network
```

```
+ DOCKER_NETWORK_OPTIONS="--bip=10.200.0.1/16 --fixed-cidr=10.200.3.0/24"
```

Assigning IPv6 Addresses

By default, Docker only assigns an IPv4 address. If the --ipv6=true option is passed to the Docker daemon, then a link-local IPv6 address in the fe80:: prefix will be assigned and the bridge will be assigned fe:80::1. If the --fixed-cidr-v6= option is passed, then the specified routable network prefix will be used (with node addresses formed from the MAC address as usual).

Hostnames and DNS

```
-h|--hostname= → Set hostname  
--add-host=[] → Add extra entries to /etc/hosts  
--dns=[] → Specify DNS server for /etc/resolv.conf  
--dns-search=[] → DNS search path for resolv.conf
```

Hostname and /etc/hosts File

By default, new container names are set to a unique, random hex value. The hostname is then set to this same value, and an entry made in the /etc/hosts file that resolves this to the assigned IP. The Docker container name and container host names can be set independently using the `--name=` and `--hostname=` options. Remember that changes to hostname are only visible within that container. When creating links, references are defined using the container id or name, and not the hostname.

In the following example output the first container is assigned a hostname of `nginx1`, and a dynamic container name and id. The hosts file in the second container shows its dynamically assigned hostname, and an entry for the linked container that has three keys: alias, hostname, and dynamically assigned container name:

```
# docker run -h nginx1 -d nginx  
dc30724b7c3e5d524c78c2f03bed3ddb9ee3a6ced706ed11732feb22d9eae4e  
# docker run --link dc30724b7:web1 busybox cat /etc/hosts  
192.168.0.11      3668b9e0a484  
127.0.0.1        localhost  
192.168.0.10     web1 nginx1 romantic_curie
```

DNS

Name resolution for a wider scope is provided by DNS. If no extra options are specified, then the host system's `resolv.conf` will be bind mounted within the container. If different DNS servers or search paths are desired, then the `--dns=` and `--dns-search=` options can

be used (multiple times if needed).

Local Host <--> Container

Direct to container IP:port

- via docker0 bridge

To published IP:port

- -p or -P
- via Docker proxy and DNAT

--ip= → **Controls host binding IP for published ports**

- Must restart Docker daemon after change

```
80/tcp -> 0.0.0.0:32769
# lsof -Pi -a -c docker
COMMAND  PID USER FD   TYPE DEVICE S/OFF NODE NAME
docker   16670 root  4u    IPv6 363639  0t0  TCP *:32768 (LISTEN)
docker   16677 root  4u    IPv6 363661  0t0  TCP *:32769 (LISTEN)
# iptables -t nat -nL DOCKER
Chain DOCKER (2 references)
target prot source          destination
DNAT  tcp  0.0.0.0/0  0.0.0.0/0  tcp dpt:32768 to:172.17.0.2:443
DNAT  tcp  0.0.0.0/0  0.0.0.0/0  tcp dpt:32769 to:172.17.0.2:80
# curl 127.0.0.1:32769
. . . snip . . .
<title>Welcome to nginx!</title>
```

Changing the Host Proxy Binding Address

The host IP used when binding the proxy for published ports can be specified with the --ip= Docker daemon option. As with all daemon options, Docker must be restarted for it to take effect. For example, to have container service only bind to the loopback address:

```
File: /etc/sysconfig/docker-network
→ DOCKER_NETWORK_OPTIONS="--ip=127.0.0.1"
```

```
# docker run -dP --name web3 nginx
417f8884e0f014c60f095520c8094460659df15735547561df4ed879ef89
# docker port web3
443/tcp -> 127.0.0.1:32768
80/tcp -> 127.0.0.1:32769
```

Connecting Direct to Container Services from the Host System

Because the Docker host has a network interface (the docker0 bridge) in the same network as the containers, the host can connect directly to the IP:port of any service run by a container. This does not require publishing ports to the outside world; for example:

```
# docker run -d --name web1 nginx
c6bda13fe3e83832a3c716367b60b05c1b8197fba1c902f1bde8389cc639
# docker inspect -f '{{ .NetworkSettings.IPAddress }}' web1
172.17.0.1
# curl 172.17.0.1
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
. . . output omitted . . .
```

Connecting to Published Container Ports from the Host System

When the -P or -p run options are used, Docker will select an available port from the ephemeral range defined in /proc/sys/net/ipv4/ip_local_port_range and run a proxy on that port. Connections to that IP:port are processed by Netfilter DNAT rules to map the packets back to the container IP:port; for example:

```
# docker run -dP --name web2 nginx
f3dbedddb7632b413c574d062db26bdb2064d62d0d90fe2109a7af70178d
# docker port web2
443/tcp -> 0.0.0.0:32768
```

Container <--> Container (same node)

Connect to direct container IP:port, or published host IP:port

--iptables=true|false → **Manual or automatic firewall rules**

--icc=true|false → **Default ACCEPT or DROP policy**

Firewall Considerations

By default, all networked containers have an interface placed in the `docker0` bridge allowing connections between them. Firewall rules on the host system can further limit connectivity between containers. If `iptables=false` is set, then you must manage any firewall rules manually. With the default of `--iptables=true`, Docker can create rules as containers are started based on the services they expose.

Assuming `--iptables=true` is set, the `--icc=` daemon option determines what the default policy is for traffic between containers. With the default of `--icc=true`, all inter-container traffic is permitted as follows:

```
# iptables -vL FORWARD
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
target prot opt in out source destination
ACCEPT all -- docker0 docker0 anywhere anywhere
. . . snip . . .
```

If `--icc=false`, then inter-container traffic is blocked unless a manual rule is added before the Docker inserted one. Note that traffic from other non-container IPs to or from containers is still permitted as follows:

```
# iptables -vL FORWARD
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
target prot opt in out source destination
DROP all -- docker0 docker0 anywhere anywhere
DOCKER all -- any docker0 anywhere anywhere
ACCEPT all -- any docker0 anywhere anywhere ctstate RELATED,ESTABLISHED
ACCEPT all -- docker0 !docker0 anywhere anywhere
```

If `--icc=false`, and the `--link` option is used to explicitly define a connection between containers, then Docker will add a pair of firewall rules to allow the traffic while retaining the default DROP rule that prevents a container from connecting to arbitrary ports on the linked container.

Container <--> Container: Links

`--link=[]` → **Make new container aware of "exposed" services in linked container**

- environment variables reveal linked service info within container
- entry made in `/etc/hosts` for linked host

```
WEB_PORT_443_TCP_PROTO=tcp
WEB_PORT_80_TCP_PROTO=tcp
WEB_PORT_80_TCP_ADDR=172.17.0.1
WEB_PORT=tcp://172.17.0.1:80
/# cat /etc/hosts
172.17.0.3      8b0b0b66665e
127.0.0.1      localhost
172.17.0.1     web c6bda13fe3e8 web1
```

Applications needing to connect to linked container's services should use hostname lookups (resolved through the hosts file instead of the environment variables. This is because, the IP assigned to a container can change over its lifetime, and the hosts entry is updated when this happens, but the environment variables are not.

Connecting Containers with Links

For a container to connect to another container's network services, it must be aware of the IP address and ports currently in use by the other container. The IP address assigned to containers is most commonly assigned dynamically. Docker images can specify what ports are used with the `EXPOSE` Dockerfile option.

The `--link=` run option is designed to make it easy to connect a new container with the services on existing containers. When used, the new container will have environment variable that contain the IP:protocol:port info for the exported services of the linked container. The new container will also have an entry added to its hosts file that resolves the linked container's name, id, and alias to its current IP; for example:

```
# docker run -d --name web1 nginx
c6bda13fe3e83832a3c716367b60b05c1b8197fba1c902f1bde8389cc639
# docker inspect -f '{{ .NetworkSettings.IPAddress }}' web1
172.17.0.1
# docker inspect -f '{{ .Config.ExposedPorts }}' web1
map[80/tcp:map[] 443/tcp:map[]]
# docker run -ti --link web1:web debian
/# env | grep WEB_PORT
WEB_PORT_443_TCP_PORT=443
WEB_PORT_443_TCP=tcp://172.17.0.1:443
WEB_PORT_80_TCP_PORT=80
WEB_PORT_80_TCP=tcp://172.17.0.1:80
WEB_PORT_443_TCP_ADDR=172.17.0.1
```

Private Networks with docker network

As previously shown, containers are connected to the default `docker0` bridge or to the existing bridge specified by the **docker run --bridge=*bridge_name***. Starting with the `docker-engine-1.9`, the **docker network** command can be used to manage additional local networks. It is modular and supports a growing number of backends for both bridge types, and IP address allocation.

When creating a new network, a driver is selected. The default is the bridge driver which will create a bridge on the local Docker host only. By default, an IP address range is also associated with the new bridged segment from the `172.X.0.0/16` range. For example:

```
# docker network create net1
231b4ae4673bf476fd7209b139ead5d6811b41f0da3b9abe93b08164e93
# docker network ls -f name=net1
NETWORK ID          NAME                DRIVER
231b4ae4673b        net1               bridge
# brctl show
bridge name        bridge id          STP enabled  interfaces
br-231b4ae4673b    8000.0242eaa09b9c  no
. . . output omitted . . .
# docker run -dti --net=net1 busybox
d232729c7814a05b80756a8bf0689c37449875a4eb185e0798ba90db233
# docker network inspect net1
[ {
  "Name": "net1",
  "Id": "231b4ae4673bf476fd720...164e93",
```

Container <--> Container: Private Network

User defined networks:

- added in docker 1.9 (links now considered legacy)
- networks created with default driver (bridge) do NOT span nodes

docker network {create,ls,inspect,rm}

Modular with backends for bridge type and IP address allocation

- `--internal` to restrict external access to/from network
- `-o "bridge_driver_opt"="value"` to specify equivalent of daemon options used by `docker0` bridge

```
"Scope": "local",
"Driver": "bridge",
"Options": {},
"Config": [
  {
    "Subnet": "172.18.0.0/16",
    "Gateway": "172.18.0.1/16"
  }
],
"Containers": {
  "d232729c7814a05b80756a8bf0689c37449875a4e...b233": {
    "Name": "determined_ritchie",
    "EndpointID": "bf067614103ecca9866ce18384bf...498a",
    "MacAddress": "02:42:ac:12:00:02",
    "IPv4Address": "172.18.0.2/16",
    "IPv6Address": ""
  }
},
"Options": {}
} ]
```


Networks can also be created with a specific bridge name, alternate IP address pool, marked as internal only, and other options equivalent to the docker daemon `--ip-masq`, `--icc`, `--ip`, `--mtu`, and `--ipv6` options. For example, the following would create a private network that does not pass traffic to/from it and other networks, with a custom name and IP range:

```
# docker network create --internal --subnet=192.168.50.0/24 -o "com.docker.network.bridge.name=docker_priv0 priv0
5719f0a6ed53721fe0593a66b5c31d82bdf9cc8e0fb0c5a009de4de6c72ed60d
# docker network ls -f name=priv0
NETWORK ID          NAME                DRIVER
5719f0a6ed53        priv0              bridge
# brctl show
bridge name      bridge id          STP enabled    interfaces
docker_priv0     8000.0242ed9f6a8b  no
. . . output omitted . . .
```

Managing Private Networks

`docker network {connect,disconnect}`

Name resolution on private networks

- docker-engine-1.9 → /etc/hosts automatically updated when hosts are connected/disconnected on that network
- docker-engine-1.10 → embedded DNS server in docker daemon

Connecting Existing Containers to Private Networks

The `docker network {connect,disconnect}` commands can be used to dynamically add and remove existing containers from networks. When changes are made, the hosts file of all containers connected to that network are also updated to reflect the change.

In the following sequence, notice how `host1` is connected to both the default bridge, and the explicitly connected `net1` bridge, while `host2` is only connected to the private `net1` bridge:

```
# docker run -dtih host1 --name host1 busybox
c3cc32a0aa163153cddd17717fec0a3d5ef7f39269c3a226bb609acd0
# docker network create net1
7403a3a68acf51e6fd663539045b5cf48658af061cda14a3223a34c14
# docker network connect net1 host1
# docker exec host1 ip -4 addr li
... snip ...
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    inet 172.17.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
27: eth1@if28: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    inet 172.18.0.2/16 scope global eth1
        valid_lft forever preferred_lft forever
# docker run -dtih host2 --name host2 --net=net1 busybox
72f6957ff44db80a682483b7d0c6dc06d703146315ef651bc576ccffbd3ee574
# docker exec host2 ip -4 addr li
... snip ...
31: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
```

```
    inet 172.18.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
# docker exec host1 cat /etc/resolv.conf
search example.com
nameserver 127.0.0.11
# docker exec host1 ping -c1 host2
PING host2 (172.18.0.4): 56 data bytes
64 bytes from 172.18.0.4: seq=0 ttl=64 time=0.049 ms
... output omitted ...
# brctl show


| bridge name     | bridge id         | STP enabled | interfaces                 |
|-----------------|-------------------|-------------|----------------------------|
| br-7403a3a68acf | 8000.0242b97a1281 | no          | veth6b441b8<br>vethb777f68 |
| docker0         | 8000.0242253f74ac | no          | veth19503eb                |


```

Remote Host <--> Container

```
--ip-forward=true|false → Controls kernel routing of packets
    • /proc/sys/net/ipv4/conf/all/forwarding
--ip-masq=true|false → SNAT for outbound packets
-p, --publish=[] → Publish specific port
-P, --publish-all=true|false → Publish all exported ports
```

Routing of Packets

By default, Docker places containers on the 172.17.0.0/26 IP network (part of the RFC1918 internal space). Connections between the containers and remote hosts require packets to be routed to this internal network. By default, the Docker daemon configures the kernel to route packets. This can be disabled by setting `--ip-forward=false` as a daemon option.

Since the RFC1918 address space is not globally routed, most network configurations will also require packets readdressing via NAT before they leave the local host or network. The default setting of `--ip-masq=true` causes the Docker daemon to add a Netfilter rule sending all container traffic leaving the host to the MASQUERADE target resulting in the traffic source address being changed to the outbound host IP.

Publishing Container Services and DNAT

Getting traffic from the containers to remote hosts is handled via normal routing, and SNAT as described. For return remote traffic to be routed back to the proper container, port forwarding via Netfilter DNAT rules is used. DNAT rules are put in place for all container services that are published via either the `-p`, or `-P` run options.

Individual container services can be published to remote hosts using the `-p IP:host_port:container_port` run option. In the following example output, a container is started mapping port 80 on the host system to port 80 in the container. Notice the Docker proxy listening

to the published port on the host, and the corresponding DNAT rule to map the traffic back to the container service:

```
# docker run --name web1 -p 80:80 -d nginx
daf4672ab6353e56d0011f21460ef5f8addcc69e2eb8ca683ddb3c6da941
# docker inspect -f '{{ .NetworkSettings.IPAddress }}' web1
172.17.0.1
# docker port web1
80/tcp -> 0.0.0.0:80
# lsof -i :80
COMMAND  PID USER FD  TYPE DEVICE SIZE/OFF NODE NAME
docker  11066 root  4u  IPv6 540231      0t0  TCP *:http (LISTEN)
# iptables -t nat -L DOCKER
Chain DOCKER (2 references)
target prot opt source      destination
DNAT  tcp  --  anywhere anywhere  tcp dpt:http to:172.17.0.1:80
```

When a container is started using the `-P` (publish all) option, all ports listed in the image EXPOSE parameter, or specified by the `--expose port` run option are mapped.

Multi-host Networks with Overlay Driver

docker network create -d overlay net_name

- Require kernel 3.16 or higher
- Key/Value store: Consul, Etcd, ZooKeeper
- Cluster of Docker hosts with connectivity to key/value store
- Each Docker host engine launched with appropriate `--cluster-store` options

Multi-host Networks Theory

In real world application, service often require a collection of containers operating together. If the containers are spread across multiple docker hosts, then network connectivity between the containers must be established. The `--link` option will not work across Docker hosts, so third-party processes like Flannel, Weave, or OpenvSwitch were used to create networks that spanned the host. Starting with the Docker 1.9 release, the provided overlay driver can be used instead. It uses VXLANs (Linux kernel 3.16 or higher) to extend bridges between the desired Docker hosts.

Service and Network Discovery

To use overlay networks, a key/value store must be created and the Docker engine on each participating host must be configured to use that store. Docker supports Consul, Etcd, and Zookeeper each of which can act as a distributed key/value store. For fault tolerance, a cluster of hosts are configured as servers for the selected store, and then the docker process on each participating host is made aware of the store by passing a `--cluster-store` (and other related options) when starting the process.

For example, the following could be used to launch a simple Consul server within a container (exposing only the REST API interface port), and to connect a docker host client to that store:

```
[consul.example.com]# docker run -d -p "8500:8500" -h consul progrium/consul -server -bootstrap
```

```
[client.example.com]# /usr/bin/docker daemon --cluster-store=consul://consul.example.com:8500 --cluster-advertise=eth0:2376
```

Creating and Using an Overlay Network

After setting each involved docker daemon to use a common key/value store, networks that spans hosts are created by specifying the overlay driver. When the first multi-host network is created, a new bridge (docker_gwbridge) is also added:

```
[node1]# docker network create -d overlay net1
8a257a4240ff33f59334a76775cb30845c78cf3180ffb4110a5b4bab9957fcd6
[node1]# docker network inspect net1
[ {
```

```

"Name": "net1",
"Id": "8a257a4240ff33f59334a76775cb30845c78cf3180ffb4110a5b4bab9957fcd6",
"Scope": "global",
"Driver": "overlay",
"IPAM": {
  "Driver": "default",
  "Options": null,
  "Config": [
    {
      "Subnet": "10.0.0.0/24",
      "Gateway": "10.0.0.1/24"
    }
  ]
},
"Containers": {
  "a7fe8a1bbcd46d192ea3b6d95386506ae9ed2b34bdf9a452167bd765de81b574": {
    "Name": "host1",
    "EndpointID": "e1b563dc87c47ee40576da53fd43b4edeefea0be1997dd694f0e4d56e3c50b53",
    "MacAddress": "02:42:0a:00:00:02",
    "IPv4Address": "10.0.0.2/24",
    "IPv6Address": ""
  }
},
"Options": {}
} ]

```

brctl show

```

bridge name bridge id STP enabled interfaces
docker0      8000.0242e79082ad no
docker_gwbridge 8000.0242e2b618da no

```

Once created, the network is seen by all hosts connected to the common key/value store. Containers running on different hosts can then be added to the network and communicate with one another:

```

[node1]# docker run -dtih container1 --net=net1 busybox
a7fe8a1bbcd46d192ea3b6d95386506ae9ed2b34bdf9a452167bd765de81b574
[node2]# docker run -dtih container2 --net=net1 busybox
[node1]# docker exec container1 ping -c3 container2
PING container2 (10.0.0.3) 56(84) bytes of data.
64 bytes from container2 (10.0.0.3): icmp_seq=1 ttl=64 time=0.348 ms
64 bytes from container2 (10.0.0.3): icmp_seq=2 ttl=64 time=0.244 ms
64 bytes from container2 (10.0.0.3): icmp_seq=3 ttl=64 time=0.268 ms

--- container2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.244/0.286/0.348/0.048 ms

```

Lab 5

Estimated Time: 90 minutes

Task 1: Docker Networking

Page: 5-17 Time: 30 minutes

Requirements: 🖥️ (1 station) 🖥️ (classroom server)

Task 2: Docker Ports and Links

Page: 5-26 Time: 30 minutes

Requirements: 🖥️ (1 station)

Task 3: Multi-host Networks

Page: 5-37 Time: 30 minutes

Requirements: 🖥️🖥️ (2 stations) 🖥️ (classroom server)

Objectives


- ☞ Specify networking options when launching a container
- ☞ Create and use private local networks

Requirements

- 🖥 (1 station) 🖥 (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Launch a simple Debian container and examine the default network state:

```
# docker run --rm -ti server1:5000/ubuntu  
/# ip addr show dev eth0  
316: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default  
    link/ether 02:42:ac:11:00:9c brd ff:ff:ff:ff:ff:ff  
    inet 172.17.0.156/16 scope global eth0  
        valid_lft forever preferred_lft forever  
    . . . snip . . .  
/# cat /etc/hosts  
172.17.0.156    6b46539dbbdf  
127.0.0.1      localhost  
    . . . snip . . .  
/# cat /etc/resolv.conf  
# Generated by NetworkManager  
search example.com  
nameserver 10.100.0.254  
/# mount | grep /etc  
/dev/mapper/vg0-var on /etc/resolv.conf type xfs  
    (rw,relatime,seclabel,attr2,inode64,noquota)  
/dev/mapper/vg0-var on /etc/hostname type xfs (rw,relatime,seclabel,attr2,inode64,noquota)  
/dev/mapper/vg0-var on /etc/hosts type xfs (rw,relatime,seclabel,attr2,inode64,noquota)  
/# ip route list  
default via 172.17.0.1 dev eth0  
172.17.0.0/16 dev eth0  proto kernel  scope link    src 172.17.0.156
```

- This address was dynamically assigned
- Hostname is random and NOT the specified container name
- This entire file is from the host system
- These files actually reside on the host (/var/lib/docker/containers/name and are mounted
- Default route points to the host docker bridge address

Lab 5

Task 1

Docker Networking

Estimated Time: 30 minutes

Do not exit yet.

3) Try to modify the network configuration (will fail):

```
/# ip link add type veth
RTNETLINK answers: Operation not permitted
/# grep CapEff /proc/self/status
CapEff: 00000000a80425fb
/# capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,
cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
/# exit
```

- Examine the Linux capabilities associated with the current shell
- Decode the string into text

An alternative command for listing the capabilities of a process by pid is
getpcaps.

4) Examine the various capabilities that exist on the system:

```
# man capabilities
/NET
CAP_NET_ADMIN
    Perform various network-related operations:
    * interface configuration;
    * administration of IP firewall, masquerading, and accounting;
    * modify routing tables;
    . . . snip . . .
q
```

The attempt to modify the interface address was blocked because the net_admin capability was not present in the container.

5) Launch another container overriding several network settings and examining the effect:

```
# docker run --cap-add NET_ADMIN \
-ti -h node1 \
--dns=8.8.8.8 --dns-search=example.com \
server1:5000/ubuntu
/# cat /etc/hosts
172.17.0.158    node1
. . . snip . . .
```

- grant the container cap_net_admin
- static hostname instead of random
- specified DNS instead of inheriting from host system


```

/# cat /etc/resolv.conf
nameserver 8.8.8.8
search example.com
/# capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
    cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpc
    ap,cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
Securebits: 00/0x0/1'b0
    secure-noroot: no (unlocked)
    secure-no-suid-fixup: no (unlocked)
    secure-keep-caps: no (unlocked)
uid=0(root)
gid=0(root)
groups=
/# ip link add type veth _____ •Works because the cap_net_admin capability is present
/# ip link show
. . . snip . . .
2: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 0a:96:f2:3b:1b:f1 brd ff:ff:ff:ff:ff:ff
3: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 1e:c3:21:ee:a1:83 brd ff:ff:ff:ff:ff:ff
/# exit

```

- 6) Restart the container and check for the presence of the manually added network interfaces:

```

# docker start -i $(docker ps -lq)
root@debl:/# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
326: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:ac:11:00:a1 brd ff:ff:ff:ff:ff:ff
# exit

```

The veth interfaces are gone. Any runtime changes to the network environment are not persistent across container runs.

Sharing a Network Namespace with Host System

- 7) Run a container that uses the network namespace of the host system instead of an isolated one:

```
# docker run --name web --net=host -d server1:5000/nginx
... output omitted ...
# lsof -i :80
COMMAND PID      USER   FD   TYPE    DEVICE  SIZE/OFF  NODE NAME
nginx   3035      root    6u   IPv4  2801789      0t0  TCP *:http (LISTEN)
lsof: no pwd entry for UID 104
nginx   3046      104     6u   IPv4  2801789      0t0  TCP *:http (LISTEN)
# curl localhost
... snip ...
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

- Launch container as daemon
- Containerized nginx listening on port 80 on the host system
- The nginx user is only defined within the container and not resolvable from the host system

- 8) Connect a shell to the running container to inspect it interactively from the inside:

```
# docker exec -ti web /bin/bash
/# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1        0  0 23:21 ?           00:00:00 nginx: master process nginx -g daemon off;
nginx        12        1  0 23:22 ?           00:00:00 nginx: worker process
root         13        0  0 23:32 ?           00:00:00 /bin/bash
root         17       13  0 23:32 ?           00:00:00 ps -ef
/# hostname
stationX.example.com
/# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:02:00:03 brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
/# exit
```

- Container has other namespaces active such as PID
- The hostname in the container is that of the host system
- All host network interfaces are seen

- 9) Remove the web container:

```
# docker rm -f web
```

Sharing a Network Namespace Between Containers

- 10) Launch a container and start a simple server within it listening on port 5000:

```
# docker tag server1:5000/ubuntu ubuntu
# docker run --name nc-server -ti ubuntu
# while true; do echo "from nc-server" | nc -lp 5000; done &
[1] 26
/# ss -tan
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
LISTEN	0	0	*:5000	*:*

Ctrl + p Ctrl + q

#

- Verify service is bound to all IPs on port 5000
- This detaches from the container, but leaves it running.

- 11) Start another container named nc-client that shares the same network namespace as nc-server, and verify that you can connect to the service running in the other container:

```
# docker run -ti --name nc-client --net=container:nc-server ubuntu
/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	19:26	?	00:00:00	/bin/bash
root	13	1	0	19:42	?	00:00:00	ps -ef

```
/# ss -tan
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
LISTEN	0	0	*:5000	*:*

```
/# nc localhost 5000
from nc-server
/# nc localhost 5000
from nc-server
Ctrl + p Ctrl + q
#
```

- The nc process is not running or visible within this container.
- The network stack does see the listening service.

- 12) Terminate the nc containers:

```
# docker rm -f nc-{server,client}
nc-server
nc-client
```

Private Inter-container Networks

13) Create two local networks that use the default bridge driver:

```
# docker network create net1
962b15f712d235849546121c277945e2fab49a75b17ecec72da3d121b654f627
# docker network create net2
1a8ba2305f8bcc74f942279dc0001d788e2cc4fe62d5dd24b71afd16542bdbcb
# docker network ls -f name=net
NETWORK ID          NAME                DRIVER
962b15f712d2        net1                bridge
1a8ba2305f8b        net2                bridge
```

14) Launch a simple container connected to the first private networks:

```
# docker run -ti --name c1 -h c1 --net=net1 server1:5000/ubuntu
/# ip -4 addr li
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    inet 172.18.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
/# ping -c 1 server1
PING server1.example.com (10.100.0.254): 56 data bytes
64 bytes from 10.100.0.254: icmp_seq=0 ttl=63 time=0.324 ms
--- server1.example.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
[Ctrl]+[p] [Ctrl]+[q]
```

• Even on a private network, containers can still connect to the outside world, just not one another.




15) Launch another container, this time connected to the second private networks:

```
# docker run -ti --name c2 -h c2 --net=net2 server1:5000/ubuntu
/# ip -4 addr li
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    inet 172.19.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
```

```
/# ping -c 1 172.18.0.2
PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
```




- Containers on different private networks can't connect to one another.

```
--- 172.18.0.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

```
++
```

- 16) Create a third container initially connected to the standard docker0 bridge network, and then connect it to both private networks after the fact:

```
# docker run -ti --name c3 -h c3 server1:5000/ubuntu
/# ip -4 addr li eth0
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    inet 172.17.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
```

```
++
```

```
# docker network connect net1 c3
# docker network connect net2 c3
# docker attach c3
# ip ro li
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link    src 172.17.0.2
172.18.0.0/16 dev eth1  proto kernel  scope link    src 172.18.0.3
172.19.0.0/16 dev eth2  proto kernel  scope link    src 172.19.0.3
```

- 17) Verify that you can connect to both c1 and c2 containers:

```
/# cat /etc/resolv.conf
search example.com
nameserver 127.0.0.11
options ndots:0
/# getent hosts c1
172.18.0.2      c1
/# ping -c 1 c1
PING c1 (172.18.0.2) 56(84) bytes of data.
64 bytes from c1.net1 (172.18.0.2): icmp_seq=1 ttl=64 time=0.036 ms

--- c1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.036/0.036/0.036/0.000 ms
/# ping -c 1 c2
```

- Note that this is NOT normal loopback, but a special DNS that Docker is running.

```
PING c2 (172.19.0.2) 56(84) bytes of data.  
64 bytes from c2.net2 (172.19.0.2): icmp_seq=1 ttl=64 time=0.054 ms
```

```
--- c2 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 0.054/0.054/0.054/0.000 ms
```

Containers in both private networks are reachable with container names resolved via DNS. Prior to 1.10 release, name resolution was accomplished via `/etc/hosts` file manipulation.

- 18) Try again to send packets between the first and second containers (routed through the third container):

```
# docker inspect c2 | grep IPA | grep [[:digit:]]  
    "IPAddress": "172.19.0.2",  
# docker exec c1 ping -c10 172.19.0.2  
PING 172.19.0.2 (172.19.0.2) 56(84) bytes of data.  
  
--- 172.19.0.2 ping statistics ---  
10 packets transmitted, 0 received, 100% packet loss, time 8999ms  
# docker inspect c1 | grep IPA | grep [[:digit:]]  
    "IPAddress": "172.18.0.2",  
# docker exec c2 ping -c10 172.18.0.2  
PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.  
  
--- 172.18.0.2 ping statistics ---  
10 packets transmitted, 0 received, 100% packet loss, time 8999ms
```

Even though a network path exists and the third container is configured to route packets, the packets are dropped because of the default Docker isolation Netfilter rules.

- 19) Find the Netfilter rules that are blocking the traffic and remove them to verify that the first two containers can indeed communicate by routing packets through the third:

```
# iptables --line-numbers -vL DOCKER-ISOLATION  
Chain DOCKER-ISOLATION (1 references)  
num pkts bytes target    prot opt in      out     source      destination  
1      0      0 DROP      all  --  docker0 br-1a8ba2305f8b  anywhere  anywhere  
2      0      0 DROP      all  --  br-1a8ba2305f8b docker0  anywhere  anywhere
```

```

3      10      840 DROP          all -- br-962b15f712d2 br-1a8ba2305f8b anywhere anywhere
4      10      840 DROP          all -- br-1a8ba2305f8b br-962b15f712d2 anywhere anywhere
. . . output omitted . . .

```

```
# iptables -D DOCKER-ISOLATION 3
```

```
# iptables -D DOCKER-ISOLATION 3
```

```
# docker exec c2 ping -c10 172.18.0.2
```

```
PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
```

```
64 bytes from 172.18.0.2: icmp_seq=1 ttl=63 time=0.094 ms
```

```
. . . snip . . .
```

```
64 bytes from 172.18.0.2: icmp_seq=9 ttl=63 time=0.073 ms
```

```
64 bytes from 172.18.0.2: icmp_seq=10 ttl=63 time=0.071 ms
```

```
--- 172.18.0.2 ping statistics ---
```

```
10 packets transmitted, 10 received, 0% packet loss, time 8999ms
```

```
rtt min/avg/max/mdev = 0.069/0.074/0.094/0.013 ms
```

- Use the rule number of the first rule in the pair identified by the matched traffic of 10 packets.
- Repeat the number since the deletion of the first rule caused the line numbers to shift up and the second rule is now the same number that the first was.

With the isolation rules deleted, the containers can now communicate.

Cleanup

20) Remove the containers and networks:

```
# docker rm -f c1 c2 c3
```

```
. . . output omitted . . .
```

```
# docker network rm net1
```

```
# docker network rm net2
```

Objectives

- ☞ Understand the use of the EXPOSE Dockerfile instruction
- ☞ Use the -P option to automatically map ports
- ☞ Manually map ports with the -p option
- ☞ Connect containers via links

Requirements

- 🖥 (1 station)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -
```

```
Password: makeitso 
```

- 2) Create a simple script to help query containers for their IP address:

```
# mkdir /root/bin
```

```
File: /root/bin/docker-ip
```

```
+ #!/bin/sh
```

```
+ exec docker inspect --format '{{ .NetworkSettings.IPAddress }}' "$@"
```

```
# chmod a+x /root/bin/docker-ip
```

EXPOSE Directive and -P Option

- 3) Create a new Dockerfile build directory and related files:

```
# rm -rf ~/build/
```

```
# mkdir ~/build
```

```
# cd ~/build
```

```
File: ~/build/Dockerfile
```

```
+ FROM server1:5000/nginx
```

```
+ COPY start.sh /
```

```
+ EXPOSE 80 443
```

```
+ CMD ["sh", "-c", "/start.sh"]
```

Lab 5

Task 2

Docker Ports and Links

Estimated Time: 30 minutes


```
File: ~/build/start.sh
```

```
+ #!/bin/sh
+ echo "$HOSTNAME website" > /usr/share/nginx/html/index.html
+ exec nginx -g 'daemon off;'
```

```
# chmod a+x start.sh
```

4) Build an image from the new definition:

```
# docker build -t nginx:v2 .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM server1:5000/nginx
---> eb4a127a1188
Step 2 : COPY start.sh /
---> 498b9b8a1a66
Removing intermediate container 679478c6c355
Step 3 : EXPOSE 80 443
---> Running in 41d176e13262
---> 50f15fa031da
Removing intermediate container 41d176e13262
Step 4 : CMD sh -c /start.sh
---> Running in 66a2ae689ac4
---> ebed4ac658f6
Removing intermediate container 66a2ae689ac4
Successfully built ebed4ac658f6
```

5) Launch a container and examine the effect of the EXPOSE directive:

```
# docker run -d --name web nginx:v2
ad2c18d36befcc70049c9b22cdf1013ecd52bf31595d8882bef3bbbbb0fa640
# docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	. . . snip . . .
ad2c18d36bef	nginx:v2	"sh -c /start.sh"	About a minute ago	Up About a minute	80/tcp, 443/tcp.	. . snip . . .

The EXPOSE'd ports are listed in the docker ps output as an indication of what ports the image might have services listening on.

6) Determine what ports the container processes are actually listening on, by examining the container from the inside with the ss command:

```
# docker exec web ss -taun
Netid  State      Recv-Q Send-Q      Local Address:Port      Peer Address:Port
tcp    LISTEN     0      128             *:80                      *:*
```

Notice that the service is not listening on port 443 (only port 80). Remember that the ports listed in the `docker ps` output are not an indication of which ports the container is actually using.

7) Try connecting to the listening service from the host system:

```
# curl $(docker-ip web):80
ad2c18d36bef website
# curl 127.0.0.1:80
curl: (7) Failed connect to 127.0.0.1:80; Connection refused
```

- Works because this IP is reachable from the host system via the Docker0 bridge.
- Fails because the service is only bound to IPs within the container network namespace.

Services within the containers are accessible from the host system by default. This is true even if the EXPOSE directive was not present.

8) Determine which ports are reachable to other external systems:

```
# docker port web
#
# iptables -t nat -L PREROUTING
Chain PREROUTING (policy ACCEPT)
target     prot opt source                destination
DOCKER     all  --  anywhere              anywhere             ADDRTYPE match dst-type LOCAL
# iptables -t nat -L DOCKER
Chain DOCKER (2 references)
target     prot opt source                destination
```

- No ports are currently mapped to ports on the host system.
- Packets entering the kernel destined to containers are sent to the DOCKER NAT chain for further processing.
- No DNAT rules exist, so the container service are not currently visible from external hosts.

9) Remove the container, and start a new one this time using the `-P` option:

```
# docker rm -f web
web
# docker run -dP --name web nginx:v2
9b93a1a55aed003e31ee583506636ef8bd6b2c2eb48c775bef29a6dd83e52a46
```

- 10) Again examine which internal container ports are mapped to host ports and subsequently reachable from outside the container:

```
# docker port web
443/tcp -> 0.0.0.0:32779
80/tcp -> 0.0.0.0:32780
# iptables -nt nat -L DOCKER
Chain DOCKER (2 references)
target      prot opt source                destination
DNAT        tcp  --  anywhere              anywhere             tcp dpt:32779 to:172.17.0.28:443
DNAT        tcp  --  anywhere              anywhere             tcp dpt:32780 to:172.17.0.28:80
```

- 11) Try to connect to the mapped ports and access the service within the container:

```
# curl $(docker-ip web):80
9b93ala55aed website
# docker port web | awk -F: '/^80/ {print $2}'
32780
# curl 127.0.0.1:32780
9b93ala55aed website
# docker rm -f web
web
```

• What host port maps to container port 80?

Manual Port Mapping with -p

- 12) Start a container with a dynamic port mapping and test:

```
# IP1=$(hostname -i)
# echo $IP1
10.100.0.X
# docker run -d -h web1 --name web1 -p 80 nginx:v2
554157c2df00c92b591beb12897ec66ee90728b08cd4c07cf8cfa5e5972fd15d
# docker port web1
80/tcp -> 0.0.0.0:32785
# curl 127.0.0.1:32785
web1 website
# curl $IP1:32785
web1 website
```

• Store main host IP in a variable for easy referencing.

• container port 80 mapped to random port on all host IPs

13) Start a container with a manually defined port mapping and test:

```
# docker run -d -h web2 --name web2 -p 8000:80 nginx:v2
2f5667b8364698b8f339cfff8bdec21cb6aef43d90c956bb92a32be69d341c3
# docker port web2
80/tcp -> 0.0.0.0:8000 • container port 80 mapped to port 8000 on all host IPs
# curl 127.0.0.1:8000
web2 website
# curl $IP1:8000
web2 website
```

14) Temporarily bind an additional IP to the host system's eth0 interface that is the same as its main IP, but with 100 added to the final octet:

```
# IP2="10.100.0.${hostname -i | awk -F. '{print $4+100}')" • Again store in variable for easy access.
# echo $IP2
10.100.0.10X
# ip addr add $IP2 dev eth0
# ip -4 addr show dev eth0 | grep inet
    inet 10.100.0.X/24 brd 10.100.0.255 scope global dynamic eth0
    inet 10.100.0.X+100/32 scope global eth0
```

15) Launch containers with mappings bound to specific IPs, but random ports:

```
# docker run -d -h web3 --name web3 -p $IP1::80 nginx:v2
869c12e47081f739613b89ae8c159c5008e579d686bc3583ea08d5e78c78223d
# docker port web3
80/tcp -> 10.100.0.X:32768 • container port 80 mapped to random port on specified host IP1
# curl $IP1:32768
web3 website
# docker run -d -h web4 --name web4 -p $IP2::80 nginx:v2
280133b8f87dd509909e537825227ece1924f931d8f28fba01957f94221bbb27
# docker port web4
80/tcp -> 10.100.0.10X:32770 • container port 80 mapped to random port on specified host IP2
# curl $IP2:32770
web4 website
```

- 16) Launch containers with mappings bound to specific IPs, and assigned ports:

```
# docker run -d -h web5 --name web5 -p $IP1:80:80 nginx:v2
de2248c8652d1745df010a3940f4aec63fd68e56b8969c8e87ce171f6bd80e51
# docker port web5
80/tcp -> 10.100.0.X:80
# curl $IP1:80
web5 website
# docker run -d -h web6 --name web6 -p $IP2:80:80 nginx:v2
fc4730490e2fd0254bde0c83843b51178ba72aff36dd6518b1105ee80f4b2a9d
# docker port web6
80/tcp -> 10.100.0.10X:80
# curl $IP2:80
web6 website
```

• container port 80 mapped to port 80 on specified host IP1

• container port 80 mapped to port 80 on specified host IP2

- 17) Examine the collective docker proxy processes listening for the six web containers and the corresponding forwarding rules in the DNAT tables:

```
# systemctl status -l docker
docker.service - Docker Application Container Engine
   Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Wed 2016-04-20 14:43:56 MDT; 1 day 2h ago
     Docs: https://docs.docker.com
    Main PID: 1024 (docker)
    CGroup: /system.slice/docker.service
            |- 1024 /usr/bin/docker daemon -H fd:// --registry-mirror http://server1:5000 --insecure-registry 10.100.0.0/2
            --storage-driver=devicemapper --storage-opt=dm.thinpooldev=/dev/mapper/vg0-dockerpool
            |-15653 docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 32770 -container-ip 172.17.0.2 -container-port 80
            |-15736 docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 8000 -container-ip 172.17.0.3 -container-port 80
            |-15867 docker-proxy -proto tcp -host-ip 10.100.0.1 -host-port 32768 -container-ip 172.17.0.4 -container-port 80
            |-15938 docker-proxy -proto tcp -host-ip 10.100.0.101 -host-port 32768 -container-ip 172.17.0.5 -container-port 80
            |-16027 docker-proxy -proto tcp -host-ip 10.100.0.1 -host-port 80 -container-ip 172.17.0.6 -container-port 80
            ^-16092 docker-proxy -proto tcp -host-ip 10.100.0.101 -host-port 80 -container-ip 172.17.0.7 -container-port 80

# lsof -Pi -a -c exe
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
exe      26337 root   4u    IPv6 3687458      0t0  TCP *:32785 (LISTEN)
exe      26458 root   4u    IPv6 3690544      0t0  TCP *:8000 (LISTEN)
exe      29928 root   4u    IPv4 3699954      0t0  TCP stationX.example.com:32769 (LISTEN)
exe      30045 root   4u    IPv4 3700689      0t0  TCP stationX+100.example.com:32768 (LISTEN)
exe      30177 root   4u    IPv4 3701508      0t0  TCP stationX.example.com:80 (LISTEN)
```

```

exe      30318 root      4u  IPv4 3702389      0t0  TCP stationX+100.example.com:80 (LISTEN)
# iptables -t nat -nL DOCKER
Chain DOCKER (2 references)
target     prot opt source                destination
DNAT      tcp  --  0.0.0.0/0               0.0.0.0/0          tcp dpt:32785 to:172.17.0.41:80
DNAT      tcp  --  0.0.0.0/0               0.0.0.0/0          tcp dpt:8000 to:172.17.0.42:80
DNAT      tcp  --  0.0.0.0/0               10.100.0.3          tcp dpt:32769 to:172.17.0.43:80
DNAT      tcp  --  0.0.0.0/0               10.100.0.103        tcp dpt:32768 to:172.17.0.44:80
DNAT      tcp  --  0.0.0.0/0               10.100.0.3          tcp dpt:80 to:172.17.0.45:80
DNAT      tcp  --  0.0.0.0/0               10.100.0.103        tcp dpt:80 to:172.17.0.46:80

```

18) Stop and remove all the web containers:

```

# for i in {1..6}; do docker rm -f web$i; done
web1
web2
. . . output omitted . . .

```

Connecting Containers with Links

19) Create a simple MySQL container:

```

# docker run --name db -e MYSQL_ROOT_PASSWORD=pass -d mysql
Unable to find image 'mysql:latest' locally
Trying to pull repository server1:5000/mysql ...
a5e5891111da: Download complete
104de4492b99: Download complete
. . . snip . . .
3796bafef56d: Download complete
Status: Downloaded newer image for server1:5000/mysql:latest
6f9cfd05c7a3abba42367496e3826dda658d2bb7825add28af7dcb6636bb364d
# docker tag server1:5000/mysql mysql

```

The script run by the default ENTRYPOINT does initial database setup and will use the MYSQL_ROOT_PASSWORD environment variable if it is present to define the database root user password.

20) Examine the environment within the db container:

```

# docker exec db env

```

```

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=3d46df778618
container_uuid=3d46df77-8618-d752-3c0c-95cc89a68997
MYSQL_ROOT_PASSWORD=pass
MYSQL_MAJOR=5.7
MYSQL_VERSION=5.7.12-1debian8
HOME=/root

```

- 21) Launch a temporary container that has a link to the MySQL db container, and examine the changes made by the link option:

```

# docker run --rm -ti --link db:db mysql bash
/# env | grep DB
DB_NAME=/sleepy_engelbart/db
DB_PORT=tcp://172.17.0.1:3306
DB_PORT_3306_TCP_PORT=3306
DB_PORT_3306_TCP_PROTO=tcp
DB_ENV_MYSQL_ROOT_PASSWORD=pass
DB_PORT_3306_TCP_ADDR=172.17.0.1
DB_PORT_3306_TCP=tcp://172.17.0.1:3306
DB_ENV_MYSQL_VERSION=5.7.12-1debian8
DB_ENV_MYSQL_MAJOR=5.7
/# grep db /etc/hosts
172.17.0.1      db 3d46df778618
/# ping -c1 db
PING db (172.17.0.1): 48 data bytes
56 bytes from 172.17.0.1: icmp_seq=0 ttl=64 time=0.067 ms
--- db ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.067/0.067/0.067/0.000 ms

```

Environment variables and a host entry allow applications to identify where the linked services from the db container are located.

- 22) Use the environment variables to provide the necessary info to connect to the MySQL server instance in the other container:

```

/# mysql -h${DB_PORT_3306_TCP_ADDR} -P${DB_PORT_3306_TCP_PORT} -uroot -p${DB_ENV_MYSQL_ROOT_PASSWORD}
. . . snip . . .
mysql> \s
-----

```

```
mysql Ver 14.14 Distrib 5.7.12, for Linux (x86_64) using EditLine wrapper
```

```
Connection id:          3
Current database:
Current user:           root@172.17.0.2
SSL:                    Not in use
Current pager:          stdout
Using outfile:           ''
Using delimiter:         ;
Server version:          5.7.12 MySQL Community Server (GPL)
Protocol version:        10
Connection:              172.17.0.9 via TCP/IP
Server characterset:     latin1
Db characterset:         latin1
Client characterset:     latin1
Conn. characterset:      latin1
TCP port:                3306
Uptime:                  2 min 52 sec
```

```
Threads: 1 Questions: 5 Slow queries: 0 Opens: 67 Flush tables: 1 Open tables: 60 Queries per second avg: 0.029
```

```
-----
```

```
mysql> \q
Bye
/# exit
#
```

23) Create another database server container:

```
# docker run --name db2 -e MYSQL_ROOT_PASSWORD=pass2 -d mysql
```

24) Launch another container that has links to both of the database containers:

```
# docker run -ti --name db-client --link db:db1 --link db2:db2 mysql /bin/bash
/# grep db /etc/hosts
172.17.0.1      db1 3d46df778618 db
172.17.0.3      db2 089d70b4f8c7
/# env | grep DB | sort
DB1_ENV_MYSQL_MAJOR=5.7
DB1_ENV_MYSQL_ROOT_PASSWORD=pass
```

- Note the use of an alias (db1) for the db container to make variable names more uniform.
- Host entries exist for both containers


```

DB1_ENV_MYSQL_VERSION=5.7.12-1debian8
DB1_NAME=/sad_mcclintock/db1
DB1_PORT=tcp://172.17.0.1:3306
DB1_PORT_3306_TCP=tcp://172.17.0.1:3306
DB1_PORT_3306_TCP_ADDR=172.17.0.1
DB1_PORT_3306_TCP_PORT=3306
DB1_PORT_3306_TCP_PROTO=tcp
DB2_ENV_MYSQL_MAJOR=5.7
DB2_ENV_MYSQL_ROOT_PASSWORD=pass2
DB2_ENV_MYSQL_VERSION=5.7.12-1debian8
DB2_NAME=/sad_mcclintock/db2
DB2_PORT=tcp://172.17.0.3:3306
DB2_PORT_3306_TCP=tcp://172.17.0.3:3306
DB2_PORT_3306_TCP_ADDR=172.17.0.3
DB2_PORT_3306_TCP_PORT=3306
DB2_PORT_3306_TCP_PROTO=tcp

```

```

Ctrl + p Ctrl + q

```

- Detach from the terminal, but leave the client container running.

- 25) Restart one of the database server container instances and compare the IP it has before and after the restart:

```

# docker-ip db
172.17.0.1
# docker stop db
db
# docker run -d --name sleeper server1:5000/busybox /bin/sleep 8000
. . . output omitted . . .
# docker start db
db
# docker-ip db
172.17.0.5

```

Remember that container IP addresses are assigned from a dynamic pool by default and subject to change. The new pluggable IP address management (IPAM) system introduced in version 1.9 gives control over this and allows different backend modules to use different schemes (including just assigning addresses statically).

- 26) Reconnect to the client container that was linked to the two database server containers and see if the new IP is recorded there:

```
# docker attach db-client
```

```
Enter
```

```
/# echo $DB1_PORT_3306_TCP_ADDR
```

```
172.17.0.1
```

• Old IP :(

```
/# grep db1 /etc/hosts
```

```
172.17.0.6      db1 3d46df778618 db
```

• New IP :)

```
/# exit
```

Whenever possible, use the hostnames, and not the link environment variables, to refer to linked hosts since they are updated dynamically if a linked container is restarted.

Cleanup

27) Remove the database client and server containers:

```
# docker rm -fv db db2 db-client sleeper
```

```
db
```

```
db2
```

```
db-client
```

```
sleeper
```

Lab 5

Task 3

Multi-host Networks

Estimated Time: 30 minutes

Objectives


- 🔗 Examine how overlay networks are built and function
- 🔗 Use the overlay driver to create Docker networks that span multiple hosts
- 🔗 Connect and disconnect containers to multi-host networks

Requirements

🖥️ (2 stations) 🖥️ (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

Configuring Docker Key/Value Store

- 2) Edit the docker daemon config options on your first node so that it uses the shared Consul server:

```
File: /etc/sysconfig/docker  
--registry-mirror http://server1:5000 \  
--insecure-registry 10.100.0.0/24 \  
--storage-driver=devicemapper \  
→ --storage-opt=dm.thinpooldev=/dev/mapper/vg0-dockerpool+ \  
+ --cluster-store=consul://server1.example.com:8500 \  
+ --cluster-advertise=eth0:2376'
```

- 3) Verify that the daemon restarts without error, and is using the defined cluster store:

```
# systemctl restart docker  
# systemctl -l status docker  
docker.service - Docker Application Container Engine  
  Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: disabled)  
  Active: active (running) since Fri 2016-04-08 11:47:03 MDT; 15s ago  
    Docs: https://docs.docker.com  
  Main PID: 13011 (docker)
```

```

CGroup: /system.slice/docker.service
    13011 /usr/bin/docker daemon -H fd:// --registry-mirror http://server1:5000 --
    --insecure-registry 10.100.0.0/24 --cluster-store=consul://server1.example.com:8500 --cluster-advertise=eth0:2376
... output omitted ...
# docker info 2>/dev/null | tail -n 2
Cluster store: consul://server1.example.com:8500
Cluster advertise: 10.100.0.X:2376

```

- 4) Configure your second node to also use the classroom Consul server by editing its config:

```

File: /root/.docker/machine/machines/stationY.example.com/config.json
... snip ...
  "HostOptions": {
    "Driver": "",
    "Memory": 0,
    "Disk": 0,
    "EngineOptions": {
      "ArbitraryFlags": {},
      "ArbitraryFlags": [
+         "cluster-store=consul://server1.example.com:8500",
+         "cluster-advertise=eth0:2376"
+       ],

```

- 5) Deploy the config to the remote docker host:

```

[node1]# docker-machine provision stationY.example.com
Waiting for SSH to be available...
Detecting the provisioner...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
[node1]# docker-machine ssh stationY.example.com
Last login: Tue Mar 29 17:23:56 2016 from stationX.example.com
[node2]# systemctl daemon-reload
[node2]# systemctl restart docker
[node2]# docker info 2>/dev/null | tail -n 2
Cluster store: consul://server1.example.com:8500
Cluster advertise: 10.100.0.X:2376

```

```
[node2]# exit
```

For most Docker Machine drivers, the docker-machine restart *host_name* (which reboots the host) would be used after updating the config. For the generic SSH driver used in the lab, this will not work.

Creating and Using Multi-host Networks

- 6) Create a multi-host network from your first node. Since the entire class is sharing a single Key/Value store, the network name must be unique within the classroom. Be sure to replace *X* and *Y* with the station numbers of your assigned nodes:

```
[node1]# docker network create -d overlay sX-sY
```

```
6f8a5d2bbe5abac56bc59325099b190c50141240baac5d5f9b908b08ea39c104
```

```
# docker network ls
```

NETWORK ID	NAME	DRIVER
697b1747f06b	bridge	bridge
49ee96048008	none	null
3dfc2fe3a2de	host	host
6f8a5d2bbe5a	sX-sY	bridge

```
[node1]# docker network inspect sX-sY
```

```
[
  {
    "Name": "sX-sY",
    "Id": "6f8a5d2bbe5abac56bc59325099b190c50141240baac5d5f9b908b08ea39c104",
    "Scope": "global",
    "Driver": "overlay",
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",
          "Gateway": "10.0.0.1/24"
        }
      ]
    },
    "Containers": {},
    "Options": {}
  }
]
```

• Note that the IP subnet allocation is automatic and uses a different default range than the local docker networks.

- 7) Launch a container attached to the overlay network:

```
[node1]# docker run -dti --net=s1-s2 --name=c1 server1:5000/busybox
fad3097a6c9a6266c86714225e46ba5546abf84c49071aa806b5e310fc5d65c6
```

- 8) Examine the network interfaces created within the new container:

```
# docker exec c1 ip -f inet addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
29: eth0@if30: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    inet 10.0.0.2/24 scope global eth0
        valid_lft forever preferred_lft forever
31: eth1@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    inet 172.18.0.2/16 scope global eth1
        valid_lft forever preferred_lft forever
```

The eth0 VETH peer interface is connected to a bridge running in a private namespace for the overlay network. That same namespace has the VXLAN interface.

The eth1 VETH peer interface is connected to the local docker_gwbridge providing direct access to external hosts.

eth0 has an MTU of 1450 to leave room for the VXLAN header.

- 9) Verify that the network is seen by your other node:

```
[node1]# eval $(docker-machine env stationY.example.com)
[node1]# docker network ls
NETWORK ID          NAME                DRIVER
6f8a5d2bbe5a        s1-s2              overlay
4a889bcfc061        none              null
42fbb76b38aa        host              host
5bb0381bdd90        bridge            bridge
```

- 10) Launch a container on your second node that is also connected to the overlay network, and examine its interfaces and addresses:

```
[node1]# docker run -dti --net=s1-s2 --name=c2 server1:5000/busybox
```

```
[node1]# docker exec c2 ip -f inet addr li
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    inet 10.0.0.3/24 scope global eth0
        valid_lft forever preferred_lft forever
9: eth1@if10: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    inet 172.18.0.2/16 scope global eth1
        valid_lft forever preferred_lft forever
```

- 11) Capture traffic on the standard VXLAN UDP port to show the encapsulated traffic flowing across the overlay network:

```
[node1]# tcpdump -nei eth0 udp port 4789 &
[1] 26390 
[node1]# docker exec c2 ping -c1 c1 &>/dev/null
18:07:22.069084 02:52:00:13:01:02 > 02:52:00:13:01:01, ethertype IPv4 (0x0800), length 148: 10.100.0.2.51734 →
    > 10.100.0.1.4789: VXLAN, flags [I] (0x08), vni 256
02:42:0a:00:00:03 > 02:42:0a:00:00:02, ethertype IPv4 (0x0800), length 98: 10.0.0.3 > 10.0.0.2: ICMP echo →
    request, id 10496, seq 0, length 64
18:07:22.069230 02:52:00:13:01:01 > 02:52:00:13:01:02, ethertype IPv4 (0x0800), length 148: 10.100.0.1.41434 →
    > 10.100.0.2.4789: VXLAN, flags [I] (0x08), vni 256
02:42:0a:00:00:02 > 02:42:0a:00:00:03, ethertype IPv4 (0x0800), length 98: 10.0.0.2 > 10.0.0.3: ICMP echo →
    reply, id 10496, seq 0, length 64
[node1]# kill %1
2 packets captured
2 packets received by filter
0 packets dropped by kernel

[1]+  Done                  tcpdump -nei eth0 udp port 4789
```

Note that a single ping (ICMP echo-request/echo-reply) is tunneled inside the VXLAN connection over UDP port 4789. The MAC/IP addresses of the "inner" frames/packets are those of the eth0 interfaces of the containers. The MAC/IP addresses of the "outer" frames/packets are those of the eth0 interfaces of the nodes.

- 12) Capture some of the Serf Gossip protocol traffic:

```
[node1]# tcpdump -c 15 -nei eth0 port 7946
```

```

1:28:36.823114 02:52:00:13:01:03 > 02:52:00:13:01:01, ethertype IPv4 (0x0800), length 81: 10.100.0.3.7946 →
> 10.100.0.1.7946: UDP, length 39
11:28:36.823257 02:52:00:13:01:01 > 02:52:00:13:01:03, ethertype IPv4 (0x0800), length 55: 10.100.0.1.7946 →
> 10.100.0.3.7946: UDP, length 13
11:28:37.517141 02:52:00:13:01:01 > 02:52:00:13:01:03, ethertype IPv4 (0x0800), length 81: 10.100.0.1.7946 →
> 10.100.0.3.7946: UDP, length 39
. . . snip . . .
15 packets captured
15 packets received by filter
0 packets dropped by kernel

```

Serf over TCP/UDP 7946 is used to propagate the MAC address info needed to populate the VXLAN bridge forwarding tables (to permit unicast frames) and is an alternative to using traditional multicast routing and multicast frames to carry VXLANs.

Cleanup

13) Remove the containers and overlay network:

```

[node1]# docker rm -f c2
c2
[node1]# unset ${!DOCKER_*}
[node1]# docker rm -f c1
c1
[node1]# docker network rm sX-sY
[node1]# docker network ls

```

NETWORK ID	NAME	DRIVER
697b1747f06b	bridge	bridge
49ee96048008	none	null
3dfc2fe3a2de	host	host
52b52168c13d	docker_gwbridge	bridge

```

. . . output omitted . . .

```

- Depending on the progress of other students, you may still see their overlay networks listed, but your network should be gone.

Content

Volume Concepts	2
Creating and Using Volumes	3
Managing Volumes (cont.)	4
Changing Data in Volumes	5
Removing Volumes	6
Backing up Volumes	7
SELinux Considerations	8
Mapping Devices	9
Lab Tasks	10
1. Docker Volumes	11

Chapter

6

DOCKER VOLUMES

Volume Concepts

Bypasses union file system

- mounts directory from host within container

Decouples data life cycle from container life cycle

Share data between containers

Volume Concepts

As previously explored, the filesystem image presented to a container consists of one or more layers that are combined together using a union file system driver. The combination of these layers is mounted read-only and then an additional write layer is mounted on top of that to hold changes. When a file within a lower layer is modified, a copy is made within the top read-write layer and then changed. Unless committed, the top layer is discarded when the container is terminated.

Volumes provide a way to bypass the normal union file system and attach a data directory from the host system. This has several advantages as it decouples the life cycle of the data volume from the life cycle of the container. Best practice for containers is to design them so that they can be easily discarded and new containers launched (ex. when upgrading a software component within the container). Volumes facilitate:

- ⌘ Persisting data independent of the container life cycle.
- ⌘ A convenient way to share data between containers (since a volume can be mounted by multiple containers simultaneously).
- ⌘ Exposing the data to the host system (ex. to allow easy editing of the data from the host).

Creating and Using Volumes

```
-v /container_dir
  • VOLUME Dockerfile instruction
  • Data in /var/lib/docker/volumes/vol_hash_id/_data
  • Meta-data linking container to volume:
    /var/lib/docker/containers/container_id/config.json
-v /host_{dir,file}:/container_{dir,file}
  • bind mounts the host directory or file inside the container
Using a data volume container
  • --volumes-from=container
:ro or :rw
```

Creating an Internal Volume

Containers that need to potentially persist data after their own deletion can store that data on an internal Docker volume. These volumes are created at the same time a container is created by using the **docker run -v /container_dir** option. The volume will be assigned a digest name and a reference to that name is added to the container's config. A directory is created under `/var/lib/docker/volumes/vol_hash_id` to store the data for the volume.

The following example shows creating an internal volume (mounted at `/data` within the container) and examining the associated backing files/directory:

```
# docker run -d -v /data db
# docker inspect -f '{{.Volumes}}' db
map[/var/lib/mysql:/var/lib/docker/volumes/122c...5806/_data]
# ls /var/lib/docker/volumes/122c...5806/_data
auto.cnf ibdata1 ib_logfile0 ib_logfile1 mysql performance_schema
```

Another way to create an internal Docker volume is through the use of the **VOLUME** Dockerfile instruction.

Creating an External Volume

A file or directory from the host system can also be bind mounted into the container. If the container is running on the local host system, this makes it easy to see and edit the data from outside the container(s) using the volume. A common use case would be mounting a local `/code` directory into the container so that the application's code files can be edited from the host, and then tested by connecting to the service running within the container that mounts that volume.

The following examples show creating an external volume:

```
$ docker run -d -v /home/bcroft/app1/code:/code_
python /code/app.py
```

When volumes are first created, any files at the specified mount point are copied into the new volume.

Managing Volumes (cont.)

`docker volume {create,rm}`

- Added in 1.9 release

`docker volume {inspect,ls}`

- Added in 1.10 release

Supports growing list of volume drivers, many which allow volume to span multiple nodes

Docker Volume Command

Starting with the 1.9 release, the Docker engine now supports a new volume sub-command. This allows named volumes to be created and managed without requiring the volumes to be tied to a container. Subsequently, the older method of creating "data volume containers" is no longer considered best practice. The following example shows creating a named volume and then attaching a container to it later. Note that this volume will persist even if all containers referencing it are removed with the `-v` option:

```
# docker volume create --name datavol1
datavol1
# docker volume inspect datavol1
[
  {
    "Name": "datavol1",
    "Driver": "local",
    "Mountpoint": "/var/lib/docker/volumes/datavol1/_data"
  }
]
# docker run -v datavol1:/data -ti --rm busybox
/ # echo "example data" > /data/file
/ # exit
# cat /var/lib/docker/volumes/datavol1/_data/file
example data
```

Volumes can be listed with optional filters, and removed. In the following example, the first three volumes are in use by containers

and the forth is an orphaned volume no longer in use:

```
# docker volume ls
DRIVER          VOLUME NAME
local           datavol3
local           datavol4
local           datavol1
local           datavol2
# docker volume rm datavol1
Error response from daemon: Conflict: remove datavol1: volume is in use by container
# docker volume ls -f dangling=true -q
datavol4
# docker volume rm datavol4
datavol4
```

Changing Data in Volumes

Edit data within container

- Disadvantage: requires editor within container

Edit on host via `/var/lib/docker/volumes/volume_{name,id}/*`

- Disadvantage: difficult to identify directory
- May not have access to host system

Use `--volumes-from` with "editor container"

Editing Files Within Volumes

There are three different ways to edit files contained within volumes:

1. Run an editor within the container that has the volumes. This is the most obvious solution but has the disadvantage of requiring editing commands to be installed within the container.
2. Run an editor on the host system. This solution works extremely well for external volumes that map to a known host directory. For internal volumes, the name of the directory will be the digest, and Docker does not currently make it easy to match this with containers. Also, you may not have access to the Docker host where the containers are running.
3. Run an editor in a new container and use the `--volumes-from` option to make the volumes visible in the "editor container". This avoids the issue of identifying the correct volume directory on the host, and also allows the other container to remain focused on the service it provides (and not have editors installed).

Removing Volumes

`docker rm -v container`

- removes volume if no other images reference it

`docker volume rm volume_{name,id}`

Removing Volumes

When a container is removed, all volumes associated with the container can be deleted by using the `-v` option as shown in the following example:

```
$ docker run -v /root/code1:/appl -v /root/code2:/app2 --name code_vol ubuntu echo "Data container for code"
$ ls /var/lib/docker/volumes/
10b42870c37cf6bb53961ffa369028c83902cbf561137877a9886276dc840291
e68bfb7b2ad5193f78e72e3b33aa07a6dd0fdcael8cd4dc33cd61b28c5928873
$ docker run --name appl -d --volumes-from code_vol ubuntu /appl/main
$ docker rm -fv appl #running container deleted, but volumes remain in spite of -v
$ ls /var/lib/docker/volumes/
10b42870c37cf6bb53961ffa369028c83902cbf561137877a9886276dc840291
e68bfb7b2ad5193f78e72e3b33aa07a6dd0fdcael8cd4dc33cd61b28c5928873
$ docker rm -v code_vol #final reference to volumes removed with -v option, so volumes removed
$ ls /var/lib/docker/volumes/
```

Orphaned Volumes

Since one of the primary purposes of volumes is the persistent storage of data, as a precaution they are never removed by default. This can easily result in orphaned volumes. Discover these using the `docker volume -f dangling=true` option as previously shown.

Backing up Volumes

docker run --volumes-from

- Allows use of full featured container to perform backup

docker cp container:/src_file /dst_file

```
$ docker cp web1 /etc - > web1_etc.tar
```

Backup of Container Volumes

The best practice for backing up container volumes is to use the **--volumes-from** option and attach the volumes to a temporary container that has the backup tools you need. For example, to backup a Postgres database which stores data in `/var/lib/postgresql/data` and is running in container `db1` you could run the following:

```
$ docker inspect postgres
```

```
... snip ...
  "Volumes": {
    "/var/lib/postgresql/data": {}
  },
```

```
... output omitted ...
```

```
$ docker run --rm --volumes-from db1 -v $(pwd):/backup debian tar cvf /backup/db1_volume.tar /var/lib/postgresql/data
```

Copying Files Out of Container

Sometimes only a few files are needed from the container. In this case, the **docker cp** command can be a good option. One disadvantage is that this only allows for copying of files out of the container (not into the container). The following example show copying the nginx config out of the container named `web1`:

```
$ docker cp web1:/etc/nginx/nginx.conf .
```

The **docker cp** can also be used to create tar archives from a specified directory within a container (started, or stopped). For example, the following would create an archive of the entire `/etc` directory:

SELinux Considerations

Volume :Z option

- relabel volume with container specific sVirt label

Volume :z option

- relabel volume with shared sVirt label

`--security-opt=label:level:level`

- label container process with specified type

`--security-opt=label:disable`

- run container in the unconfined domain

Labeling Volumes on Container Start

Depending on the current labels assigned on the host system, external volumes may not be readable, and almost certainly would not be writable by default. To guarantee access within the container, Docker can relabel the files to `svirt_sandbox_file_t` plus the dynamically assigned container MCS label. This is done by appending the **:Z** suffix to the volume option for example:

```
$ ls -Z /home/bcroft/dv011
unconfined_u:object_r:user_home_t:s0 /dv011
$ docker run -ti -v /home/bcroft/dv011:/data1:Z centos
/$ ls -Zd /data1
system_u:object_r:svirt_sandbox_file_t:s0:c45,c608 /data1
```

The security label is not reset back to the original value when the container stops. Be aware that the new label may cause access problems for processes running on the host system.

Shared Volumes and SELinux MCS

If a volume must be shared between multiple containers, the MCS label can either be manually set to a common value via the `--security-opt=label:level:` option, or the `-z` option can be used to set the label to `s0` (which is read/write to all containers).

Disable SELinux for a Container

A container can be run in the `unconfined_t` domain by launching it with the `--security-opt=label:disable` option.

Mapping Devices

Don't map devices as volumes on SELinux enabled hosts

- normal host labels will block access to device within container
- using **-Z** to relabel will block access to device from host

`--device=host_devname:container_devname`

- does not use bind mounting, creates separate device file
- host device untouched, internal device labeled correctly
- may require additional capabilities within container

Making Host Devices Visible Within a Container

There are times when it is desirable to make a host device visible within a container. It is possible to use the volume mapping function to map individual files (including device files). However, doing this on an SELinux enabled host will either break host or container access to the device (due to the different label types required by each).

Instead of using volumes which simply result in the same file being visible within the container via a bind mount, a different file is needed. The `--device=` run option directs Docker to create the specified device within the container; for example:

```
# docker run --rm -ti --device=/dev/sda:/dev/sdb ubuntu
/# ls -Z /dev/sdb
system_u:object_r:svirt_sandbox_file_t:s0:c443,c781 /dev/sdb
/# fdisk -l /dev/sdb
. . . snip . . .
  Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *          2048     1026047     512000    83   Linux
/dev/sdb2           1026048     72706047    35840000    8e   Linux LVM
```

When mapping a device, an additional suffix can be appended to indicate what permissions are granted to the mapped device.

r ⇒ grants read access to the device
w ⇒ grants write access to the device
m ⇒ grants the ability to mknod that device

Lab 6

Estimated Time: 30 minutes

Task 1: Docker Volumes

Page: 6-11 Time: 30 minutes

Requirements: 🖥️ (1 station)

Lab 6

Task 1

Docker Volumes

Estimated Time: 30 minutes

Objectives


- ☞ Run a container with a persistent internal data volume attached
- ☞ Attach a host directory to a container as a volume
- ☞ Attach a host file to a container as a volume
- ☞ Create and use a data-volume container
- ☞ Map a host device into a container

Requirements

- 🖥 (1 station)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Remove any existing containers, images, and volumes:

```
# docker rm -f $(docker ps -aq)  
. . . output omitted . . .  
# docker rmi -f $(docker images -q)  
. . . output omitted . . .  
# docker volume rm $(docker volume ls -q)
```

- 3) Download and inspect the mysql image to see what internal volumes it defines:

```
# docker pull server1:5000/mysql  
Trying to pull repository server1:5000/mysql ...  
319d2015d149: Download complete  
64e5325c0d9d: Download complete  
. . . snip . . .  
107c338c1d31: Download complete  
Status: Downloaded newer image for server1:5000/mysql:latest  
# docker tag server1:5000/mysql mysql  
# docker inspect -f '{{ .Config.Volumes }}' mysql  
map[/var/lib/mysql:{}]
```

When containers are created from this image, Docker will create an internal volume that will be mounted at `/var/lib/mysql/` within the container.

- 4) Start a container and verify that the volume required by the image config is created by the Docker daemon:

```
# docker run --name db -d -e MYSQL_ALLOW_EMPTY_PASSWORD=yes mysql
... output omitted ...
# docker volume ls
DRIVER          VOLUME NAME
local          7448534abed1aa122088c5d9c1943c5f74c6906c4e38c30638f17f9a09b23864
```

- 5) Identify the directory storing that volume and list its contents:

```
# docker inspect db | grep -A10 Mounts
  "Mounts": [
    {
      "Name": "7448534abed1aa122088c5d9c1943c5f74c6906c4e38c30638f17f9a09b23864",
      "Source": "/var/lib/docker/volumes/7448534abed1aa122088c5d9c1943c5f74c6906c4e38c30638f17f9a09b23864/_data",
      "Destination": "/var/lib/mysql",
      "Driver": "local",
      "Mode": "",
      "RW": true,
      "Propagation": ""
    }
  ],
# ls -l /var/lib/docker/volumes/7448534abed1aa122088c5d9c1943c5f74c6906c4e38c30638f17f9a09b23864/_data/
total 188452
-rw-r-----. 1 systemd-bus-proxy ssh_keys      56 Apr 18 14:17 auto.cnf
-rw-r-----. 1 systemd-bus-proxy ssh_keys    1325 Apr 18 14:17 ib_buffer_pool
-rw-r-----. 1 systemd-bus-proxy ssh_keys 79691776 Apr 18 14:18 ibdata1
... output omitted ...
```

- 6) Remove the container and its associated volumes:

```
# docker rm -fv db
db
```

If the container is removed without specifying the `-v` option then its volumes are orphaned. They exist, but no current Docker command can display or remove them. Additionally, when removing a container, there is currently no Docker command to remove specific volumes but leave others.

- 7) Create a container with a persistent external data volume:

```
# docker run --name data1 -ti -v /data server1:5000/busybox
Unable to find image 'busybox:latest' locally
Trying to pull repository server1:5000/busybox ...
8c2e06607696: Download complete
cf2616975b4a: Download complete
6ce2e90b0bc7: Download complete
Status: Downloaded newer image for server1:5000/busybox:latest
/# df /data
Filesystem            1K-blocks      Used Available Use% Mounted on
/dev/mapper/vg0-var    5232640    262316   4970324   5% /data
/# echo "persistent data" > /data/file
/# exit
```

- 8) Restart the container and verify that the data in the volume is persistent across container runs:

```
# docker start -i data1
/# ls /data
file
/# cat /data/file
persistent data
/# exit
```

- 9) Access the file contained in the data volume directly from the host system:

```
# cat /var/lib/docker/volumes/*/_data/file
persistent data
```

- 10) Start another container that uses the volumes defined by the data1 container:

```
# docker run --name data2 --volumes-from data1 server1:5000/busybox cat /data/file
persistent data
```

- 11) Remove the data* containers and associated volumes:

```
# docker rm -v data1
data1
# docker start -i data2
persistent data
# docker rm -v data2
data2
# ls /var/lib/docker/volumes/
# docker volume ls
DRIVER          VOLUME NAME
```

- Docker doesn't remove the volume even though the `-v` option is used because another container references the volume.
- When the last container referencing a volume is removed, the volume is removed if the `-v` option is used, otherwise the volume becomes orphaned.

Mounting Host Directories as Volumes

- 12) Create a new project directory and related files to host a simple web application:

```
# mkdir -p ~/webapp/content
# cd ~/webapp
# echo "container web content" > content/index.html
```

- 13) Start a container that mounts the `content/` directory as the document root for the web server and access the content:

```
# docker run --name web1 -dP -v /root/webapp/content:/usr/share/nginx/html server1:5000/nginx
# curl $(docker-ip web1)
container web content
```

Although SELinux is enabled, the docker engine is not currently configured to run in a protected domain, and so the content is accessible to the container even though the SELinux types are incorrect.

- 14) Configure Docker to use SELinux enforcing on your first assigned node by adding the following daemon option to the existing config:

File: /etc/sysconfig/docker

```
... snip ...
--cluster-store=consul://server1.example.com:8500 \
→ --cluster-advertise=eth0:2376 \
+ --selinux-enabled'
```

```
[node1]# systemctl restart docker
```

- 15) Restart the container and test whether it can access the content with SELinux now enabled:

```
# docker start web1
web1
# curl $(docker-ip web1)
container web content
# docker inspect web1 | egrep '(Mount|Process)Label'
    "MountLabel": "",
    "ProcessLabel": "",
# docker stop web1
web1
```

The existing container can still access the content because it is not running in a protected SELinux domain (note the empty ProcessLabel). While running unconfined, the SELinux checks don't prevent access.

- 16) Start a new container this time running as a protected sVirt type, and attempt to access the content (will fail):

```
# docker run --name web2 -dP -v /root/webapp/content:/usr/share/nginx/html server1:5000/nginx
f4b1cc9752159768490246e9f6219c3f16503b62757c2e64ba6da1ff86f8a95e
# docker inspect web2 | egrep '(Mount|Process)Label'
    "MountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c84,c933",
    "ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c84,c933",
# ls -lZ ~/webapp/content/
-rw-r--r--. root root unconfined_u:object_r:admin_home_t:s0 index.html
# curl $(docker-ip web2)
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.9.3</center>
</body>
</html>
# docker stop web2
web2
```

The denial is caused by SELinux label mismatches as expected.

- 17) Start a new container requesting the volume content to be relabeled for exclusive access by this container, and test access:

```
# docker run --name web3 -dp -v /root/webapp/content:/usr/share/nginx/html:Z server1:5000/nginx
3a76da1274457e3b7121c2788d56d94f3c8768385ae2786afedb22cbadf76c45
# docker inspect web3 | egrep '(Mount|Process)Label'
    "MountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c360,c613",
    "ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c360,c613",
# ls -lZ ~/webapp/content/
-rw-r--r--. root root system_u:object_r:svirt_sandbox_file_t:s0:c360,c613 index.html
# curl $(docker-ip web3)
container web content
# docker stop web3
```

The files within the volume are properly labeled because of the :Z prefix used when declaring the volume.

- 18) Create two new containers that use a shared sVirt label to share access to SELinux protected volume content:

```
# docker run --name web4 -dp 8000:80 -v /root/webapp/content:/usr/share/nginx/html:z server1:5000/nginx
2f0734ad2bd72551c7e83bd033dbc031403b11eb93b8c661d585bd848445f75c
# docker run --name web5 -dp 8001:80 -v /root/webapp/content:/usr/share/nginx/html:ro,z server1:5000/nginx
49a71c0919341232503f47e65eeddeled2558183e3025b7ce8480fbc0d3bd632
# ls -Z content/
-rw-r--r--. root root system_u:object_r:svirt_sandbox_file_t:s0 index.html
# curl localhost:8000
container web content
# curl localhost:8001
container web content
```

Both containers can access the content because it was labeled without restricting sVirt categories. The content is still protected from other apps due to the svirt_sandbox_file_t type label.

- 19) Try creating content from with the containers to test the operation of mounting the volume RW vs. RO:

```
# docker exec web4 bin/sh -c "echo web4 > /usr/share/nginx/html/web4"
# docker exec web5 bin/sh -c "echo web5 > /usr/share/nginx/html/two"
bin/sh: 1: cannot create /usr/share/nginx/html/two: Read-only file system
```


20) Compare the config info related to the bind mounted volumes from the host:

```
# for i in web{1..5}; do docker inspect -f '{{json .Mounts }}' $i; done
[{"Source":"/root/webapp/content","Destination":"/usr/share/nginx/html","Mode":"","RW":true,"Propagation":"rprivate"}]
[{"Source":"/root/webapp/content","Destination":"/usr/share/nginx/html","Mode":"","RW":true,"Propagation":"rprivate"}]
[{"Source":"/root/webapp/content","Destination":"/usr/share/nginx/html","Mode":"Z","RW":true,"Propagation":"rprivate"}]
[{"Source":"/root/webapp/content","Destination":"/usr/share/nginx/html","Mode":"z","RW":true,"Propagation":"rprivate"}]
[{"Source":"/root/webapp/content","Destination":"/usr/share/nginx/html","Mode":"ro,z","RW":false,"Propagation":"rprivate"}]
```

21) Remove the containers:

```
# docker rm -fv web{1..5}
web1
. . . output omitted . . .
# ls ~/webapp/content

index.html web4
# cd; rm -rf ~/webapp/content
```

- Volumes that map to a directory on the host system are not removed even when the `-v` option is used.

Mounting Individual Files as Volumes

22) Launch a container that has an individual file mounted from the host system:

```
# touch ~/container_history
# docker run -ti --rm -v /root/container_history:/root/.bash_history:Z
server1:5000/ubuntu /bin/bash
/# ls -l
. . . output omitted . . .
/# ps -ef
. . . output omitted . . .
/# exit
```

- File must exist or Docker will create a directory by that name

23) Verify the command history was recorded to the persistent file on the host:

```
# cat ~/container_history
ls -l
ps -ef
exit
```

- 24) Launch a container that runs unconfined and logs to the logging socket of the host system:

```
# docker run --rm -ti --security-opt=label:disable -v /dev/log:/dev/log server1:5000/ubuntu
/# ls -lZ /dev/log
srw-rw-rw-. 1 root root system_u:object_r:devlog_t:s0:c0 0 Aug 19 21:36 /dev/log
/# logger "Test message from within container"
/# exit
exit
# journalctl -rn1 _COMM=logger
-- Logs begin at Wed 2015-08-19 15:36:05 MDT, end at Fri 2015-08-21 12:21:20 MDT. --
Aug 21 12:21:08 stationX.example.com logger[1577]: Test message from within container
```

Bonus: Dealing with Device Files

- 25) Map the host primary disk device into a container as a read only device:

```
# docker run --rm -ti --device=/dev/sda:/dev/sdb:r server1:5000/ubuntu
/# ls -l /dev/sdb
brw-rw----. 1 root disk 8, 0 Aug 21 18:26 /dev/sdb
/# fdisk /dev/sdb
You will not be able to write the partition table.
Command (m for help): p
. . . snip . . .
  Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *          2048      1026047       512000   83   Linux
/dev/sdb2            1026048      72706047      35840000   8e   Linux LVM
Command (m for help): q
/# exit
```

• Device file shows read/write permissions.

• fdisk program correctly detects and reports that writes are actually blocked.

- 26) Map the host primary disk device into a container this time read/write:

```
# docker run --rm -ti --device=/dev/sda:/dev/sdb:rw server1:5000/ubuntu
/# ls -l /dev/sdb
brw-rw----. 1 root disk 8, 0 Aug 21 18:26 /dev/sdb
/# fdisk /dev/sdb
Command (m for help): q
/# exit
```

• Device file shows read/write permissions.

• fdisk program does not display warning that writes are blocked.

Content

Concepts	2
Compose CLI	3
Defining a Service Set	4
Docker Swarm	5
Lab Tasks	7
1. Docker Compose	8
2. Docker Swarm	19

Chapter

7

DOCKER COMPOSE/SWARM

Managing Sets of Containers

Simple applications often reside within a single container. As applications become more complex, or need to be scaled to handle more load, they generally span a set of interconnected containers. Controlling a set of containers can become difficult with the basic Docker commands. The Docker Compose tool allows you to describe a set of containers and their options in a succinct way and provides commands to manage the group of containers together.

docker-compose.yml

The description of the set of containers used by Docker Compose is contained in a simple text file using YAML syntax. Most of the lines in this file map to options used by the **docker run** command. Structural hierarchy of data in YAML is largely defined by indentation, so care must be taken to properly align data within the file. Often, the file will make reference to other files (such as Dockerfile(s) needed to build images. Best practice is to group all referenced files into a single parent directory.

The docker-compose Command

Once a set of containers is properly described, the **docker-compose** commands can be used to build, start, stop, and otherwise manage the set of containers. For example, the following compares starting two containers using **docker-compose** versus the equivalent **docker** commands:

Concepts

Define and run multi-container applications

- currently all containers on a single host

docker-compose.yml → **describes a set of containers**

docker-compose build → **build set**

docker-compose up → **start set**

File: docker-compose.yml

```
web:
  build: .
  ports:
    - "80:80"
  links:
    - db
db:
  image: postgres
```

\$ **docker-compose up**

Is roughly equivalent to:

\$ **docker run --name db -i postgres**

\$ **docker build . -t web**

\$ **docker run --name web -i -p 80:80 --link db:db web**

Compose CLI

docker-compose *command*

- operates on set of containers
- looks for `./docker-compose.yml`

<https://docs.docker.com/compose/cli/>

Compose CLI

The **docker-compose** commands are designed to be similar to the **docker** commands. The difference being that they operate on a set of containers instead of a single container. Compose is a relatively new part of the Docker ecosystem and is still evolving rapidly. The reference documentation for **docker-compose** is found at <https://docs.docker.com/compose/cli/>.

When running **docker-compose**, it looks in the current directory for a `docker-compose.yml` to determine what containers to operate on, so current working directory is essential. Current compose CLI commands include:

build ⇒ build image(s) from the specified Dockerfile.

kill ⇒ send sigkill to processes in set of containers. A more graceful termination is done by pressing `Ctrl+C` from the terminal attached to the set of containers.

logs ⇒ prints aggregated log output from set of containers. This is identical to the output seen if the set was started in the foreground.

port ⇒ shows mapped ports for set of containers.

ps ⇒ container state for each member of set.

pull ⇒ pre-pull images needed by each member of set.

restart ⇒ restart services.

rm ⇒ remove stopped set of containers.

run ⇒ run a command within the specified service's container.

Similar to the **docker exec** command.

scale ⇒ launch the specified number of instances for the listed

service.

start ⇒ start set without recreating them.

stop ⇒ stop set of containers without removing them.

up ⇒ remove and recreate set of containers. Normally only used when first starting a set of containers. When used with the **-d** option, it will daemonize (detach the service from the terminal).

Defining a Service Set

docker-compose.yml

- <https://docs.docker.com/compose/yml/>

Requires either image or build

Remaining keys map to docker run options

docker-compose.yml

The heart of Docker Compose is the YAML formatted file that defines the set of services to launch. This is looked for in the current directory, and relative references to other files are relative to the location of the file. The only required key is either **image** which specifies the name of the image to use for the container, or **build** which specifies the location of the Dockerfile to build an image from. The remaining keys all correspond to options to the **docker run** command.

The following table shows the relationship between a few common keys used in a Compose YAML file, and the corresponding **docker run** options:

docker-compose key	docker run option
image:	<i>IMAGE[:TAG @DIGEST]</i>
command:	<i>[COMMAND] [ARG...]</i>
entrypoint:	<i>--entrypoint=</i>
links:	<i>--link</i>
ports:	<i>-p</i>
expose:	<i>--expose=</i>
environment:	<i>--env=</i>
volumes:	<i>-v</i>

In addition to the keys that map to **docker run** options, Compose also has a few keys that are unique to its operation. For example:

external_links: ⇒ defines a link to a container outside of the set described by the local YAML file.

dockerfile ⇒ an alternate name for the file used to build that service. Allows a collection of Dockerfile's to share the same directory.

extends: ⇒ an alternative to **image:** or **build:**, this essentially provides a reference to another YAML file and service name and then extends that definition.

labels: ⇒ equivalent to the LABEL directive in a Dockerfile. Allows adding arbitrary metadata layers to the image used by that service.

env_file: ⇒ adds environment variables from a file.

Docker Swarm

Clustering of Docker hosts

- based on Docker API

Needs key/value store for operation: Consul, Etcd, Zookeeper, Docker Hub

Includes basic scheduler: spread, binpack, random

Filters implement constraints (label driven)

Can run in HA configuration: primary + secondary manager(s)

Clustering Hosts via Docker Swarm

Managing a group of Docker hosts brings additional challenges related to scheduling (where a container should run), orchestration (controlling the full life-cycle of the container), and service discovery (tracking where the containers are located and providing a method to connect to them). Several excellent third party solutions exist such as Kubernetes, Mesos, and Core-OS Fleet. Docker Swarm is the Docker native solution that essentially acts as a Docker API compatible proxy to a collection of Docker hosts.

To track the nodes, containers, networks, volumes, etc. within a swarm, Docker Swarm needs access to a key/value store. Currently there are 4 methods supported: Consul, Etcd, Zookeeper, and the Docker Hub. It is recommended that the selected service be configured in a highly available fashion (consult respective docs for details). Docker Hub can be used for small clusters (not designed for production use). When using the Docker Hub, a token must be generated (when the cluster is created), and then specified when each node joins the cluster for example:

```
[manager]# SWARM_TOKEN="$(docker run --rm swarm create)"
[manager]# echo $SWARM_TOKEN
1c9aa4430018723150b2da9f241fe1b5
[nodeX..Y]# docker run -d swarm join --advertise=10.100.0.1:2376 token://$SWARM_TOKEN

[client]# docker run -dp 4000:2375 swarm manage token://$SWARM_TOKEN
[client]# docker -H localhost:4000 docker_command
```

Swarm Container Scheduling

When a client requests a container run on the swarm, the manager process must decide which node within the swarm will host the container. The first thing considered when selecting a node is the strategy which can be one of the following three types:

spread ⇒ place new container on node with the fewest current containers

binpack ⇒ place new container on the node with the most current containers that still reports available RAM sufficient to run the new container

random ⇒ as per name

The second factor involved in scheduling containers is the use of filters which provide further constraints on the list of nodes the container is allowed to run on. Current filters include:

Node filters ⇒ Attributes of the node in the form of arbitrary labels (specified when launching the Docker daemon for that node). Some labels are defined automatically such as: node (name or ID of node), storagedriver, executiondriver, kernelversion, and operatingsystem.

Container affinity filters ⇒ Constrain new container to a node: running a particular container, which has the specified image, or has a container with a specific label.

Container dependency filters ⇒ Constrain new container to a node which has a container matching the specified: `--volumes-from`, `--link`, or `--net=container` run option.

Port filter ⇒ New container can only run on a node where the requested exposed port is not already taken by another container.

Filter expressions are specified at container run time using the `-e` option as if they were environment variables. For example: **`docker run -d -e affinity:image==baseapp app1`**. The general form for filters is: `[filter_type]:[filter_name][operator][filter_value]`.

Lab 7

Estimated Time: 80 minutes

Task 1: Docker Compose

Page: 7-8 Time: 60 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Task 2: Docker Swarm

Page: 7-19 Time: 20 minutes

Requirements: 🖥️ (1 station) 🖨️ (classroom server)

Objectives


- 🔗 Create a .yaml config for Docker Compose that launches multiple connected containers
- 🔗 Build and run a set of containers using docker-compose
- 🔗 Use docker-compose commands to manage a set of containers

Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -  
Password: makeitso 
```

- 2) Download and inspect the Redis image

```
# docker pull server1:5000/redis  
. . . output omitted . . .  
# docker tag server1:5000/redis redis  
# docker inspect -f '{{ .Config.Volumes }}' redis  
map[/data] ————— • Internal volume for persistent data for Redis server  
# docker inspect -f '{{ .Config.ExposedPorts }}' redis  
map[6379/tcp] ————— • Standard port for Redis exposed
```

- 3) Test the Redis server using the Redis CLI tools:

```
# docker run -d --name redis redis  
1c21656f42bf13d8db9a20429d7ed694c23ed32aac7cb5e0fe26459f121300d3  
# docker exec -ti redis /bin/bash  
/# redis-cli  
127.0.0.1:6379> ping  
PONG  
127.0.0.1:6379> set test value1  
OK  
127.0.0.1:6379> get test  
"value1"  
127.0.0.1:6379> del test
```

Lab 7

Task 1

Docker Compose

Estimated Time: 60 minutes

```
(integer) 1
127.0.0.1:6379> incr hits
(integer) 1
127.0.0.1:6379> incr hits
(integer) 2
127.0.0.1:6379> del hits
(integer) 1
127.0.0.1:6379> quit
/# exit
```

- Exit Redis CLi
- Exit container

- 4) Create an empty directory to contain all the files for building this set of containers:

```
# mkdir ~/compose
# cd ~/compose
```

- 5) Create a Dockerfile that can be used for building a web application container:

File: ~/compose/Dockerfile	
+	FROM server1:5000/python:2.7
+	ADD . /code
+	WORKDIR /code
+	RUN pip install --trusted-host=server1.example.com -r requirements.txt
+	CMD python app.py

The `--trusted-host` option is to allow pip to use the local server over HTTP instead of requiring HTTPS and a trusted cert.

- 6) Create the referenced requirements.txt file:

File: ~/compose/requirements.txt	
+	--index-url=http://server1.example.com/packages/simple/
+	flask
+	redis

The `--index-url` option is to point pip to the locally hosted repo instead of the normal Internet PyPI repo.

- 7) Create the referenced app.py file with the following content:

```
File: ~/compose/app.py
+ from flask import Flask
+ from redis import Redis
+ import os
+ app = Flask(__name__)
+ redis = Redis(host='redis', port=6379)
+
+ @app.route('/')
+ def hello():
+     redis.incr('hits')
+     return 'Hello from container %s. Site accessed %s times.\n' % (
+         os.environ.get('HOSTNAME'), redis.get('hits'))
+
+ if __name__ == "__main__":
+     app.run(host="0.0.0.0", debug=True)
```

- 8) Build an image with your new app using the Dockerfile:

```
# docker build -t myapp .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM python:2.7
Trying to pull repository server1:5000/python ...
0d1c644f790b: Download complete
... snip ...
32590848322d: Download complete
Status: Downloaded newer image for server1:5000/python:2.7
---> 0d1c644f790b
Step 1 : ADD . /code
---> 27b4539d8adf
Removing intermediate container ecb8a566061e
Step 2 : WORKDIR /code
---> Running in eb3a720d11d2
---> e4ce077eaa0d
Removing intermediate container eb3a720d11d2
Step 3 : RUN pip install --trusted-host=server1.example.com -r requirements.txt
---> Running in d8fb4c472027
```

```

Collecting flask (from -r requirements.txt (line 2))
  Downloading http://server1.example.com/packages/simple/flask/Flask-0.10.1.tar.gz (544kB)
  . . . snip . . .
Successfully built flask redis itsdangerous MarkupSafe
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, itsdangerous, flask, redis
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.10.4 flask-0.10.1 itsdangerous-0.24 redis-2.10.3
---> 08b077cca263
Removing intermediate container d8fb4c472027
Step 4 : CMD python app.py
---> Running in e24a2cb7129c
---> 2b5d27e56990
Removing intermediate container e24a2cb7129c
Successfully built 2b5d27e56990

```

- 9) Test the application by launching a container and connecting to the exposed application port several times:

```

# docker run --name apptest -p 80:5000 --link redis:redis -d myapp
8b41b1c9f015fd5aba580bfa925412d60389c3a037292031bc001fc76853cbca
# docker logs apptest
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
# docker port apptest
5000/tcp -> 0.0.0.0:80
# curl localhost
Hello from container 8b41b1c9f015f Site accessed 1 times.
# curl localhost
Hello from container 8b41b1c9f015f Site accessed 2 times.
# curl localhost
Hello from container 8b41b1c9f015f Site accessed 3 times.

```

- 10) Remove both containers in preparation of building them with Docker Compose:

```

# docker rm -fv apptest redis
apptest
redis

```

- 11) Create a new file with the following content (YAML files are very sensitive to whitespace. Use spaces not tabs and align things carefully):

File: ~/compose/docker-compose.yml

```
+ myapp:
+   build: .
+   ports:
+     - "80:5000"
+   volumes:
+     - ../code:Z
+   links:
+     - redis
+ redis:
+   image: redis
```

12) Install the Docker Compose command onto your system and verify it runs:

```
# curl server1:/packages/docker-compose > /usr/bin/docker-compose
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           % Dload  % Upload   Total    Spent    Left    Speed
100    216  100    216    0     0  1360      0 --:--:-- --:--:-- --:--:--  1367
# chmod a+x /usr/bin/docker-compose
# docker-compose -v
docker-compose version 1.7.0, build 0d7bf73
# docker-compose -h
Define and run multi-container applications with Docker.

Usage:
  docker-compose [options] [COMMAND] [ARGS...]
  docker-compose -h|--help
. . . snip . . .
```

13) Use Docker Compose to build and launch the set of containers:

```
# docker-compose up
Creating compose_redis_1...
Building myapp...
Step 0 : FROM python:2.7
----> 0d1c644f790b
Step 1 : ADD . /code
----> 74aa96b220ec
Removing intermediate container a47d114f34b1
Step 2 : WORKDIR /code
```

```

... snip ...
Step 4 : CMD python app.py
---> Running in f353e94e4a8f
---> 1e11bec6bae5
Removing intermediate container f353e94e4a8f
Successfully built 1e11bec6bae5
Creating compose_myapp_1...
Attaching to compose_redis_1, compose_myapp_1
redis_1 | 1:C 13 Jul 19:15:33.148 # Warning: no config file specified, using the default config.
In order to specify a config file use redis-server /path/to/redis.conf
redis_1 |
redis_1 |
redis_1 |
Redis 3.0.2 (00000000/0) 64 bit
... snip ...
redis_1 | 1:M 13 Jul 19:15:33.150 * The server is now ready to accept connections on port 6379
myapp_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
myapp_1 | * Restarting with stat

```

Leave this terminal window running with both containers connected.

- 14)** In another terminal window, verify that you can connect to the application and that it is using the redis container for data storage:

```
# curl localhost
Hello from container 8f5ef700507. Site accessed 1 times.
# curl localhost
Hello from container 8f5ef700507. Site accessed 2 times.
```

- 15)** List the location where the code volume maps to:

```
# docker inspect -f '{{ .HostConfig.Binds }}' compose_myapp_1
[/root/compose:/code:Z]
```

Mapping the volume this way allows you to edit the code of the application when it is running (without having to rebuild the image).

- 16)** Edit the code changing it to display the hostname instead of the container ID:

```
File: ~/compose/app.py
def hello():
    redis.incr('hits')
→    return 'Hello from container %s Site accessed %s times.\n' % (
        os.environ.get('HOSTNAME'),redis.get('hits'))
```

17) Test that the changes to the code took effect:

```
# curl localhost
Hello from host 045fdeace2f3. Site accessed 3 times.
```

18) Return to the original terminal where the containers are connected and stop them:

```
redis_1 | 1:M 13 Jul 19:22:02.727 * The server is now ready to accept connections on port 6379
myapp_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
myapp_1 | * Restarting with stat
[Ctrl]+[C] Gracefully stopping... (press Ctrl+C again to force)
Stopping compose_myapp_1... done
Stopping compose_redis_1... done
```

19) Launch the containers again this time as daemons:

```
# docker-compose up -d
Starting compose_redis_1...
Starting compose_myapp_1...
```

20) List all containers that are defined by the docker-compose.yml in the current directory:

```
# docker-compose ps
```

Name	Command	State	Ports
compose_myapp_1	/bin/sh -c python app.py	Up	0.0.0.0:80->5000/tcp
compose_redis_1	/entrypoint.sh redis-server	Up	6379/tcp

21) Run a command inside one of the containers that is part of this compose group:

```
# docker-compose run --rm myapp cat /etc/hosts
```



```

. . . snip . . .
172.17.0.2      compose_redis_1 1778a66alcc4
172.17.0.2      redis 1778a66alcc4 compose_redis_1
172.17.0.2      redis_1 1778a66alcc4 compose_redis_1
172.17.0.3      compose_myapp_1 87633385b0c6
172.17.0.3      myapp 87633385b0c6 compose_myapp_1
172.17.0.3      myapp_1 87633385b0c6 compose_myapp_1
172.17.0.4      6c1f7fd6ad24

```

Note the additional aliases that include a number, for example `redis_1`. These are used when the scale features of Docker Compose are invoked.

- 22) Stop the containers in this compose group:

```

# docker-compose stop
Stopping compose_myapp_1...
Stopping compose_redis_1... done

```

Bonus: Adding a Web Proxy and Manually Scaling the Application

- 23) Create a set of subdirectories to store the builds and move the application into the new location:

```

# cd ~/compose
# mkdir {web,app}
# mv app.py requirements.txt Dockerfile app/

```

- 24) Create a new Dockerfile for building the Nginx web frontend:

File: ~/compose/web/Dockerfile	
+	FROM nginx
+	COPY nginx.conf /etc/nginx/nginx.conf

- 25) Create a config for nginx that can proxy connections on port 80 to two servers each on port 5000:

File: web/nginx.conf

```
+ events { worker_connections 1024; }
+ http {
+     upstream myapp {
+         server myapp1:5000;
+         server myapp2:5000;
+     }
+     server {
+         listen 80;
+         location / {
+             proxy_pass http://myapp;
+             proxy_http_version 1.1;
+             proxy_set_header Upgrade $http_upgrade;
+             proxy_set_header Connection 'upgrade';
+             proxy_set_header Host $host;
+             proxy_cache_bypass $http_upgrade;
+         }
+     }
+ }
```

26) Replace the existing docker-compose.yml with the following:

File: docker-compose.yml

```
+ web:
+   build: ./web
+   links:
+     - myapp1:myapp1
+     - myapp2:myapp2
+   ports:
+     - "80:80"
+ myapp1:
+   build: ./app
+   ports:
+     - "5000"
+   volumes:
+     - ./app:/code:z
+   links:
+     - redis
+ myapp2:
+   build: ./app
+   ports:
+     - "5000"
+   volumes:
+     - ./app:/code:z
+   links:
+     - redis
+ redis:
+   image: redis
```

27) Build and start the new set of containers:

```
# docker-compose up
... snip ...
redis_1 | 1:M 14 Jul 02:38:41.540 * DB loaded from disk: 0.000 seconds
redis_1 | 1:M 14 Jul 02:38:41.540 * The server is now ready to accept connections on port 6379
myapp2_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
myapp2_1 | * Restarting with stat
myapp1_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
myapp1_1 | * Restarting with stat
```

28) In another terminal window, examine the set of containers:

```
# docker-compose ps
      Name                                Command                                State      Ports
-----
compose_myapp1_1  /bin/sh -c python app.py             Up          0.0.0.0:32780->5000/tcp
compose_myapp2_1  /bin/sh -c python app.py             Up          0.0.0.0:32779->5000/tcp
compose_redis_1   /entrypoint.sh redis-server          Up          6379/tcp
compose_web_1     nginx -g daemon off;                 Up          443/tcp, 0.0.0.0:80->80/tcp
```

29) Test the round robin load balancing by connecting to port 80 twice and examining the host names to verify you are directed to both containers:

```
# curl localhost
container 51eb8e777a21 accessed 3 times.
# curl localhost
container a959cc019324 accessed 4 times.
```

30) Return to the original terminal where you launched the containers and stop them by pressing `Ctrl+C`:

```
myapp1_1 | 172.17.0.69 - - [14/Jul/2015 02:39:14] "GET / HTTP/1.1" 200 -
web_1    | 172.17.42.1 - - [14/Jul/2015:02:39:14 +0000] "GET / HTTP/1.1" 200 53 "-" "curl/7.29.0"
myapp2_1 | 172.17.0.69 - - [14/Jul/2015 02:39:16] "GET / HTTP/1.1" 200 -
web_1    | 172.17.42.1 - - [14/Jul/2015:02:39:16 +0000] "GET / HTTP/1.1" 200 53 "-" "curl/7.29.0"
Ctrl+C Gracefully stopping... (press Ctrl+C again to force)
Stopping compose_web_1 ... done
Stopping compose_myapp1_1 ... done
Stopping compose_myapp2_1 ... done
Stopping compose_redis_1 ... done
```

Lab 7

Task 2

Docker Swarm

Estimated Time: 20 minutes

Objectives

- ☞ Connect a Docker host to an existing Swarm using the containerized Swarm binary.
- ☞ Explore the differences between a single Docker host, and a swarm when creating containers and networks.

Requirements

- 🖥 (1 station) 🖥 (classroom server)

Relevance

Docker can be scaled to handle larger production demands by clustering a group of Docker hosts together. Docker Swarm manages a collection of Docker hosts and provides a Docker API compliant interface to manage that set of hosts.

Notices

- ☞ This lab task requires the shared classroom server to be running a Swarm Primary Manager process listening on port 4000, and a Consul server listening on port 8500. Normally this is setup ahead of time by the instructor. The following two containers (running on the server) will provide the needed services:

```
# docker run -dp 8500:8500 -h consul --name consul \
--restart=always consul -server -bootstrap
```

```
# docker run -d -p 4000:4000 --name swarm --restart=always \
swarm manage -H :4000 --replication \
--advertise 10.100.0.254:4000 consul://10.100.0.254:8500
```

- 1) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -
```

```
Password: makeitso 
```

- 2) Reconfigure your first assigned node so that the Docker process listens on the network as well as the normal UNIX socket:

File: /etc/sysconfig/docker

```
→ OPTIONS='-H fd://-H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock \  
  --registry-mirror http://server1:5000 \  
  --insecure-registry 10.100.0.0/24 \  
  . . . snip . . .
```

```
# systemctl restart docker
```

```
# lsof -i :2376
```

```
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME  
docker  24886 root   7u    IPv6 869765      0t0  TCP *:2376 (LISTEN)
```

3) Examine the options needed to join a Swarm:

```
# docker run --rm server1:5000/swarm join --help
```

4) Run a swarm node process as a container to connect it to the existing primary manager running on the shared server:

```
# docker run -d --restart=always --name swarm server1:5000/swarm join --advertise=10.100.0.X:2376  
  consul://10.100.0.254:8500  
daa8bea7e9b1fd4b48c747f76ebc532b6294b574761c7a50e434e9bc2331393d
```

5) Query the Swarm primary manager to verify that your node is seen as part of the swarm and reports a status of Healthy:

```
# docker -H 10.100.0.254:4000 info | grep -A8 stationX  
station1.example.com: 10.100.0.X:2376  
  ` Status: Healthy  
  ` Containers: 1  
  ` Reserved CPUs: 0 / 1  
  ` Reserved Memory: 0 B / 777.2 MiB  
  ` Labels: executiondriver=native-0.2, kernelversion=3.10.0-327.4.4.el7.x86_64, operatingsystem=Red Hat  
Enterprise Linux Server 7.2 (Maipo), storagedriver=devicemapper  
  ` Error: (none)  
  ` UpdatedAt: 2016-04-26T23:39:29Z  
  ` ServerVersion: 1.10.3
```

6) Identify how many nodes have joined the swarm:

```
# docker -H 10.100.0.254:4000 info | grep Nodes
Nodes: 2
```

Depending on the progress of other students, you may see fewer, or a greater number of nodes than the sample output shown.

7) Configure your environment so that the Docker client uses the Swarm by default:

```
# docker info | grep ^Ser
Server Version: 1.10.3
# export DOCKER_HOST='tcp://server1:4000'
# docker info | grep ^Ser
Server Version: swarm/1.2.0
```

- Local Docker daemon
- Docker Swarm primary manager running on the shared server

8) Create a network and launch a container connected to that network (be sure to replace all instances of *X* with the number of your first assigned node):

```
# docker network create sX-net
c5653601efcdd54aa6097289fdb4929871ef794283e52fca9b56511c97a8d7cc
# docker run -dti --name sX-test --net=sX-net server1:5000/busybox
09af5238111a7f02449ba8df71ea377b8521795f49e0621dcdb01b1c6a7f0e3d
```

9) Examine the network that was created:

```
# docker network ls | grep sX-net
c5653601efcd      sX-net          overlay
# docker network ls
. . . output omitted . . .
```

Note that networks created by the Swarm use the overlay driver automatically and are available to all hosts within the swarm.

10) Discover which node your container is running on:

```
# docker inspect -f '{{.Node.Name}}' sX-test
station3.example.com
```

- 11) List all the containers in the Swarm showing their container IDs, and names sorted by the node they are running on:

```
# docker ps --format '{{.ID}} - {{.Names}}' | sort -k2
c97ec28c7d2b - station1.example.com/s3-test
daa8bea7e9b1 - station1.example.com/swarm
09af5238111a - station3.example.com/s1-test
91d15acf97ad - station3.example.com/swarm
. . . snip . . .
```

Both container IDs and container names must be unique when starting containers on a Swarm. The swarm node containers have the same name because they were started on individual nodes before the swarm was formed. Attempts to start containers with duplicate names after joining the swarm will be blocked and return an error.

- 12) Pull the names of all containers within the Swarm directly via the API endpoint:

```
# curl -s server1:4000/containers/json | egrep -o 'Names[^\]]+]'
Names":["/station1.example.com/swarm"]
Names":["/station1.example.com/s1-test"]
Names":["/station3.example.com/swarm"]
Names":["/station1.example.com/s3-test"]
```

- 13) Run a command in your test container:

```
# docker exec s1-test ip addr li
Error response from daemon: No such container sX-test
# docker exec stationY.example.com/sX-test ip addr li eth1
54: eth1@if55: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth1
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
        valid_lft forever preferred_lft forever
```

- When using Swarm, the container name must also include the node it is running on. Consult the output from the previous step if needed.

Cleanup

- 14) Remove the test container and network, and leave the Swarm:


```
# docker rm -f stationY.example.com/sX-test
station3.example.com/sX-test
# docker network rm sX-net
# unset DOCKER_HOST
# docker stop swarm
swarm
# docker rm swarm
```




Appendix

A

**CONTINUOUS
INTEGRATION
WITH GITLAB,
GITLAB CI, AND
DOCKER**

Lab 1

Estimated Time: 135 minutes

Task 1: GitLab and GitLab CI Setup

Page: 1-9 Time: 15 minutes

Requirements: (0 stations) 🖥️ (classroom server)

Task 2: Unit and Functional Tests

Page: 1-15 Time: 120 minutes

Requirements: 🖥️ (1 station) 🖥️ (classroom server)

Objectives

- 🔗 Install and configure GitLab
- 🔗 Install and configure GitLab CI

Requirements

(0 stations) 🖥️ (classroom server)

Relevance

GitLab and GitLab CI is one of the most popular open-source combinations for a CI enabled workflow.

Notices

- 🔗 A single instance of this service is shared in the classroom and the instructor will normally perform this lab as a demo on the shared classroom server.

- 1) DO NOT DO THIS LAB (on student stations). The below steps are for your reference. The lab is completed by the instructor on the classroom server system.

```
# [ "$(hostname -s)" = "server1" ] || echo 'Do not proceed!'
```

- 2) Verify that the system has at least 4G of RAM and 2 CPU cores:

```
# free
      total        used        free      shared  buff/cache   available
Mem:    4047620    701960    2829652       10720       516008       3088692
Swap:    524284           0       524284
# grep processor /proc/cpuinfo
processor       : 0
processor       : 1
. . . output omitted . . .
```

- 3) Install the GitLab-ce package:

```
# yum localinstall -y /export/courserepos/R7/docker/gitlab-ce-7*
. . . output omitted . . .
```

- 4) Add new host names to DNS for the GitLab services:

```
# echo "git IN A 10.100.0.254" >> /var/named/data/named.example.com
```

Lab 1

Task 1

GitLab and GitLab CI Setup

Estimated Time: 15 minutes

```
# echo "ci IN A 10.100.0.254" >> /var/named/data/named.example.com
# systemctl restart named
# host ci
ci.example.com has address 10.100.0.254
# host git
git.example.com has address 10.100.0.254
```

- 5) GitLab sends some email to admin. Create an alias that directs that email to root:

```
# echo -e "admin:\troot" >> /etc/aliases
# newaliases
```

- 6) Run a reconfigure to produce a base config:

```
# gitlab-ctl reconfigure
. . . output omitted (This can take a few minutes) . . .
```

Hangs during install can be caused by the gitlab-runsvdir.service not being started.
Manually start and retry if a hang occurs.

- 7) Modify the config, changing URLs used by GitLab:

File: /etc/gitlab/gitlab.rb	
-	external_url 'http://server1.example.com'
+	external_url 'http://git.example.com:8000'
	. . . output omitted . . .
-	# ci_external_url 'http://ci.example.com'
+	ci_external_url 'http://ci.example.com:8001'

- 8) Configure GitLab to use the classroom LDAP server for accounts:

File: /etc/gitlab/gitlab.rb

```
+ gitlab_rails['ldap_enabled'] = true
+ gitlab_rails['ldap_servers'] = YAML.load <<-'EOS'
+   main:
+     label: 'Server1 LDAP Accounts'
+     host: 'server1.example.com'
+     port: 389
+     uid: 'uid'
+     method: 'tls'
+     bind_dn: 'cn=Manager,dc=example,dc=com'
+     password: 'secret-ldap-pass'
+     active_directory: false
+     base: 'ou=DevAccts,dc=example,dc=com'
+ EOS
```

9) Configure GitLab to not use Gravatar:

File: /etc/gitlab/gitlab.rb

```
+ gitlab_rails['gravatar_plain_url'] = 'http://127.0.0.1'
+ gitlab_rails['gravatar_ssl_url'] = 'https://127.0.0.1'
+ gitlab_ci['gravatar_enabled'] = false
```

10) Reconfigure GitLab so the accumulated changes take effect:

```
# gitlab-ctl reconfigure && sleep 15
. . . output omitted . . .
# lsof -i :8000-8200,5000 -nP
. . . output omitted . . .
```

11) Change the password for the root GitLab user by signing in via the web interface:

<http://git.example.com:8000>

Current credentials: u: root, p: 5iveL!fe

12) Connect to the GitLab CI site to show the authorization token for registering shared runners:

Open a browser to ci.example.com:8001

The GitLab CI main page opens

Click 'Login with GitLabs account.'

The page refreshes to the GitLabs login

Login as the root user

The page redirects to an 'Authorize Required' page

Click 'Authorize' The page refreshes to the GitLab Ci Dashboard

Click on 'Admin' --> 'Runners'

The needed token is displayed

Objectives

- 🔗 Create and use a project within GitLab
- 🔗 Use the Python virtualenvwrapper to create a project sandbox
- 🔗 Create unit tests, and functional tests

Requirements

- 💻 (1 station) 🖥️ (classroom server)

Relevance

Continuous integration relies on unit and functional tests run by the CI server to determine if code commits still result in working software.

Lab 1

Task 2

Unit and Functional Tests

Estimated Time: 120 minutes

1) Create a project within GitLab to hold your code:

login to the <http://git.example.com:8000> URL with the assigned devX LDAP user (password=password).
The 'Welcome to GitLab!' screen is displayed

Click 'New Project'

The screen refreshes to the 'New Project' page

Enter a project path of: mod10-example-stationX

Click to change 'Visibility Level' to public.

Click 'Create Project'

The screen refreshes and confirms that the project was created and is currently empty

Click on the HTTP button

The HTTP project address is displayed instead of the SSH address

Scroll down to where the example git commands are shown for a reference

2) Install Git onto the system:

```
$ sudo yum install -y git  
... output omitted ...
```

3) As the guru user, do basic Git configuration for your project:

```
$ git config --global user.name "devX"  
$ git config --global user.email "guru@stationX.example.com"  
$ cd; mkdir mod10-example  
$ cd mod10-example
```

```
$ git init
Initialized empty Git repository in /home/guru/mod10-example/.git/
```

- 4) Commit a description file for the project:

```
$ echo "Credit card number validator via mod10 Luhn" > README.md
$ git add README.md
$ git commit -m "Project description"
[master (root-commit) 7b3fa07] Project description
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

- 5) Create a file with your HTTP auth info:

File: /home/guru/.netrc	
+	machine git.example.com
+	login devX
+	password password

- 6) Configure remote HTTP access to GitLab and push into project (Be sure to replace the X with your proper user/station number):

```
$ git remote add gitlab http://git.example.com:8000/devX/mod10-example-stationX.git
$ git push gitlab master
Counting objects: 3, done.
Writing objects: 100% (3/3), 261 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://git.example.com:8000/devX/mod10-example-stationX.git
* [new branch]      master -> master
```

Creating and Using a Virtual Environment

- 7) Install the Python virtual environment package:

```
$ sudo yum install -y python-pip
... snip ...
Installed:
python-pip.noarch 0:7.1.0-1.el7

Complete!
```

```
$ sudo pip install --trusted-host=server1.example.com --index-url=http://server1.example.com/packages/simple virtualenvwrapper
... snip ...
Collecting virtualenv (from virtualenvwrapper)
  Downloading http://server1.example.com/packages/simple/virtualenv/virtualenv-13.1.2-py2.py3-none-any.whl (1.7MB)
... snip ...
Running setup.py install for virtualenv-clone
Successfully installed argparse-1.3.0 pbr-1.6.0 six-1.9.0 stevedore-1.7.0 virtualenv-13.1.2 virtualenv-clone-0.2.6 virtualenvwrapper-4.7.0
```

- 8) Create a directory to serve as a sandbox environment home and set your shell to load the related functions on login:

```
$ mkdir -p ~/AppData/env
$ echo "export WORKON_HOME=$HOME/AppData/env" >> ~/.bashrc
$ echo "source /bin/virtualenvwrapper.sh" >> ~/.bashrc
```

- 9) Create and activate a new virtual environment for the project:

```
$ source ~/.bashrc
$ mkvirtualenv mod10-example
New python executable in mod10-example/bin/python
Installing setuptools, pip, wheel...done.
(mod10-example)[guru@stationX ~]$
$ workon
(mod10-example)
```

- Prompt changes to reflect current environment.
- Can be used anytime to see current environment name.

- 10) Examine the files created to support the new virtual environment:

```
$ du -sh ~/AppData/env/mod10-example/
9.5M    AppData/env/mod10-example/
$ ls -R ~/AppData/env/mod10-example/
... output omitted ...
```

- 11) Configure the structure of the mod10-example project and source tree. Due to its module structure, Python allows for great flexibility in how the code tree can be structured. There are a few general guidelines, however, so that project's are generally compatible with pip:

- ✂ scripts/ or bin/ folder for command-line interfaces to the program or library
- ✂ tests/ folder for tests
- ✂ lib/ for C-language libraries
- ✂ doc/ for documentation
- ✂ a folder for the source code of the program, in this case mod10/

In addition to the recommended directories, we will create an ftests/ directory for functional tests. Create the basic directories to hold the actual project code and test code:

```
$ cd ~/mod10-example/
$ mkdir {tests,ftests,mod10}
$ touch ./{tests,ftests,mod10}/__init__.py
```

• Allow directories to be used as Python packages

Main Application Code

- 12) The Luhn algorithm (also called the "modulus 10" or "mod 10" algorithm) is a simple way to partially validate whether a credit card number is valid without having to explicitly sending a request to the credit card company. Note: it doesn't tell you anything about whether the card is owned by anyone, or if it is authorized for charges. That still requires a request to a payment processor.

Implemented as a simple checksum, mod10 is a component of the numbering systems used by all of the major credit card companies today. Though it isn't cryptographically secure, it is a useful check for mistaken or mistyped credit card numbers. Briefly, here is how it determines whether a credit card number is valid:

1. Reverse the card number, collect the digits in odd and even positions.
2. For digits in even positions, multiply by two, this creates the "Luhn" digits.
3. For Luhn digits greater than 10, subtract nine.
4. Add the Luhn and off-digit numbers together
5. Take the number from step 4 and calculate the modulo 10. If there is no remainder, the credit card number is valid.

Note: The next steps show the code listing as a reference. Your instructor may provide you with the completed code to save time in class.

- 13) Create the core mod10 library which provides the logic for the Luhn credit card validation:

File: ~/mod10-example/mod10/mod10.py

```
+ import six
+ import string

+ SET_DIGITS = set(string.digits)

+ def ccnum_isvalid(ccstr, length_limit=30):
+     ''' Validate a credit card number via the Luhn algorithm
+         ::input ccstr (str, unicode): Credit card number to be validated
+         ::Input lenth_limit (int ,default=30): Maximum allowed
+             credit card length
+         ::returns bool: True if valid, false otherwise
+     '''
```

14) Add the basic checks:

File: ~/mod10-example/mod10/mod10.py

```
         ::returns bool: True if valid, false otherwise
         ...

+     # Validate input as a string
+     if not isinstance(ccstr, six.string_types):
+         raise TypeError(
+             ( 'Invalid input format %r, CC numbers should be input'
+               'as a valid Python string type') % type(ccstr) )

+     # Check input length, credit card numbers are always a fixed length
+     if len(ccstr) > length_limit:
+         raise ValueError(
+             ('Invalid credit card input number, value is longer than allowed '
+              'maximum length'))

+     # Ensure that only numeric characters are present
+     if set(ccstr).difference(SET_DIGITS):
+         raise ValueError(
+             ('Invalid credit card number %s, credit card numbers may not '
+              'contain letter values') % ccstr)
```

15) Add the Luhn checksum check:

File: ~/mod10-example/mod10/mod10.py

```
+         ('Invalid credit card number %s, credit card numbers may not '  
+         'contain letter values') % ccstr)  
+     # Reverse the card number, collect the digits in odd and even positions  
+     # For digits in even positions, multiply by two  
+     off_digits = [ int(ccdigit) for ccdigit in ccstr[-1::-2] ]  
+     luhn_digits = [ int(ccdigit)*2 for ccdigit in ccstr[-2::-2] ]  
  
+     # For luhn digits greater than 10, scale the digits  
+     scaled_luhn_digits = map(lambda i: i-9 if i > 9 else i, luhn_digits)  
  
+     # Take the sum of the luhn digits add the sum of the off digits,  
+     # calculate modulo 10, if there is no remainder, the card is valid  
+     return (sum(scaled_luhn_digits) + sum(off_digits)) % 10 == 0
```

16) The server component of the credit card validation is implemented as a simple Python web app using the Flask framework:

- ✎ It will have a single `/validate` endpoint that will take a credit card number encoded as a POST parameter.
- ✎ It will use the mod10 library to determine if the number is valid.
- ✎ If the number is valid, it will return a HTTP 200 status code with a JSON body.
- ✎ If the number is invalid (or the request was badly formed), it will return a HTTP 400 status code with a JSON body providing details as to why the credit card number was invalid.

17) Create the Python web app that uses the CC validation library starting with boilerplate includes:

File: ~/mod10-example/mod10/server.py

```
+ import json
+
+ # Import Flask components
+ from flask import Flask
+ from flask import request
+ from flask import Response
+
+ # Import mod10 validation
+ from mod10 import ccnum_isvalid
```

18) Add the core Flask based web framework:

File: ~/mod10-example/mod10/server.py

```
# Import mod10 validation
from mod10 import ccnum_isvalid

+ # Create application instance
+ app = Flask(__name__)

+ # Create default resource
+ @app.route('/validate/', methods=['POST'])
+ def validate_ccnum():
+     response = {'operation': 'validate-credit-card'}

+     # Retrieve credit card number from the request and validate
+     try:
+         ccvalid = ccnum_isvalid(request.headers.get('Credit-Card'))
+         if ccvalid:
+             response['valid'] = True
+             return Response(
+                 json.dumps(response),
+                 status=200,
+                 mimetype='application/json')

+         raise ValueError('Invalid credit card number: %s'
+             % request.headers.get('Credit-Card'))
```

19) Add the error checking and application startup:

```
File: ~/mod10-example/mod10/server.py
    raise ValueError('Invalid credit card number: %s'
        % request.headers.get('Credit-Card'))

+   # Expected errors, TypeError/ValueError
+   except (TypeError, ValueError) as err:
+       response['valid'] = False
+       response['error'] = str(err)
+       statuscode = 400

+   # Unspecified errors
+   except Exception as err:
+       response['valid'] = False
+       response['error'] = str(err)
+       statuscode = 500

+   return Response(
+       json.dumps(response),
+       status=statuscode,
+       mimetype='application/json')

+ # Start the application instance when script is run
+ if __name__ == '__main__':
+     app.run(host='0.0.0.0', port=8080)
```

20) Create a file that specifies the required libraries to run the app:

```
File: ~/mod10-example/requirements.txt
+ --index-url=http://server1.example.com/packages/simple/
+ six==1.9.0
+ Flask==0.10.1
+ requests==2.7.0
+ argparse==1.3.0
```


21) Install the required libraries:

```
$ pip install --trusted-host=server1.example.com -r requirements.txt
... snip ...
Installing collected packages: six, Werkzeug, MarkupSafe, Jinja2, ↵
    itsdangerous, Flask, requests, argparse
Successfully installed Flask-0.10.1 Jinja2-2.8 MarkupSafe-0.23↵
    Werkzeug-0.10.4 argparse-1.3.0 itsdangerous-0.24 requests-2.7.0 six-1.9.0
```

Note that root access is not required because the packages are being installed into the virtual environment.

22) Test that the application launches and listens on local port 8080:

```
$ python mod10/server.py
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
[Ctrl]+[C]
```

23) Configure git to ignore the compiled Python code:

```
$ echo "*.pyc" >> .gitignore
```

24) Commit the working core code to git and post to GitLab:

```
$ git add .gitignore mod10 requirements.txt
$ git commit -m "Core project code"
[master c282199] Core project code
6 files changed, 104 insertions(+)
create mode 100644 .gitignore
create mode 100644 mod10/__init__.py
create mode 100644 mod10/hello.py
create mode 100644 mod10/mod10.py
create mode 100644 mod10/server.py
create mode 100644 requirements.txt
$ git push gitlab master
Counting objects: 10, done.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 2.05 KiB | 0 bytes/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To http://git.example.com:8000/dev1/mod10-example-station1.git
```

Creating a Unit Test Suite

- 25) With the project skeleton, source code, and dependencies in place; a robust test suite can be created that will ensure the code functions the way that it is supposed to. Provided as part of the Python standard library is a robust testing system called unittest, which is inspired by a similar system written for the Java programming language called JUnit. Both unittest and JUnit are the de-facto standards for unit testing for Python and Java, respectively.

Unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collection, and the independence of the tests from the reporting framework. A full discussion of its capabilities and features is well beyond the scope of this course. However, to be comfortable with the testing code provided below, there are several high level concepts which you should be aware.

Unittest organizes its code into fixtures, cases, and suites. These are in turn run by a test runner.

- ✎ Test fixtures represent the preparation needed to perform one of more tests, and any associated cleanup actions.
- ✎ Test cases are the smallest unit of testing. They are used to check for specific responses to a particular set of inputs. Unittest provides a class which implements many of the methods needed for test cases.
- ✎ Test suites represent a collection of test cases, tests suites, or both. It is used to aggregate tests which should be executed together.

The `TestCase` class provided by unittest is the primary tool that we will use to create and execute tests. We will, in turn, use the default runner provided by unittest to run the tests and report the results. We will keep all of our tests cases in the `tests/` folder, which helps to separate it from the program logic. The test suite should cover:

1. A set of "valid" numbers, following the general rules for Visa, American Express, and MasterCard.
2. A set of "invalid" numbers, which are plausible but do not validate.
3. A number which includes alphabetic characters to check that the method raises a `ValueError`.

4. A number which is submitted as a integer or float type (rather than a string) to ensure that a `TypeError` is raised.
5. A number longer than the maximum number of allowed digits to ensure that a `ValueError` is raised.

Due to the way that unittest organizes test suites, each test case will be implemented on a common `TestCase` class with a method name which begins with `test_`. Individual test assertions are performed using the assert methods provided by the `TestCase` class.

- 26)** Start by creating a small suite of tests which checks the logic for our `ccnum_isvalid` method:

File: ~/mod10-example/tests/test_mod10.py

```
+ import os, sys
+
+ # Add the project root to the Python path to allow the import of project code
+ TESTS_ROOT = os.path.abspath(os.path.dirname(__file__))
+ PROJECT_ROOT = os.path.dirname(TESTS_ROOT)
+ if not PROJECT_ROOT in sys.path:
+     sys.path.append(PROJECT_ROOT)
+
+ # Test imports
+ import unittest
+ from mod10.mod10 import ccnum_isvalid
+
+ class Mod10ValidationTestCases(unittest.TestCase):
+     ''' Test cases for mod10 validation
+     '''
+
+     def test_ValidMod10CreditCards(self):
+         ''' Check mod10 validation of valid credit card numbers
+         '''
+
+         # American Express Test Number
+         self.assertEqual(ccnum_isvalid('378282246310005'), True)
+         self.assertEqual(ccnum_isvalid('371449635398431'), True)
+         # Visa
+         self.assertEqual(ccnum_isvalid('4111111111111111'), True)
+         self.assertEqual(ccnum_isvalid('4012888888881881'), True)
+         # MasterCard
+         self.assertEqual(ccnum_isvalid('5555555555554444'), True)
+         self.assertEqual(ccnum_isvalid('5105105105105100'), True)
```

27) Run the test to verify that it works:

```
$ python -m unittest tests.test_mod10
```

.

Ran 1 test in 0.000s

OK

28) Add additional tests by appending the following lines to the already existing file:

```
File: ~/mod10-example/tests/test_mod10.py
+ def test_InvalidMod10CreditCards(self):
+     ''' Check mod10 validation of invalid credit card numbers
+     ...
+     self.assertEqual(ccnum_isvalid('123456780123'), False)
+     self.assertEqual(ccnum_isvalid('98765421'), False)
+     self.assertEqual(ccnum_isvalid('4111123456789000'), False)
+     self.assertEqual(ccnum_isvalid('5105015987654321'), False)
+     self.assertEqual(ccnum_isvalid('3782822463100013'), False)
+
+ def test_InvalidCreditCardErrors(self):
+     ''' Check that ccnum_isvalid raises expected errors
+     ...
+     self.assertRaises(TypeError, ccnum_isvalid, 123456780123)
+     self.assertRaises(ValueError, ccnum_isvalid, '378282246310005a')
+     self.assertRaises(ValueError, ccnum_isvalid,
+         '378282246310005378282246310005378282246310005378282246310005')
```

29) Run the tests again to verify that they work:

```
$ python -m unittest tests.test_mod10
```

```
.
```

```
-----
Ran 3 test in 0.000s
```

```
OK
```

30) Commit the working unit test code to git and post to GitLab:

```
$ git add tests/*.py
```

```
$ git commit -m "Unit test code"
```

```
. . . output omitted . . .
```

```
$ git push gitlab master
```

```
. . . output omitted . . .
```

Creating a Functional Test Suite

- 31) Unit tests provide a good degree of confidence that the core logic of the program are working as expected, but are only a component of a robust testing strategy. Also important are functional tests. As opposed to unit tests (which only test individual components), functional tests run against a running program and ensure that it is working according to specification.

The functional tests for our program are kept in the `ftests/` directory, and invoked using a custom runner script.

Create the runner script:

```
File: ~/mod10-example/ftests.py
+ # Functional tests for the credit card validation server
+ import argparse, logging, os, sys, urlparse
+ import unittest
+ # from ftests.ftests_mod10validation import Mod10HttpTest

+ # Configure test runner logging options
+ logger = logging.getLogger(__name__)
+ logoptions = {
+     'format': '%(levelname)s %(asctime)s: %(message)s',
+     'datefmt': '%Y-%m-%d %H:%M',
+     'level': 'DEBUG'
+ }

+ # Configure arguments for the test runner
+ parser = argparse.ArgumentParser(
+     description='Functional tests cases for mod10 verification server')
+ parser.add_argument('--loglevel', help='logging level at which messages should be reported '
+     + '(DEBUG, INFO, ERROR, CRITICAL)')
+ parser.add_argument('--testdetail', help='level of detail that should be provided when '
+     + 'test cases are running. For higher numbers, more detail will be provided. '
+     + 'Default=2.')
+ parser.add_argument('--method', help='the name of the specific method to test')
```

- 32) Functional test runner part 2 (append to end of existing file):

File: ~/mod10-example/ftests.py

```
+ def add_functional_test_suite(testcase_class=None, method=None):  
+     ''' Helper function to load test classes into a custom runner  
+     ...  
+     suite = unittest.TestSuite()  
+     if testcase_class:  
+         testloader = unittest.TestLoader()  
+         testnames = testloader.getTestCaseNames(testcase_class)  
+         for name in testnames:  
+             if name == method or method == None:  
+                 suite.addTest(testcase_class(name))  
+     return suite
```

33) Functional test runner part 3 (append to end of existing file):

File: ~/mod10-example/ftests.py

```
+ # Execute test cases  
+ if __name__ == '__main__':  
+     args = parser.parse_args()  
+  
+     # Log level and script verbosity  
+     if args.loglevel:  
+         logoptions['level'] = args.loglevel  
+         logging.basicConfig(**logoptions)  
+     if args.testdetail:  
+         verbosity = int(args.testdetail)  
+     else:  
+         verbosity = 2  
+  
+     # Create testing suite  
+     # Add calls to tests below:  
+     testloader = unittest.TestLoader()  
+     # mod10_suite = unittest.TestSuite()  
+     # mod10_suite.addTest(add_functional_test_suite(Mod10HttpTest, args.method))  
+     # result = unittest.TextTestRunner(verbosity=int(verbosity)).run(mod10_suite)  
+     sys.exit(not result.wasSuccessful())
```

- 34) Create a functional test that uses the REST API of the CC validation server to check a sample card:

```
File: ~/mod10-example/ftests/ftests_mod10validation.py
+ import os, logging, posixpath, unittest, requests, json
+
+ logger = logging.getLogger(__name__)
+
+ class Mod10HttpTest(unittest.TestCase):
+
+     def test_Mod10CCNumValidation(self):
+         r = requests.post('http://127.0.0.1:8080/validate',
+             headers={'Credit-Card': '378282246310005'})
+         self.assertEqual(r.status_code, 200)
+         rdata = json.loads(r.content)
+         self.assertEqual(rdata.get('valid'), True)
```

- 35) Activate the new functional test by adding, or uncommenting, the following lines within the test runner script:

```
File: ~/mod10-example/ftests.py
import argparse, logging, os, sys, urlparse
import unittest
+ from ftests.ftests_mod10validation import Mod10HttpTest

# Configure test runner logging options
logger = logging.getLogger(__name__)

... snip ...

testloader = unittest.TestLoader()
+ mod10_suite = unittest.TestSuite()
+ mod10_suite.addTest(add_functional_test_suite(Mod10HttpTest, args.method))
+ result = unittest.TextTestRunner(verbosity=int(verbosity)).run(mod10_suite)
sys.exit(not result.wasSuccessful())
```


- 36) Start the server application in the background and then run the functional test to verify that it works:

```
$ cd ~/mod10-example/
$ python mod10/server.py &
[1] 4819
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
$ python ftests.py
test_Mod10CCNumValidation (ftests.ftests_mod10validation.Mod10HttpTest) ...
INFO 2015-08-07 15:07: Starting new HTTP connection (1): 127.0.0.1
127.0.0.1 - - [07/Aug/2015 15:07:07] "POST /validate HTTP/1.1" 200 -
DEBUG 2015-08-07 15:07: "POST /validate HTTP/1.1" 200 52
ok

-----
Ran 1 test in 0.020s

OK
$ kill %1
$ 
[1]+  Terminated                  python mod10/server.py
```

- 37) Commit the working functional test code to git and post to GitLab:

```
$ git add ftests.py ftests/*.py
$ git commit -m "Functional test code"
. . . output omitted . . .
$ git push gitlab master
. . . output omitted . . .
```


Creating and Registering a CI Test Runner

- 38) Click on the GitLab 'Dashboard' link in the browser and confirm that you see the master branch and list of recent commits.

Installing and Registering a CI Runner

- 39) The following actions require administrative privileges. **Switch** to a root login shell:

```
$ su -
```

Password: *makeitso* 

40) Install the GitLab CI test runner:

```
# yum install -y gitlab-ci-multi-runner
... snip ...
Installed:
  gitlab-ci-multi-runner.x86_64 0:0.5.5_1_g69bc934-1

Complete!
```

41) Add your project to the CI server:

Open a web browser to the <http://ci.example.com:8001> URL
The main GitLab CI page is displayed

Click on 'Login via GitLab'
The page refreshes and shows either the GitLab login page, or the GitLab authorize page if you previously clicked "Remember me".

Login and Authorize as needed
The page refreshes and displays the GitLab CI main page that displays your project.

Click 'Add project to CI'
The page refreshes and displays the 'Project was successfully created' message.

42) Obtain the security token from GitLab CI that is needed to register a runner for your project:

Click on the 'Runners' link
The page refreshes to display the runners setup info

Scroll down and identify the specific runners security token (will be a 31 hex number)

43) Register a runner with the CI server:

```
# gitlab-ci-multi-runner register
Please enter the gitlab-ci coordinator URL (e.g. http://gitlab-ci.org:3000/):
http://ci.example.com:8001
Please enter the gitlab-ci token for this runner:
46eb7d1074a9f68dcc5fb5bd7c03c0
```

Please enter the gitlab-ci description for this runner:

[server1.example.com]:

INFO[0077] 46eb7d1 Registering runner... succeeded

Please enter the executor: ssh, shell, parallels, docker, docker-ssh:

[shell]: **docker**

Please enter the Docker image (eg. ruby:2.1):

ubuntu:14.04

If you want to enable mysql please enter version (X.Y) or enter latest?

If you want to enable postgres please enter version (X.Y) or enter latest?

If you want to enable redis please enter version (X.Y) or enter latest?

If you want to enable mongo please enter version (X.Y) or enter latest?

INFO[0399] Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

- 44) GitLab CI limits the use of custom images when invoking builds through Docker. The runner register program doesn't allow passing the config option to enable the images and services needed for this project.

As root, edit the config adding the needed lines:

File: /etc/gitlab-runner/config.toml

```
concurrent = 1

[[runners]]
  name = "stationX.example.com"
  url = "http://ci.example.com:8001"
  token = "46eb7d1074a9f68dcc5fb5bd7c03c0"
  limit = 1
  executor = "docker"
  [runners.docker]
    image = "ubuntu:14.04"
+   allowed_images = ["ubuntu:*", "python:*"]
+   allowed_services = ["mysql:*", "redis:*", "postgres:*"]
    privileged = false
    volumes = ["/cache"]
```

With concurrent set to 1, the tests will run serially. For parallel operation, you can change that value to 2 or more.

- 45) Verify the runner shows an active (running) status before proceeding:

```
# systemctl restart gitlab-runner
# systemctl status gitlab-runner
gitlab-runner.service - GitLab Runner
   Loaded: loaded (/etc/systemd/system/gitlab-runner.service; enabled)
   Active: active (running) since Thu 2015-08-06 15:15:26 MDT; 3min 49s ago
   . . . output omitted . . .
```

- 46) Refresh the GitLab CI web page showing your project runners and verify your runner is listed as 'Activated' for the project (you may need to scroll down the page).

Do not proceed until you have confirmed the runner is activated with the CI server!

- 47) Administrative privileges are no longer required; **exit** the root shell to return to an unprivileged account:

```
# exit
```

Configuring a CI Runner for Unit Testing

- 48) To prepare for builds to run, you must create a configuration file to manage the execution of scripts. GitLab makes use of a configuration file (named `.gitlab-ci.yml`) which describes how a project will be built. The file uses YAML syntax for specifying setup, configuration, and testing. The file should be created in the root of the repository.

A GitLab CI configuration includes several sections:

- ❏ optional parameters which describe the docker environment (the `image` and `services` parameters)
- ❏ parameters which control the environment configuration (`before_script`)
- ❏ parameters which govern the types of jobs which can be defined (`types`)
- ❏ specific jobs which will be executed (the initial config file for this project will not have any specified jobs, they will be defined later)

To have GitLab CI use Docker for tests and builds, we must provide information about the Docker configuration. This includes which image (`ubuntu:14.04`) and services that are required for testing.

- ✎ `image` specifies the name of a Docker image that is present in a local Docker repository, or a repository that can be found at Docker Hub.
- ✎ `services` are separate images that run at the same time and are linked to the build. Common services include `postgres` or `mysql` for data access.

The internal configuration of the script environment is controlled via the `before_script`. Before script is a set of commands that will be executed prior to all other running jobs. For this project, we will use it to configure the environment by setting up a `virtualenv` for the code and downloading project dependencies.

GitLab CI allows three types of jobs that can run (via the "types" argument): `build`, `test`, and `deploy`.

1. Build jobs are used to determine if the project successfully builds.
2. If build jobs execute successfully, then test jobs are executed which can be used to execute unit, integration, and functional tests.
3. If test jobs execute successfully, then deploy jobs are executed, which can be used to prepare and deploy a new build of the software.

For this example project, only build and test jobs will be used.

- 49) As the `guru` user, create the following file that will serve as the foundation of this project's CI configuration and setup:

File: `~/mod10-example/.gitlab-ci.yml`

```
+ image: ubuntu:14.04
+
+ before_script:
+   - ./envprep.sh
+
+ types:
+   - build
+   - test
```

- 50) Commit the changes, and push to the code server:

```
$ git add .gitlab-ci.yml
```

```
$ git commit .gitlab-ci.yml -m "Added GitLab CI base configuration"
... output omitted ...
```

51) For builds to execute correctly, the referenced environment program must be created which will:

1. Ensure that a virtual environment is available
2. Activate the virtual environment
3. Download and install Python dependencies into the virtual environment
4. Performing other necessary configuration for the program prior to running other tasks

Create a script that performs the setup for the ubuntu:14.04 Docker image used for testing this project:

```
File: ~/mod10-example/envprep.sh
+ # Install virtual environment if not available
+ if command -v virtualenv > /dev/null; then
+     echo "virtualenv installed"
+ else
+     echo "virtualenv not installed, install before proceeding"
+     apt-get install python-virtualenv
+ fi
+
+ # Create and activate virtual environment for dependencies
+ if [ -d ./env/bin ]; then
+     echo "virtualenv exists, activate"
+     source ./env/bin/activate
+ else
+     echo "unable to locate virtual environment, create and activate"
+     virtualenv ./env
+     source ./env/bin/activate
+ fi
+
+ # Install project dependencies
+ pip install --trusted-host=server1.example.com -r ./requirements.txt
```

52) Set the new script executable and commit the change:

```
$ chmod +x envprep.sh
$ git add envprep.sh
$ git commit envprep.sh -m "Script to build Python virtual environment"
. . . output omitted . . .
```

53) Modify the `gitlab-ci.yml` configuration to include a job that runs the unit tests for the project. Add the code in the following listing at the bottom of the existing file:

```
File: ~/modl0-example/.gitlab-ci.yml
+ modl0_unittest:
+   type: test
+   script:
+     - source ./env/bin/activate
+     - python -m unittest tests.test_modl0
```

54) Commit the changes:

```
$ git commit .gitlab-ci.yml -m "Added unittest task to CI"
. . . output omitted . . .
```

55) Push the accumulated commits to the remote Git repository:

```
$ git push gitlab master
. . . output omitted . . .
```

This will kick off a new build attempting to run the unit tests. Refreshing the CI project web page will show the job has been dispatched to the project runner.

56) Use the GitLab CI interface to **examine** the commit and associated build. **Notice** that it is failing because the CI server is trying to clone the project code into the container, but the container lacks the `git` command.

57) Create a `Dockerfile` to build an image that includes the `git` command:

File: ~/mod10-example/Dockerfile

```
+ FROM ubuntu:14.04
+ # Install git for GitLab CI tests
+ RUN apt-get update && apt-get install -y --force-yes git
```

58) Build the new image as root:

```
$ sudo docker build --tag ubuntu:14.04-mod10 .
Sending build context to Docker daemon 219.6 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
---> 8251da35e7a7
Step 1 : RUN apt-get install git -y
---> Running in eebcd682d01f
. . . snip . . .
Processing triggers for ureadahead (0.100.0-16) ...
---> abaf7cf188fc
Removing intermediate container eebcd682d01f
Successfully built abaf7cf188fc
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu               14.04-mod10        abaf7cf188fc       About a minute ago  226 MB
. . . snip . . .
```

59) Modify the .gitlab-ci.yml so that the Ci server uses the new image:

File: ~/mod10-example/.gitlab-ci.yml

```
→ image: ubuntu:14.04-mod10
```

60) Spawn a new build with the new image, by committing the Dockerfile and modified .gitlab-ci.yml to the source code repository:

```
$ git add Dockerfile .gitlab-ci.yml
$ git commit -m "Build and use custom image that includes Git"
[master 7b77e71] Added Dockerfile and updated .gitlab-ci.yml to use custom image
2 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 Dockerfile
$ git push gitlab master
```


. . . output omitted . . .

- 61) Again, use the GitLab CI web interface to **examine** the commit and associated build. **Notice** that it is still failing because the `envprep.sh` is trying to install the Python `virtualenv` package, but the minimal image lacks the correct repo.

Modify the Dockerfile so the image includes the needed items:

```
File: ~/mod10-example/Dockerfile
FROM ubuntu:14.04-mod10

# Install git for GitLab CI tests
RUN apt-get update && apt-get install -y --force-yes git

+ # Install python, pip, and virtualenv
+ ENV PIP_TRUSTED_HOST=server1.example.com \
+     PIP_INDEX_URL=http://server1.example.com/packages/simple
+ RUN apt-get install -y --force-yes python-pip python python-dev && \
+     pip install virtualenv
```

- 62) Build the new image and commit the change the change to git:

```
$ sudo docker build --tag ubuntu:14.04-mod10 .
. . . snip . . .
Installing collected packages: virtualenv
Successfully installed virtualenv
Cleaning up...
---> 5b6eb69ffff1f
Removing intermediate container 66a20d780096
Successfully built 5b6eb69ffff1f
$ git commit Dockerfile -m "Build image with Python and virtualenv"
[master 1aca5c3] Build image with Python and virtualenv
1 file changed, 9 insertions(+), 1 deletion(-)
```

- 63) Push the commit to GitLab (which will trigger another build):

```
$ git push gitlab master
```

Use the GitLab CI web interface to **examine** the build. This time it should pass!

Adding a CI Runner for Functional Testing

- 64) Knowing that the unit tests are passing correctly provides a great degree of comfort, but only assures us that the components of the program are working as expected. Equally important is to ensure that the functional tests are also passing. Green functional tests provide a degree of confidence that the program is working according to specification and are an integral part of any testing strategy.

Create a second CI job to execute the functional test suite by appending the following to the bottom of the `.gitlab-ci.yml` config file:

```
File: ~/modl0-example/.gitlab-ci.yml
modl0_unittest:
  type: test
  script:
    - source ./env/bin/activate
    - python -m unittest tests.test_modl0
+ modl0_func_test:
+   type: test
+   script:
+     - source ./env/bin/activate
+     - ./ftests.sh
```

Similar to the first job, this attempts to activate the virtualenv (with dependencies) and run a set of tests. Unlike the unit tests, however, the functional tests require a functional server to connect to in order to successfully run the tests.

There are many ways that this problem could be solved. One option might be to build two separate docker containers, one with the server source code and the second with the test client. Another might be to configure and launch the server as a background daemon in the container and then run the testing client. The solution used in the lab is to use a shell script, `ftests.sh`, to manage the functional test process.

- 65) Create the `ftests.sh` shell scripts with the following content:

```
File: ~/mod10-example/ftests.sh
```

```
+ #!/bin/bash
+ python ./mod10/server.py &
+ sleep 1
+ python ftests.py
+ testexit=$?
+ pkill python
+ exit $testexit
```

The shell script:

1. Starts the server as a background process, and then pauses the script for 1 second (this is done to allow the server to initialize and avoid race conditions).
2. Launches the functional test client (ftests.py), which will run the test suite.
3. Capture the ftests.py exit code so that it can be returned at the end of the script. This is necessary so that it can signal to the test runner whether the tests passed or failed.
4. Shuts down the server (using pkill).
5. Exits with the exit value of the functional test client.

66) Set permissions, commit and push:

```
$ chmod +x ftests.sh
$ git add ftests.sh .gitlab-ci.yml
$ git commit -m "Added functional test job and management script to CI"
2 files changed, 13 insertions(+)
create mode 100755 ftests.sh
$ git push gitlab master
```

67) Again **examine** the GitLab CI entry for the new commit and associated builds. Both unit and functional tests should now be passing (listing status of 'success').

At this point, the suite of tests will be run after every commit, and the CI server could now be configured (see services link in web interface) to provide updates via a number of services if something has been broken.

Bonus: Using Branches and Verifying Breakage is Quickly Reported

68) Create a "feature" branch to do new work and check it out:

```
$ git branch guru/mod10_enhancements
$ git checkout guru/mod10_enhancements
$ git branch
* guru/mod10_enhancements
master
```

69) Edit the existing mod10.py file (which contains the ccnum_isvalid source, and modify the line where the luhn_digits are calculated:

File: mod10/mod10.py	
→	off_digits = [int(ccdigit) for ccdigit in ccstr[-1::-2]] luhn_digits = [int(ccdigit)*23 for ccdigit in ccstr[-2::-2]] scaled_luhn_digits = map(lambda i: i-9 if i > 9 else i, luhn_digits)

If multiplying by two is good, then multiplying by three must be better. :)

70)

```
$ git commit mod10/mod10.py -m "Enhancements to Luhn algorithm"
$ git push gitlab guru/mod10_enhancements
Counting objects: 7, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 427 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To http://git.example.com:8000/dev3/mod10-example-station3.git
 * [new branch]      guru/mod10_enhancements -> guru/mod10_enhancements
```

71) Return to the Git Lab CI web interface and **examine** the latest commit under the 'All Commits' tab. It should report an status of 'Failed' for the build.

When there are multiple Git Lab branches, the build status will be neatly separated by branch. This makes it convenient to see which branches may be passing or failing the test suite. Then you can further drill down into the build to get additional details about which jobs and tests are failing.

The modifications made represent a fundamental change to the algorithm. For that reason, both the functional and the unit tests are both failing. If the program were more complex, you can break unit tests into multiple jobs in order to get a high level overview of which groups of tests are failing. Clicking on the Build ID link further lets us drop down and review the logs from the test run, providing still further detail about where the failures occur.

The reporting and automation tools that continuous integration brings make it a valuable tool in the arsenal of any organization that works with software. When coupled with other best practices, such as test driven development, continuous integration can improve the quality and compliance of code. Because tests can be run after every commit, it becomes much easier to find bugs and breaks earlier in development, and patch them more quickly.