# Docker Administration and Operations Advanced

## Exercise Guide

EG4199C-001

# Exercise handbook

## Introduction

The following exercise book is designed to assist you with the exercises in the Advanced Docker Topics training course. The book contains all the exercises that your instructor will present during the training session as well the solutions to each exercises.

The exercises are designed in a way so that you should be able to follow each step by step instruction to complete the task required. Your instructor will demonstrate each step to give you an overview of what the task involves. On the power point slides, each step will be listed to guide you along. However in this exercise handbook, you will also see the solutions for each step outlined. For example, if a step requires you to enter a particular command, the command will be listed for you along with a sample of what output to expect on your terminal. Where necessary, you will see explanations of each step and why that step was necessary to complete the overall task. This is not to say that the solution provided is the only way to complete the task. The solution is designed so that students will have a reference they can refer to if they are stuck on a particular exercise.

# Exercise 1.1 - Start and Stop Docker

**1) Stop the Docker daemon which should be running as a service**

The command to run is `sudo service docker stop`

```
docker@52.26.42.249 ~: sudo service docker stop
docker stop/waiting
docker@52.26.42.249 ~:
```

**2) Run `docker version`. What do you notice on the output?**

You will notice that the Docker client version is displayed but the Docker server version cannot be retrieved. This is because our Docker client can no longer connect to the daemon because it is no longer running. We stopped it in step 1.

```
docker@52.26.42.249 ~: docker version
Client version: 1.6.2
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 7c8fca2
OS/Arch (client): linux/amd64
FATA[0000] Cannot connect to the Docker daemon. Is 'docker -d' running on
this host?
```

**3) Now start the daemon again but this time use the docker client and specify the daemon command**

`sudo docker daemon &`

We run the Docker client and specify `daemon` for daemon mode. The "&" is so we can get out terminal back. You will notice the Daemon log being written on the console.

```
docker@52.26.42.249 ~: sudo docker daemon &
[1] 12135
docker@52.26.42.249 ~: INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] -job init_networkdriver() = OK (0)
WARN[0000] Your kernel does not support cgroup swap limit.
INFO[0000] Loading containers: start.
.....................................................
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.6.2 7c8fca2; execdriver: native-0.2;
graphdriver: aufs
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
```

**4) Run `docker version` again and verify that you can see the server version**

This time we should be able to see the server version as our daemon is running

```
docker@52.26.42.249 ~: docker version
Client version: 1.6.2
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 7c8fca2
OS/Arch (client): linux/amd64
INFO[0052] GET /v1.18/version
INFO[0052] +job version()
INFO[0052] -job version() = OK (0)
Server version: 1.6.2
Server API version: 1.18
Go version (server): go1.4.2
Git commit (server): 7c8fca2
OS/Arch (server): linux/amd64
docker@52.26.42.249 ~:
```

**5) Find the process ID of Docker and us it to stop the Docker daemon**

`sudo kill $(pidof docker)`

We run `pidof docker` as an easy way to retrieve the process ID of our Docker daemon. The output is fed into `kill` as a parameter

```
docker@52.26.42.249 ~: sudo kill $(pidof docker)
INFO[0087] Received signal 'terminated', starting shutdown of docker...
docker@52.26.42.249 ~: INFO[0087] -job
serveapi(unix:///var/run/docker.sock) = OK (0)

[1]+  Done                    sudo docker daemon
docker@52.26.42.249 ~:
```

**6) Start the daemon again using the service command**

`sudo service docker start`

```
docker@52.26.42.249 ~: sudo service docker start
docker start/running, process 12254
docker@52.26.42.249 ~:
```

## Exercise 1.2 - Setup daemon logging

**1) Stop the Docker service or process that is currently running**

At this point in the course you should be running the Docker daemon as a service. To stop we use
`sudo service docker stop`

```
$ sudo service docker stop
[sudo] password for johnnytu:
docker stop/waiting
```

If you are not running as a service use
`sudo kill $(pidof docker)`

**2) Start Docker in daemon mode and specify the debug logging level**

`sudo docker daemon --log-level=debug`

```
$ sudo docker daemon --log-level=debug &
[1] 4940
$ DEBU[0000] waiting for daemon to initialize
DEBU[0000] Using graph driver aufs
DEBU[0000] Migrating existing containers
DEBU[0000] Creating images graph
DEBU[0000] Restored 276 elements
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
DEBU[0000] Registering GET, /containers/{name:.*}/export
DEBU[0000] Registering GET, /containers/{name:.*}/json
DEBU[0000] Registering GET, /_ping


...
...
DEBU[0000] Loaded container
cdcec09ee42b317144c2bc3857d666bff59fa7f2c1bcf800c01e195fe03ddde8
DEBU[0000] Loaded container
dadbdf53aef01cbe5529cafafe3a36acc59eff8cb61d7737f552c4f4ce9e4f1f
DEBU[0000] Loaded container
f638d4c945040bc63f05672c3a76e01c68eb30ba17d6fe3b32820fb9e4c0405a
DEBU[0000] Restarting containers...
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.6.0 4749651; execdriver: native-0.2;
graphdriver: aufs
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
```

**3) Run a few Docker commands and observe the log output**

Let's try `docker ps`

```
$ docker ps
DEBU[0250] Calling GET /containers/json
INFO[0250] GET /v1.18/containers/json
INFO[0250] +job containers()
INFO[0250] -job containers() = OK (0)
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
$
```

Notice the debug log entry.

Now let's try `docker images`

```
$ docker images
DEBU[0435] Calling GET /images/json
INFO[0435] GET /v1.18/images/json
INFO[0435] +job images()
INFO[0436] -job images() = OK (0)
REPOSITORY                                  TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
107.170.229.60/johnny/myubuntu              1.1                 bdcf2496b10d
3 days ago          188.3 MB
107.170.229.60/johnny/myubuntu              1.0                 af35aebc0cd1
3 days ago          188.3 MB
...
```

**4) Stop the Docker daemon**

`sudo kill $(pidof docker)`

```
$ sudo kill $(pidof docker)
INFO[0713] Received signal 'terminated', starting shutdown of docker...
johnnytu@docker-ubuntu:~$ DEBU[0713] starting clean shutdown of all
containers...
INFO[0713] -job serveapi(unix:///var/run/docker.sock) = OK (0)
[1]+  Done                    sudo docker daemon --log-level=debug
$
```

**5) Start it again and change the log level to info.**

`sudo docker daemon --log-level=info &`

```
$ sudo docker daemon --log-level info &
[1] 5187
$ INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] -job init_networkdriver() = OK (0)
WARN[0000] Your kernel does not support cgroup swap limit.
INFO[0000] Loading containers: start.
.....................
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.6.0 4749651; execdriver: native-0.2;
graphdriver: aufs
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
$
```

**6) Run some Docker commands and observe the log output**

Let's try `docker ps` again

```
$ docker ps
INFO[0121] GET /v1.18/containers/json
INFO[0121] +job containers()
INFO[0121] -job containers() = OK (0)
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
$
```

This time you'll notice there are no debug entries in the log

## Exercise 1.3 – More logging

**1) Stop the Docker daemon**

Run `sudo service docker stop`

**2) Open the `/etc/default/docker` file**

**3) Configure the DOCKER_OPTS flag to add the log level and specify info as the level**
DOCKER_OPTS="--log-level=info"

Your `/etc/default/docker` file should look something like the one below. Note that in your setup, the DOCKER_OPTS variable may already have flags configured. Do not remove the existing flags. Just add `--log-level` to it.

```
# Docker Upstart and SysVinit configuration file
# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"
# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
DOCKER_OPTS="-H unix:///var/run/docker.sock -H tcp://127.0.0.1:2375
--log-level=info"
```

**4) Start the Docker daemon using the service command**

`sudo service docker start`

**5) Check the log file at /var/log/upstart/docker.log and notice the info level output**

As we are not logged in as root, we have to use `sudo` to be able to `cat` the file

```
docker@52.26.42.249 ~: sudo cat /var/log/upstart/docker.log
INFO[971450] Received signal 'terminated', starting shutdown of docker...
INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock, tcp://127.0.0.1:2375)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Listening for HTTP on tcp (127.0.0.1:2375)
INFO[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify IF
YOU DON'T KNOW WHAT YOU'RE DOING /!\
INFO[0000] -job init_networkdriver() = OK (0)
WARN[0000] Your kernel does not support cgroup swap limit.
INFO[0000] Loading containers: start.
.................................................
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.6.2 7c8fca2; execdriver: native-0.2;
graphdriver: aufs
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
INFO[0512] Received signal 'terminated', starting shutdown of docker...
INFO[0512] -job serveapi(unix:///var/run/docker.sock, tcp://127.0.0.1:2375)
= OK (0)
INFO[0000] +job serveapi(unix:///var/run/docker.sock, tcp://127.0.0.1:2375)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Listening for HTTP on tcp (127.0.0.1:2375)
INFO[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify IF
YOU DON'T KNOW WHAT YOU'RE DOING /!\
INFO[0000] +job init_networkdriver()
INFO[0000] -job init_networkdriver() = OK (0)
WARN[0000] Your kernel does not support cgroup swap limit.
INFO[0000] Loading containers: start.
.................................................
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.6.2 7c8fca2; execdriver: native-0.2;
graphdriver: aufs
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
docker@52.26.42.249 ~:
```

**6) Run some simple command such as docker ps and check the log file again**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE          COMMAND        CREATED        STATUS
PORTS          NAMES
docker@52.26.42.249 ~:

docker@52.26.42.249 ~: sudo cat /var/log/upstart/docker.log
INFO[971450] Received signal 'terminated', starting shutdown of docker...
INFO[0000] +job init_networkdriver()
...
...
...
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
INFO[0108] GET /v1.18/containers/json
INFO[0108] +job containers()
INFO[0108] -job containers() = OK (0)
docker@52.26.42.249 ~:
```

In step 5), when we start the daemon, the last log line was
"INFO[0000] Daemon has completed initialization"

Now we can see the line where a request was made for
"GET /v1.18/containers/json". This was the API call made to get containers as part of the `docker ps` command

**7) Change the log level to warn and restart the service**

First, open the `/etc/default/docker` file. Remember to use `sudo`, otherwise your text editor may not allow you to save the changes.
The we modify DOCKER_OPTS.
The DOCKER_OPS line should be similar to

```
DOCKER_OPTS="-H unix:///var/run/docker.sock -H tcp://127.0.0.1:2375
--log-level=warn"
```

Then restart the service with `sudo service docker restart`

**8) Run some commands, check the log file again and notice the difference in output**

You will only see a WARN line. You will still see the previous log entries with the INFO output. But from now on only WARN entries and higher will appear.

# Exercise 1.4 – Sockets

**1) Stop your Docker daemon**

Run `sudo service docker stop`

**2) Try to run the following commands**

a) `docker`
b) `docker version`
c) `docker ps`

**3) Notice the error message that appears on b) and c) and how it points to the socket file (/var/run/docker.sock)**
The first command, was just to run the Docker client. This will work fine even when the Daemon is not running.
On b) and c), the client needs to make an API call to the daemon, which is why the error occurs since our daemon is not running.

```
ubuntu@node-0:~$ docker version
Client:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:30:23 2016
 OS/Arch:      linux/amd64
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
ubuntu@node-0:~$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
```

## Exercise 1.5 – Listen on TCP

**1) Open `/etc/default/docker`**

**2) Configure the DOCKER_OPTS variable to add the option for docker to listen on a tcp socket on any network interface**
`DOCKER_OPTS="-H tcp://0.0.0.0:2375"`

Remember, do not delete any other flags configured on DOCKER_OPTS. Just add to them.

**3) Start the Docker service**

```
sudo service docker start
```

**4) Try to run a command such as `docker version` and `docker ps`**

```
ubuntu@node-0:~$ docker version
Client:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:30:23 2016
 OS/Arch:      linux/amd64
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
ubuntu@node-0:~$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
```

Note: Your instructor may have already setup Docker to listen on TCP and the unix socket on your environment. If this is the case, you will not get the errors

**5) Notice you get the same error message as before (The error message from step 2 in Exercise 11.1)**

**6) Why do you think this is?**

We have configured our Daemon to listen only on a TCP socket. However, the Docker client does not know this and by default it tries to connect on the Unix socket.

Note: Your instructor may have already setup Docker to listen on TCP and the unix socket on your environment. If this is the case, you will not get the errors

# Exercise 1.6 – Specify host on client

**1) Run a docker command with the -H option to connect to your daemon which is now listening on a TCP socket**
`docker -H tcp://localhost:2375 ps`

```
docker@52.26.42.249 ~: docker -H tcp://localhost:2375 ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
docker@52.26.42.249 ~:
```

**2) Now set the DOCKER_HOST environment variable to `tcp://localhost:2375`**
`export DOCKER_HOST="tcp://localhost:2375"`

This is a more effective method because we won't have to specify the `-H` flag everytime

**3) Run some docker commands without the `-H` flag. Verify that there are no error messages**

Let's try `docker ps` again.

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
docker@52.26.42.249 ~:
```

# Exercise 1.7 – Connect to remote daemon

**1) Pair up with the student next to you**

**2) Get the IP address of their AWS host**

Let's pretend that we are paired up and the other students IP is 52.26.37.124.

**3) Connect your Docker client to the daemon running on the other students host**

To connect, we simply set the DOCKER_HOST variable to the Docker daemon running on the other host.

`export DOCKER_HOST="tcp://52.26.37.124:2375"`

Note: Your partner's Docker daemon needs to be listening on TCP and on the 0.0.0.0 interface.
(i.e. `DOCKER_OPTS=" -H tcp://0.0.0.0:2375"`)

**4) Run `docker ps -a` to see what containers the student has run**

**5) Run a new NGINX container in detached mode**

`docker run -d nginx`

**6) Have to other student verify that the container is running on their host**

Your partner should run `docker ps` and see an NGINX container running

**7) Now disconnect the client from the other students host and swap roles. Let the other student connect their Docker client to your daemon**

To disconnect, you need to unset the DOCKER_HOST variable.

```
docker@52.26.42.249 ~: unset DOCKER_HOST
```

# Exercise 1.8 – Multiple sockets

**Note:** The Docker daemon on the provided training environment may already be configured to listen on both TCP and a Unix socket. If this is the case, there is no need to do this exercise. Your instructor will advise you accordingly.

**1) From the end of the last exercise, your Docker client has been disconnected from the daemon on the remote host. Try and run a command such as docker ps. Notice the error since the client is trying to use the unix socket**

```
docker@52.26.42.249 ~: docker ps
FATA[0000] Get http:///var/run/docker.sock/v1.18/containers/json: dial unix
/var/run/docker.sock:
no such file or directory. Are you trying to connect to a TLS-enabled
daemon without TLS?
docker@52.26.42.249 ~:
```

**2) Open `/etc/default/docker` and configure Docker to listen on both the TCP and unix socket**
We modify the `DOCKER_OPTS` variable to include another `-H` option and then specify the Unix socket.
Your `DOCKER_OPTS` should now have two `-H` options configured such as the example shown below

`DOCKER_OPTS="-H tcp://0.0.0.0:2375 –H unix:///var/run/docker.sock"`

**3) Restart the Docker daemon**

Run `sudo service docker restart`

**4) Now run `docker ps` again and verify that it works**

# Exercise 2.1 – Create the CA

**1) First we need to setup our folder to store all the keys. Create a folder called docker-ca**

**2) Run `chmod 0700 docker-ca` to set the correct permission on the folder**

**3) Go into the folder**

The output for step 1 to 3 would be as shown below

```
docker@52.26.42.249 ~: mkdir docker-ca
docker@52.26.42.249 ~: chmod 0700 docker-ca/
docker@52.26.42.249 ~: cd docker-ca/
docker@52.26.42.249 ~/docker-ca:
```

**4) Now create the Certificate Authority private key**
Run `openssl genrsa -aes256 -out ca-key.pem 2048`

```
docker@52.26.42.249 ~/docker-ca: openssl genrsa -aes256 -out ca-key.pem
2048
Generating RSA private key, 2048 bit long modulus
..................+++
...................+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
docker@52.26.42.249 ~/docker-ca:
```

You will be prompted for a pass phrase. Make sure you remember your pass phrase.

**5) Using the CA private key, create the CA certificate**
Run `openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem`

You will get prompted to enter in details. You can skip the details

```
docker@52.26.42.249 ~/docker-ca: openssl req -new -x509 -days 365 -key
ca-key.pem -sha256 -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
docker@52.26.42.249 ~/docker-ca:
```

# Exercise 2.2 – Setup server key

**1) Create the server private key**

Run the command below
`openssl genrsa -out server-key.pem 2048`

```
docker@52.26.42.249 ~/docker-ca: openssl genrsa -out server-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................
.............
.....................................................................
.......+++
.....................+++
e is 65537 (0x10001)
docker@52.26.42.249 ~/docker-ca:
```

**2) Create the certificate signing request (CSR) and specify the IP address of the machine that the Docker daemon is running on**
Run
`openssl req -subj "/CN=<host name>" -new -key server-key.pem -out server.csr`

Replace <host name> with the IP address of your machine.

```
docker@52.26.42.249 ~/docker-ca: openssl req -subj "/CN=52.26.42.249" -new
-key server-key.pem -out server.csr
docker@52.26.42.249 ~/docker-ca:
```

**3) So far you should have the following files in your docker-ca folder**
a) CA private key – `ca-key.pem`
b) CA certificate – `ca.pem`
c) Server private key – `server-key.pem`
d) Server CSR – `server.csr`

```
docker@52.26.42.249 ~/docker-ca: ls
ca-key.pem  ca.pem  server.csr  server-key.pem
docker@52.26.42.249 ~/docker-ca:
```

# Exercise 2.3 – Sign the server key

**1) Create a file called extfile.cnf**

```
touch extfile.cnf
```

**2) Open the file and using the `subjectAltName` extension to specify the IP addresses to allow connections to. You will need to specify the IP of your Docker daemon host and also 127.0.0.1**

Your extfile.cnf should have the following line
```
subjectAltName = IP:<host IP>,IP:127.0.0.1
```

**3) Sign the server key**
```
openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile extfile.cnf
```

Remember your pass phrase

```
docker@52.26.42.249 ~/docker-ca: openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=<host name>
Getting CA Private Key
Enter pass phrase for ca-key.pem:
docker@52.26.42.249 ~/docker-ca:
```

## Exercise 2.4 – Create client key

**1) Create the client private key. Call it client-key.pem to make is easy to distinguish from our server key**
```
openssl genrsa -out client-key.pem 2048
```

```
docker@52.26.42.249 ~/docker-ca: openssl genrsa -out client-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................
....+++
.......+++
e is 65537 (0x10001)
```

**2) Create the client CSR using the following command**
```
openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr
```

```
docker@52.26.42.249 ~/docker-ca: openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr
docker@52.26.42.249 ~/docker-ca: ls
ca-key.pem  ca.pem  ca.srl  client.csr  client-key.pem  extfile.cnf
server-cert.pem  server.csr  server-key.pem
docker@52.26.42.249 ~/docker-ca:
```

# Exercise 2.5 – Sign the client key

**1) Create the extension file to make the key suitable for client authentication**

We don't need the previous line in our extfile.cnf file anymore so we can override it

```
echo extendedKeyUsage = clientAuth > extfile.cnf
```

**2) Sign the clients public key**

```
openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out
client-cert.pem -extfile extfile.cnf
```

```
docker@52.26.42.249 ~/docker-ca: openssl x509 -req -days 365 -in client.csr
-CA ca.pem -CAkey ca-key.pem -CAcreateserial -out client-cert.pem -extfile
extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
docker@52.26.42.249 ~/docker-ca:
```

Your folder should now look like:

```
docker@52.26.42.249 ~/docker-ca: ls
ca-key.pem  ca.pem  ca.srl  client-cert.pem  client.csr  client-key.pem
extfile.cnf  extfile.cnf  server-cert.pem  server.csr  server-key.pem
```

# Exercise 2.6 – Securing the keys

**1) In the docker-ca folder, make the CA, server and client private keys readable only to yourself.**

```
chmod -v 0400 ca-key.pem client-key.pem server-key.pem
```

```
docker@52.26.42.249 ~/docker-ca: chmod -v 0400 ca-key.pem client-key.pem
server-key.pem
mode of ‘ca-key.pem’ changed from 0664 (rw-rw-r--) to 0400 (r--------)
mode of ‘client-key.pem’ changed from 0664 (rw-rw-r--) to 0400 (r--------)
chmod: cannot access ‘server-key.pem’: No such file or directory
failed to change mode of ‘server-key.pem’ from 0000 (---------) to 0000
(---------)
docker@52.26.42.249 ~/docker-ca:
```

**2) Remove write access to all certificates.**

```
chmod -v 0444 ca.pem server-cert.pem client-cert.pem
```

```
docker@52.26.42.249 ~/docker-ca: chmod -v 0444 ca.pem server-cert.pem
client-cert.pem
mode of ‘ca.pem’ changed from 0664 (rw-rw-r--) to 0444 (r--r--r--)
mode of ‘server-cert.pem’ changed from 0664 (rw-rw-r--) to 0444 (r--r--r--)
chmod: cannot access ‘client-cert.pem’: No such file or directory
failed to change mode of ‘client-cert.pem’ from 0000 (---------) to 0000
(---------)
docker@52.26.42.249 ~/docker-ca:
```

**3) Create folder `/etc/docker` if it does not already exist.**

**4) Make yourself the owner of the /etc/docker folder.**
`sudo chown <username>:docker /etc/docker`

In our example, the username is docker. Once we've run the command above, we can check the permissions on the folder by running `ls -l`

```
docker@52.26.42.249 ~/docker-ca: sudo chown docker:docker /etc/docker
docker@52.26.42.249 ~/docker-ca: ls -l
total 44
-r-------- 1 docker docker 1766 Jun 16 03:32 ca-key.pem
-r--r--r-- 1 docker docker 1229 Jun 16 04:04 ca.pem
-rw-rw-r-- 1 docker docker   17 Jun 16 05:03 ca.srl
-rw-rw-r-- 1 docker docker 1078 Jun 16 05:03 client-cert.pem
-rw-rw-r-- 1 docker docker  887 Jun 16 04:54 client.csr
-r-------- 1 docker docker 1679 Jun 16 04:54 client-key.pem
-rw-rw-r-- 1 docker docker   46 Jun 16 04:38 extfile.cnf
-rw-rw-r-- 1 docker docker   30 Jun 16 05:03 extfile.cnf
-r--r--r-- 1 docker docker 1090 Jun 16 04:50 server-cert.pem
-rw-rw-r-- 1 docker docker  895 Jun 16 04:47 server.csr
-rw-rw-r-- 1 docker docker 1675 Jun 16 04:06 server-key.pem
docker@52.26.42.249 ~/docker-ca:
```

**5) Set read, write and execution permissions for yourself only on the /etc/docker folder.**
`sudo chmod 700 /etc/docker`

**6) Copy the CA key, server key and server certificate to the `/etc/docker` folder.**
sudo cp ~/docker-ca/{ca,server-key,server-cert}.pem /etc/docker

## Exercise 2.7 – Enable TLS on the daemon

**1) Open the `/etc/default/docker` file**

**2) Change the DOCKER_OPTS variable to the following**

When we use TLS, the standard TCP port is 2376. Without TLS, the standard port is 2375. You will want to keep the existing flags defined for DOCKER_OPTS. The example below shows just the flags you should add. Here we are specifying the Certificate Authority's certificate (ca.pem), the server certificate (server-cert.pem) and server key (server-key.pem)

(note: this needs to be done on one line)

```
DOCKER_OPTS="-H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/etc/docker/ca.pem
--tlscert=/etc/docker/server-cert.pem --tlskey=/etc/docker/server-key.pem"
```

**3) Restart the docker service**

```
sudo service docker restart
```

## Exercise 2.8 – Run the client

**1) Go into your docker-ca folder and run:**

```
docker --tlsverify --tlscacert=ca.pem --tlscert=client-cert.pem --tlskey=client-key.pem -H tcp://127.0.0.1:2
376 ps
```

```
docker@52.26.42.249 ~/docker-ca: docker --tlsverify --tlscacert=ca.pem
--tlscert=client-cert.pem --tlskey=client-key.pem -H=tcp://127.0.0.1:2376
ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
```

**2) Verify that the client is able to talk to the daemon. Try a few more commands**

Let's try a simple `docker version`

```
docker@52.26.42.249 ~/docker-ca: docker --tlsverify --tlscacert=ca.pem
--tlscert=client-cert.pem --tlskey=client-key.pem -H=tcp://127.0.0.1:2376
version
Client version: 1.6.2
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 7c8fca2
OS/Arch (client): linux/amd64
Server version: 1.6.2
Server API version: 1.18
Go version (server): go1.4.2
Git commit (server): 7c8fca2
OS/Arch (server): linux/amd64
docker@52.26.42.249 ~/docker-ca:
```

## Exercise 2.9 – Set environment variables

**1) Create the .docker folder in your home directory**

```
mkdir ~/.docker
```

**2) In your docker-ca folder Copy `ca.pem`, `client-cert.pem` and `client-key.pem` into the `.docker` folder**

**3) Cd into `~/.docker` and rename `client-cert.pem` to `cert.pem`**

**4) Rename `client-key.pem` to `key.pem`**

The output of the first 4 steps should look like the following

```
docker@52.26.42.249 ~: mkdir ~/.docker
docker@52.26.42.249 ~:
docker@52.26.42.249 ~: cd docker-ca/
docker@52.26.42.249 ~/docker-ca: cp ca.pem client-cert.pem client-key.pem
~/.docker/
docker@52.26.42.249 ~/docker-ca: cd ~/.docker/
docker@52.26.42.249 ~/.docker: ls
ca.pem  client-cert.pem  client-key.pem
docker@52.26.42.249 ~/.docker:
docker@52.26.42.249 ~/.docker: mv client-cert.pem cert.pem
docker@52.26.42.249 ~/.docker: mv client-key.pem key.pem
docker@52.26.42.249 ~/.docker: ls
ca.pem  cert.pem  key.pem
docker@52.26.42.249 ~/.docker:
```

**5) Set the `DOCKER_HOST` environment variable to 127.0.0.1**

```
export DOCKER_HOST="tcp://127.0.0.1:2376"
```

**6) Set the `DOCKER_TLS_VERIFY` variable to 1**

```
export DOCKER_TLS_VERIFY=1
```

**7) Now run a few docker commands with specifying any flags and verify that it works**

Try:

- `docker ps`
- docker images

```
docker@52.26.42.249 ~/.docker: docker ps
CONTAINER ID        IMAGE           COMMAND         CREATED         STATUS
PORTS           NAMES
docker@52.26.42.249 ~/.docker: docker images
REPOSITORY                    TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
javahelloworld                1.0                 2c7d99476900        3
days ago          587.3 MB
johnnytu/myimage              1.0                 d8d368a95d82        3
days ago          257.6 MB
...
...
myimage                       latest              175d74beca12        3
days ago          215.4 MB
docker@52.26.42.249 ~/.docker:
```

**8) Clean up your setup by removing all TLS settings**

**a) Open `/etc/default/docker` and remove all `--tls` options from `DOCKER_OPTS`**

**b) Change `-H` option to tcp://0.0.0.0:2375 and add another `-H` option with value `unix:///var/run/docker.sock`**

**c) Restart the Docker daemon**

**d) Run `unset DOCKER_HOST` and `unset DOCKER_TLS_VERIFY`**

The DOCKER_OPTS variable in your /etc/default/docker file should look like this after step b)

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --log-level=info -H
tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock"
```

# Exercise 3.1 – Setup the multi-host network

**1) Make sure you have no containers running on any of your nodes**

At this stage of the course, only the master node should have been used.

**2) Make sure you have the IP address of your Master Node**

In this example, our Master IP address is 192.241.227.195

**3) Follow steps 1, 2 and 3 on the previous slides to create your multi-host network**

The first thing to do is to setup our key-value store. We will use `Consul` for this and run it from a Docker container

Run the command:
`docker run -d -p 8500:8500 -h consul --name consul progrium/consul -server –bootstrap`

```
student@master:~$ docker run -d -p 8500:8500 -h consul --name consul
progrium/consul -server –bootstrap
Unable to find image 'progrium/consul:latest' locally
latest: Pulling from progrium/consul
3b4d28ce80e4: Pull complete
e5ab901dcf2d: Pull complete
30ad296c0ea0: Pull complete
3dba40dec256: Pull complete
f2ef4387b95e: Pull complete
.....
.....
Digest:
sha256:8cc8023462905929df9a79ff67ee435a36848ce7a10f18d6d0faba9306b97274
Status: Downloaded newer image for progrium/consul:latest
62507e05badecd058403581a8b335c510d27e6c15ba8b126510e51d1ca2cd77c
```

```
student@master:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  ...
STATUS          PORTS
NAMES
62507e05bade        progrium/consul     "/bin/start -server \xe2"  ...
Up 2 minutes     53/tcp, 53/udp, 8300-8302/tcp, 8400/tcp,
0.0.0.0:8500->8500/tcp, 8301-8302/udp   consul
```

Next, we need to configure the Docker Engine on both Node1 and Node2. We need to configure two things on the Docker Engine:

- Docker Engine to listen on TCP port 2375
- Use the Consul key-value store on our master node

**Switch over to your Node 1 VM**

We will open our `/etc/default/docker` file. Configure the DOCKER_OPTS variable to the following:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 -H tcp://0.0.0.0:2375 -H
unix:///var/run/docker.sock
--cluster-store=consul://192.241.227.195:8500/network
--cluster-advertise=eth0:2375"
```

For the --cluster-store option, make sure to specify the IP address of your master node. In our case the IP is 192.241.227.195.

After this save the file and restart the docker server with the command `sudo service docker restart`

**Now repeat the same process on your Node 2 VM.**

At the end of this exercise your Node 1 and Node 2 VMs should be discoverable by our `consul` container running on our **Master** VM

**Configure the overlay Network**

For this part, you can use either your Node 1 or Node 2 VM.

Run the following command:
`$ docker network create -d overlay --subnet 10.10.10.0/24 multinet`

```
student@node-1:~$ docker network create -d overlay --subnet 10.10.10.0/24
multinet
2435db48aeaf44065b28545ab6a8ddfc184d42160a8a428984ffe31a619fb449
student@node-1:~$ docker network ls
NETWORK ID          NAME                DRIVER
2435db48aeaf        multinet            overlay
2b4f7a3fa403        bridge              bridge
06ea9f238b31        none                null
01200ab2df58        host                host
student@node-1:~$
```

# Exercise 3.2 – Run container on multi-host network

**For this first part, Use your Node1 VM**

**1) Run an ubuntu container  on your multi-host network called c1**
`$ docker run -itd --name c1 --net multinet ubuntu:14.04`

```
student@node-1:~$ docker run -itd --name c1 --net multinet ubuntu:14.04
Unable to find image 'ubuntu:14.04' locally
14.04: Pulling from library/ubuntu
92ec6d044cb3: Pull complete
2ef91804894a: Pull complete
f80999a1f330: Pull complete
6cc0fc2a5ee3: Pull complete
Digest:
sha256:0844055d30c0cad5ac58097597a94640b0102f47d6fa972c94b7c129d87a44b7
Status: Downloaded newer image for ubuntu:14.04
c8e78a215841437a638efbf6eba37b2793acc1a3e8ba7613397719f7c9098521
```

**2) Check the network configuration of c1 and verify that you can see an eth0 and eth1 network**

`$ docker exec c1 ifconfig`

```
student@node-1:~$ docker exec c1 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:0a:0a:02
          inet addr:10.10.10.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe0a:a02/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1116 (1.1 KB)  TX bytes:648 (648.0 B)
eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1206 (1.2 KB)  TX bytes:648 (648.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
student@node-1:~$
```

**Now switch over to your Node2 VM**

**3) Run an NGINX container on your multi-host network called nginx. Do not specify any port mapping**

`$ docker run -d --name nginx --net multinet nginx`

```
student@node-2:~$ docker run -d --name nginx --net multinet nginx
2753b577a5a6581ba7f421b82962df52956090491c1d7e69f934482d6ee5bf36
```

**4) Verify that you can ping your c1 container**

```
$ docker exec nginx ping c1
```

```
student@node-2:~$ docker exec nginx ping c1
PING c1 (10.10.10.2): 56 data bytes
64 bytes from 10.10.10.2: icmp_seq=0 ttl=64 time=0.622 ms
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=0.581 ms
64 bytes from 10.10.10.2: icmp_seq=2 ttl=64 time=0.619 ms
64 bytes from 10.10.10.2: icmp_seq=3 ttl=64 time=0.599 ms
^C--- c1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.581/0.605/0.622/0.000 ms
root@2753b577a5a6:/#
```

**Now switch back to your Node1 VM**

**6) Verify that you can ping the nginx container**

```
$ docker exec c1 ping nginx
```

```
student@node-1:~$ docker exec nginx ping c1
PING nginx (10.10.10.3): 56 data bytes
64 bytes from 10.10.10.3: icmp_seq=0 ttl=64 time=0.592 ms
64 bytes from 10.10.10.3: icmp_seq=1 ttl=64 time=0.591 ms
64 bytes from 10.10.10.3: icmp_seq=2 ttl=64 time=0.609 ms
64 bytes from 10.10.10.3: icmp_seq=3 ttl=64 time=0.613 ms
^C--- nginx ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.581/0.605/0.622/0.000 ms
root@2753b577a5a6:/#
```

**7) Get terminal access into c1**

```
$ docker exec -it c1 bash
```

**8) Install curl**

```
$ apt-get install curl
```

**9) Use curl to make a request to the NGINX server running on Node2. You should notice that we can request for the NGINX welcome page despite our NGINX server running in a container on a different hosts without any port mapping**

```
$ curl nginx
```

```
root@c8e78a215841:/# curl nginx
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
root@c8e78a215841:/#
```

## Exercise 4.1 – Using signed images

**1) Login to your Docker Hub account on your terminal**

```
student@master:~$ docker login
Username: trainingteam
Password:
...
WARNING: login credentials saved in /home/student/.docker/config.json
Login Succeeded
student@master:~$
```

**2) Pull the latest Ubuntu image**

```
$ docker pull ubuntu:latest
```

**3) Create a new tag for the image using your Docker Hub username. Call your new repository trustedubuntu**

```
$ docker tag ubuntu:latest <username>/trustedubuntu:latest
```

**4) Push the image to Docker Hub**

```
$ docker push <username>/trustedubuntu:latest
```

**5) Login to your Docker Hub account on a browser and check to make sure you can find your new Repository and Tag**

**6) Enable content trust by setting the environment variable**

```
$ export DOCKER_CONTENT_TRUST=1
```

**7) Try and run a container using your trustedubuntu image**

```
student@master:~$ docker run -it trainingteam/ubuntu:latest
no trust data available
student@master:~$
```

**8) Notice the error message saying:**
**"no trust data available"**
This is because the image we are trying to run the container from is not signed.

## Exercise 4.2 – Sign an image

**1) At this stage, content trust should be enabled. Tag your `trustedubuntu` image again as 1.0**
`$ docker tag <username>/trustedubuntu:latest <username>/trustedubuntu:1.0`

**2) Push your new image tag to Docker Hub. You will be prompted for a root key passphrase and repository key passphrase. Pick a passphrase and make sure you remember it**
`$ docker push <username>/trustedubuntu:1.0`

```
student@master:~$ docker push trainingteam/trustedubuntu:1.0
The push refers to a repository [docker.io/trainingteam/trustedubuntu]
(len: 1)
ac6a7980c6c2: Image already exists
c00ef186408b: Image already exists
1.1: digest:
sha256:55c4bdbdd39f9cbfaab4692a3c03c39c3b3cdd6d4b4a9a4b7f6ff60c4a344db2
size: 2742
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system.
Please
choose a long, complex passphrase and be careful to keep the password and
the
key file itself secure and backed up. It is highly recommended that you use
a
password manager to generate the passphrase and keep it safe. There will be
no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id 8344fe3:
Repeat passphrase for new root key with id 8344fe3:
Enter passphrase for new repository key with id
docker.io/trainingteam/trustedubuntu (4b3c817):
Repeat passphrase for new repository key with id
docker.io/trainingteam/trustedubuntu (4b3c817):
Finished initializing "docker.io/trainingteam/trustedubuntu"
```

**3) Change directory into ~/.docker/trust/private/root_keys. Verify you can see your root key in here**

```
student@master:~$ cd ~/.docker/trust/private/
student@master:~/.docker/trust/private$ cd root_keys/
student@master:~/.docker/trust/private/root_keys$ ls
8344fe38e82ff775c7667178cf325b9bfb5b67e3aa16722b6b85416908c9b903_root.key
b92ad6fb897e5cb67463a383ff67aafb2b24d7a7ed2dadf7f6d2932ae21009d3_root.key
```

4) Change directory into ~/.docker/trust/private/tuf_keys/docker.io. You should see a folder named after your Docker Hub username and underneath that another folder called `trustedubuntu`.

5) Change directory into the `trustedubuntu` folder in step 4

```
student@master:~/.docker/trust/private/root_keys$ cd ..
student@master:~/.docker/trust/private$ cd tuf_keys/
student@master:~/.docker/trust/private/tuf_keys$ ls
docker.io
student@master:~/.docker/trust/private/tuf_keys$ cd docker.io/trainingteam/
student@master:~/.docker/trust/private/tuf_keys/docker.io/trainingteam$ ls
trustedubuntu
student@master:~/.docker/trust/private/tuf_keys/docker.io/trainingteam$ cd
trustedubuntu/
student@master:~/.docker/trust/private/tuf_keys/docker.io/trainingteam/tru
stedubuntu$ ls
3461d59645403d0e411e0702b369f57f1d80261a86f394416b16f63b09be72f2_snapshot.
key
4b3c817179d9c2016b1463afcef70806e9bdf9fa3e3770d8fceb5f21d0ac32ed_targets.k
ey
student@master:~/.docker/trust/private/tuf_keys/docker.io/trainingteam/tru
stedubuntu$
```

**6) You should see your repository keys in this folder**

```
student@master:~/.docker/trust/private/tuf_keys/docker.io/trainingteam/tru
stedubuntu$ ls
3461d59645403d0e411e0702b369f57f1d80261a86f394416b16f63b09be72f2_snapshot.
key
4b3c817179d9c2016b1463afcef70806e9bdf9fa3e3770d8fceb5f21d0ac32ed_targets.k
ey
```

**7) Run a container using your newly signed trustedubuntu:1.0 image**
`$ docker run -it trainingteam/trustedubuntu:1.0`

## Exercise 4.3 – Signed and unsigned tags

**Using your Master Node**

**1) Turn off content trust**

```
$ unset DOCKER_CONTENT_TRUST
```

**2) Run a container using the trustedubuntu:1.0 image from Exercise 4.2. Get terminal access into the container**

```
$ docker run -it <username>/trustedubuntu:1.0 bash
```

**3) Add a file called unsigned.txt**

```
student@master:~$ docker run -it trainingteam/trustedubuntu:1.0 bash
root@28e36e7433b8:/# touch unsigned.txt
```

**4) Exit the container**

```
root@28e36e7433b8:/# exit
```

**5) Commit the container as a new unsigned image with the 1.0 tag.**

```
student@master:~$ docker commit 28e36e7433b8 trainingteam/trustedubuntu:1.0
f0eed5f2fdef5042d55c8afefa3babe176270dc2a66fa490b2bbaf51d0519444
```

**6) Push the unsigned image to Docker Hub**

```
student@master:~$ docker push trainingteam/trustedubuntu:1.0
The push refers to a repository [docker.io/trainingteam/trustedubuntu]
(len: 1)
f0eed5f2fdef: Pushed
e9ae3c220b23: Image already exists
a6785352b25c: Image already exists
0998bf8fb9e9: Image already exists
0a85502c06c9: Image already exists
1.0: digest:
sha256:2536fd316ef585127232409599d464636d8e2724bd95d5177cf6771b2e773216
size: 8016
```

**Switch over to your Node 1 instance. At this stage, content trust should be disabled**

**7) Run a container with a bash terminal using the trustedubuntu:1.0 image**

```
$ docker run -it <username>/trustedubuntu:1.0 bash
```

**8) On the container terminal, check to see that you have the unsigned.txt file underneath the / folder**

```
student@node-1:~$ docker run -it trainingteam/trustedubuntu:1.0 bash
Unable to find image 'trainingteam/trustedubuntu:1.0' locally
1.0: Pulling from trainingteam/trustedubuntu
0a85502c06c9: Pull complete
0998bf8fb9e9: Pull complete
a6785352b25c: Pull complete
e9ae3c220b23: Pull complete
9223cedda76a: Pull complete
Digest:
sha256:2536fd316ef585127232409599d464636d8e2724bd95d5177cf6771b2e773216
Status: Downloaded newer image for trainingteam/trustedubuntu:1.0
root@75080b8235b8:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  unsigned.txt  usr  var
root@75080b8235b8:/#
```

**9) Exit the container**

**10) Run another container using the same image but this time run it with content trust enabled**

`$ docker run -it --disable-content-trust=false <username>/trustedubuntu:1.0 bash`

```
student@node-1:~$ docker run -it --disable-content-trust=false
trainingteam/trustedubuntu:1.0 bash
Unable to find image 'trainingteam/trustedubuntu:1.0' locally
sha256:4c30742e696946d4e6c9b56ac8a12feda0d606c3d9a0d2be469bfa8a11b95086:
Pulling from trainingteam/trustedubuntu
Digest:
sha256:4c30742e696946d4e6c9b56ac8a12feda0d606c3d9a0d2be469bfa8a11b95086
Status: Downloaded newer image for
trainingteam/trustedubuntu@sha256:4c30742e696946d4e6c9b56ac8a12feda0d606c3
d9a0d2be469bfa8a11b95086
Tagging
trainingteam/trustedubuntu@sha256:4c30742e696946d4e6c9b56ac8a12feda0d606c3
d9a0d2be469bfa8a11b95086 as trainingteam/trustedubuntu:1.0
root@84df330ae913:/#
```

**11) Try and find the `unsigned.txt` file in the new container**

**12) Notice it is not present because our signed image is currently slightly older than the unsigned image and we did not create that file in it**

```
root@84df330ae913:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  usr  var
root@84df330ae913:/#
```

# Exercise 5.1 – Setup a registry server

**1) Setup a registry server by running the official registry image. Map port 5000 on the container to port 5000 on your host**
Run the command
`docker run –d –p 5000:5000 registry:2.0`

```
docker@52.26.42.249 ~: docker run -d -p 5000:5000 registry:2.0
Unable to find image 'registry:2.0' locally
2.0: Pulling from registry
61b3964dfa70: Pull complete
f5224fc54ad2: Pull complete
...
...
...
ba48d91fd1d8: Pull complete
7461f951ff8e: Pull complete
3b34ad1544ad: Already exists
registry:2.0: The image you are pulling has been verified. Important: image
verification is a tech preview feature and should not be relied on to
provide security.
Digest:
sha256:3920a3e76211bfd33b9d8bf5a741731bead2e3bdfbd82e2c28c1e227fd1e5d28
Status: Downloaded newer image for registry:2.0
f2ceb990da8e3279713399e67bfc524f5af57f21accc34dbd034a88bae5c3d97
docker@52.26.42.249 ~:
```

**2) To verify that the server is up and running you can try and make a basic API call to http://<server url>:5000/v2/. You should get a HTTP 200 response. On your terminal run**
**`curl -i http://localhost:5000/v2/`**

```
docker@52.26.42.249 ~: curl -i http://localhost:5000/v2/
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
Date: Tue, 16 Jun 2015 17:42:47 GMT
{}docker@52.26.42.249 ~:
```

**3) Verify that you get a HTTP 200 response**

# Exercise 5.2 – Push images

**1) Pull the latest busybox image from Docker Hub**

`docker pull busybox`

```
docker@52.26.42.249 ~: docker pull busybox
latest: Pulling from busybox
cf2616975b4a: Already exists
6ce2e90b0bc7: Already exists
8c2e06607696: Already exists
Digest:
sha256:38a203e1986cf79639cfb9b2e1d6e773de84002feea2d4eb006b52004ee8502d
Status: Image is up to date for busybox:latest
docker@52.26.42.249 ~:
```

**2) Tag the image as mybusybox and specify the registry host URL. Also, specify your name as part of the repository. Specify a 1.0 tag. For example:**

When running the step below, make sure to replace "johnnytu" with your name. Once we tag our image, we will run `docker images` to verify

`docker tag busybox localhost:5000/johnnytu/mybusybox:1.0`

```
docker@52.26.42.249 ~: docker tag busybox
localhost:5000/johnnytu/mybusybox:1.0
docker@52.26.42.249 ~:
docker@52.26.42.249 ~: docker images
REPOSITORY                          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
registry                            2.0                 3b34ad1544ad
2 days ago          548.6 MB
...
...
busybox                             latest              8c2e06607696
8 weeks ago         2.433 MB
localhost:5000/johnnytu/mybusybox   1.0                 8c2e06607696
8 weeks ago         2.433 MB
```

**3) Push the image to your registry server**

`docker push localhost:5000/johnnytu/mybusybox:1.0`

```
docker@52.26.42.249 ~: docker push localhost:5000/johnnytu/mybusybox:1.0
The push refers to a repository [localhost:5000/johnnytu/mybusybox] (len:
1)
8c2e06607696: Image already exists
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image successfully pushed
Digest:
sha256:3f3f99b78b8bca268c2e8ebb1dc57c68142223fcc10ef0790d70b9b369a1bb3f
```

**4) Tag the local busybox image again, this time use 1.1**
`docker tag busybox localhost:5000/johnnytu/mybusybox:1.1`

Let's tag and then run `docker images` again to verify

```
docker@52.26.42.249 ~: docker tag busybox
localhost:5000/johnnytu/mybusybox:1.1
docker@52.26.42.249 ~:
docker@52.26.42.249 ~: docker images
REPOSITORY                          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
registry                            2.0                 3b34ad1544ad
2 days ago          548.6 MB
...
...
busybox                             latest              8c2e06607696
8 weeks ago         2.433 MB
localhost:5000/johnnytu/mybusybox   1.0                 8c2e06607696
8 weeks ago         2.433 MB
localhost:5000/johnnytu/mybusybox   1.1                 8c2e06607696
8 weeks ago         2.433 MB
```

**5) Push the image from 4) into the registry server**
`docker push localhost:5000/johnnytu/mybusybox:1.1`

```
docker@52.26.42.249 ~: docker push localhost:5000/johnnytu/mybusybox:1.1
The push refers to a repository [localhost:5000/johnnytu/mybusybox] (len:
1)
8c2e06607696: Image already exists
6ce2e90b0bc7: Image already exists
cf2616975b4a: Image already exists
Digest:
sha256:baa3ec5396638e9aed659cb34e5e00e99bc95e4a689619fc9736d9cc094b57b3
docker@52.26.42.249 ~:
```

**6) Delete the local busybox images**

Since all our busybox images are the same. A fast way to delete all our busybox images would be to use `docker rmi -f` and specify the image ID.

In the example below, our `busybox` image has ID of 8c2e06607696

```
docker@52.26.42.249 ~: docker rmi -f 8c2e06607696
Untagged: busybox:latest
docker@52.26.42.249 ~:
```

# Exercise 5.3 – Pull from the registry

**1) Pair up with another student. Grab their AWS server URL and image repository details**

For this, let's pretend that we have paired up with Jane Doe and Jane's AWS server URL/IP is 52.26.37.124.
Jane has pushed her busybox image using the repository name of `janedoe/mybusybox:1.0`

**2) On your terminal, run a command to display the tags for the busybox image on your partner's registry server**

The command syntax to make the API call is
`curl http://<server url>:5000/v2/<name>/mybusybox/tags/list`

In our example we run

```
docker@52.26.42.249 ~: curl
http://52.26.37.124:5000/v2/janedoe/mybusybox/tags/list
{"name":"janedoe/mybusybox","tags":["1.0","1.1"]}
```

You can see the output list out the two tags 1.0 and 1.1

**3) Try to pull the image with tag 1.0**
`docker pull <server url>:5000/<name>/mybusybox:1.0`

```
docker@52.26.42.249 ~: docker pull 52.26.37.124:5000/janedoe/mybusybox:1.0
FATA[0000] Error response from daemon: v1 ping attempt failed with error:
Get https://52.26.37.124:5000/v1/_ping: tls: oversized record received with
length 20527. If this private registry supports only HTTP or HTTPS with an
unknown CA certificate, please add `--insecure-registry 52.26.37.124:5000`
to the daemon's arguments. In the case of HTTPS, if you have access to the
registry's CA certificate, no need for the flag; simply place the CA
certificate at /etc/docker/certs.d/52.26.37.124:5000/ca.crt
docker@52.26.42.249 ~:
```

**4) What do you notice?**

You can see from the error below, that Docker does not allow you to connect to the registry running on a different host. More on this in the course slides

# Exercise 5.4 – Configure insecure registry

**1) Open your `/etc/default/docker` file**

**2) Add the `--insecure-registry` option into `DOCKER_OPTS` and specify the IP address or domain of your partner's register server**
`DOCKER_OPTS="--insecure-registry <ip address>:5000"`

In the previous exercise, we established that our partner was Jane Doe and the IP of her host was 52.26.37.124. When following the examples here, remember to replace the details with the details provided to you by your partner

Your `DOCKER_OPTS` variable by now should look like the following:

```
DOCKER_OPTS="-H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock -H
tcp://0.0.0.0:2376 --tlsverify --tlscacert=/etc/docker/ca.pem
--tlscert=/etc/docker/server-cert.pem --tlskey=/etc/docker/server-key.pem
--insecure-registry 52.26.37.124:5000"
```

**3) Restart the Docker service**

```
sudo service docker restart
```

**4) Now try and pull the busybox image from your partner's registry again. This time, it should work.**
`docker pull <server url>:5000/<name>/mybusybox:1.0`

For our example, we specify

```
docker pull 52.26.37.124:5000/janedoe/mybusybox:1.0
```

```
docker@52.26.42.249 ~: docker pull 52.26.37.124:5000/janedoe/mybusybox:1.0
1.0: Pulling from 52.26.37.124:5000/janedoe/mybusybox
cf2616975b4a: Pull complete
6ce2e90b0bc7: Pull complete
8c2e06607696: Already exists
Digest:
sha256:b8e19268a9ba0001f3d223b012f49326e94735715ff9aa5d4bf7b9fef7885ca5
Status: Downloaded newer image for 52.26.37.124:5000/janedoe/mybusybox:1.0
```

**5) Check that the image has been pulled into your local box**
`docker images`

```
docker@52.26.42.249 ~: docker images
REPOSITORY                              TAG      IMAGE ID            CREATED
VIRTUAL SIZE
registry                                2.0      3b34ad1544ad        2 days
ago      548.6 MB
...
...
52.26.37.124:5000/janedoe/mybusybox   1.0      8c2e06607696        8 weeks
ago     2.433 MB
docker@52.26.42.249 ~:
```

# Exercise 7.1 – Install machine on Ubuntu

### Use your master instance

1) Download the Linux binary from https://github.com/docker/machine/releases/download/v0.5.2/docker-machine_linux-amd64.zip

```
student@master:~$ wget
https://github.com/docker/machine/releases/download/v0.5.2/docker-machine_
linux-amd64.zip
--2016-01-15 22:24:58--
https://github.com/docker/machine/releases/download/v0.5.2/docker-machine_
linux-amd64.zip
Resolving github.com (github.com)... 192.30.252.130
Connecting to github.com (github.com)|192.30.252.130|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location:
https://github-cloud.s3.amazonaws.com/releases/27494663/10d090be-97ba-11e5
-8d9e-c67088ab09a4.zip?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=A
KIAISTNZFOVBIJMK3TQ%2F20160116%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=
20160116T032458Z&X-Amz-Expires=300&X-Amz-Signature=a2852984c2f874df5e9e706
000068e8c84e4363494b916507172ea4899614663&X-Amz-SignedHeaders=host&actor_i
d=0&response-content-disposition=attachment%3B%20filename%3Ddocker-machine
_linux-amd64.zip&response-content-type=application%2Foctet-stream
[following]
--2016-01-15 22:24:58--
https://github-cloud.s3.amazonaws.com/releases/27494663/10d090be-97ba-11e5
-8d9e-c67088ab09a4.zip?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=A
KIAISTNZFOVBIJMK3TQ%2F20160116%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=
20160116T032458Z&X-Amz-Expires=300&X-Amz-Signature=a2852984c2f874df5e9e706
000068e8c84e4363494b916507172ea4899614663&X-Amz-SignedHeaders=host&actor_i
d=0&response-content-disposition=attachment%3B%20filename%3Ddocker-machine
_linux-amd64.zip&response-content-type=application%2Foctet-stream
Resolving github-cloud.s3.amazonaws.com (github-cloud.s3.amazonaws.com)...
54.231.98.80
Connecting to github-cloud.s3.amazonaws.com
(github-cloud.s3.amazonaws.com)|54.231.98.80|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 35410200 (34M) [application/octet-stream]
Saving to: ‘docker-machine_linux-amd64.zip’
100%[=======================================================================
======================>] 35,410,200  9.91MB/s   in 4.9s
2016-01-15 22:25:04 (6.88 MB/s) - ‘docker-machine_linux-amd64.zip’ saved
[35410200/35410200]
student@master:~$ ls
docker-machine_linux-amd64.zip
student@master:~$
```

**2) Unzip the file into the /usr/local/bin folder (you may need to install the unzip program)**

```
$ sudo apt-get install unzip
```

Once uzip has been installed, run

```
$ sudo unzip docker-machine_linux-amd64.zip -d /usr/local/bin
```

```
student@master:~$ sudo unzip docker-machine_linux-amd64.zip -d
/usr/local/bin
sudo unzip docker-machine_linux-amd64.zip -d /usr/local/bin
Archive:  docker-machine_linux-amd64.zip
  inflating: /usr/local/bin/docker-machine
  inflating: /usr/local/bin/docker-machine-driver-amazonec2
  inflating: /usr/local/bin/docker-machine-driver-azure
  inflating: /usr/local/bin/docker-machine-driver-digitalocean
  inflating: /usr/local/bin/docker-machine-driver-exoscale
  inflating: /usr/local/bin/docker-machine-driver-generic
  inflating: /usr/local/bin/docker-machine-driver-google
  inflating: /usr/local/bin/docker-machine-driver-hyperv
  inflating: /usr/local/bin/docker-machine-driver-none
  inflating: /usr/local/bin/docker-machine-driver-openstack
  inflating: /usr/local/bin/docker-machine-driver-rackspace
  inflating: /usr/local/bin/docker-machine-driver-softlayer
  inflating: /usr/local/bin/docker-machine-driver-virtualbox
  inflating: /usr/local/bin/docker-machine-driver-vmwarefusion
  inflating: /usr/local/bin/docker-machine-driver-vmwarevcloudair
  inflating: /usr/local/bin/docker-machine-driver-vmwarevsphere
student@master:~$
```

## Exercise 7.2 – Install Docker Toolbox

> If you already have Docker Toolbox installed on your PC or MAC, you can skip this exercise

**1) Go to https://www.docker.com/docker-toolbox. Download and install Docker Toolbox on your laptop. This will install Docker Machine as well**

**For Mac OSX Users**

**2) Open your terminal and run `docker-machine -v` and make sure you can see the Docker Machine version**

**For Windows Users**

**2) Open your CMD command line terminal and run `docker-machine -v` and make sure you can see the Docker Machine version**

```
C:\Users\Johnny>docker-machine -v
docker-machine version 0.4.1 (e2c88d6)
C:\Users\Johnny>
```

## Exercise 7.3 – Create machine on VirtualBox

**The following examples were done in Windows using `msysgit`. OSX users can just use their terminal and run the same commands**

**listed. Windows users can also use CMD terminal**

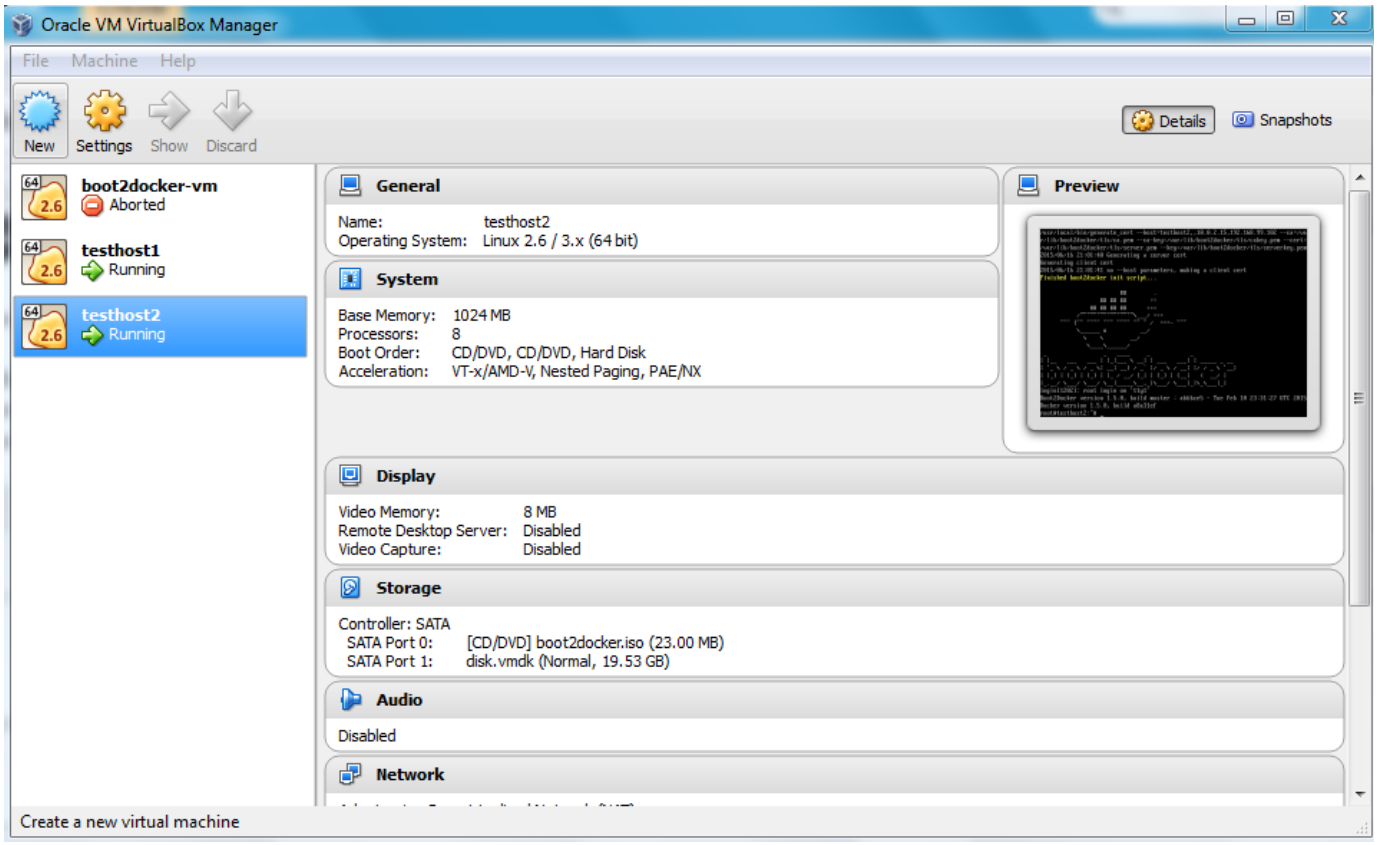
**1) Open your PC or Mac terminal.**

**2) Create two hosts on VirtualBox called testhost1 and testhost2**

```
docker-machine create --driver virtualbox testhost1
docker-machine create --driver virtualbox testhost2
```

```
 Johnny@JTCOMMANDCENTER ~
$ docker-machine create --driver virtualbox testhost1
[34mINFO[0m[0000] Creating SSH key...
[34mINFO[0m[0003] Creating VirtualBox VM...
[34mINFO[0m[0026] Starting VirtualBox VM...
[34mINFO[0m[0033] Waiting for VM to start...
[34mINFO[0m[0100] "testhost1" has been created and is now the active
machine.
[34mINFO[0m[0100] To point your Docker client at it, run this in your
shell: eval "$(C:\Program Files (x86)\Git\bin\docker-machine env
testhost1)"
Johnny@JTCOMMANDCENTER ~
Johnny@JTCOMMANDCENTER ~
$ docker-machine create --driver virtualbox testhost2
[34mINFO[0m[0000] Creating SSH key...
[34mINFO[0m[0001] Creating VirtualBox VM...
[34mINFO[0m[0023] Starting VirtualBox VM...
[34mINFO[0m[0044] Waiting for VM to start...
[34mINFO[0m[0111] "testhost2" has been created and is now the active
machine.
[34mINFO[0m[0111] To point your Docker client at it, run this in your
shell: eval "$(C:\Program Files (x86)\Git\bin\docker-machine env
testhost2)"
```

**3) Open VirtualBox and check that both hosts are present**




# Exercise 7.4 – Provision cloud hosts

**1) On your Ubuntu AWS instance, use docker-machine to create two hosts in a Cloud environment.**
**Your instructor will provide you with the necessary details**

In this example we will use a DigitalOcean account. To create hosts on DigitalOcean, we need an access token. For our example in this, the access token is de4fcf18f7d36914dd30d9b62e44c85269bd4820055b5db2ddf203ac7e7f2344

**2) Alternatively, if you have your own AWS or DigitalOcean account or an account with any other supported provider you may create the hosts there**

**3) Call your hosts cloudhost1 and cloudhost2 and prepend it with your name. For example: jtucloudhost1**

```
student@DockerTraining:~$ docker-machine create --driver digitalocean
--digitalocean-access-token
b392ac396b8c4caebd345f3a63fae4465b6ac46e07b4cb3a30802d2e1b3d7447
jtucloudhost1
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env
jtucloudhost1
student@DockerTraining:~$ docker-machine create --driver digitalocean
--digitalocean-access-token
b392ac396b8c4caebd345f3a63fae4465b6ac46e07b4cb3a30802d2e1b3d7447
jtucloudhost2
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env
jtucloudhost2
```

**4) Repeat the same steps but this time, provision the host from the docker-machine running on your PC or Mac. Call your host `cloudhostfrompc` and prepend with your name. For example:**
`jtucloudhostfrompc`

Note: If you do not have `docker-machine` setup on your PC or Mac and are unable to install due to insufficient privileges, you can ignore this step

```
 Johnny@JTCOMMANDCENTER ~
$ docker-machine create --driver digitalocean --digitalocean-access-token
de4fcf18f7d36914dd30d9b62e44c85269bd4820055b5db2ddf203ac7e7f2
344 jtucloudhostfrompc1
[34mINFO[0m[0001] Creating SSH key...
[34mINFO[0m[0002] Creating Digital Ocean droplet...
[34mINFO[0m[0135] "jtucloudhostfrompc1" has been created and is now the
active machine.
[34mINFO[0m[0135] To point your Docker client at it, run this in your
shell:
eval "$(C:\Program Files (x86)\Git\bin\docker-machine env
jtucloudhostfrompc1)"
```

## Exercise 7.5 – List machines

**1) Using your Ubuntu AWS instance, list out all the hosts provisioned by docker-machine**
`docker-machine ls`

```
docker@52.26.42.249 ~: docker-machine ls
NAME             ACTIVE   DRIVER         STATE     URL
SWARM
jtucloudhost1            digitalocean   Running   tcp://104.236.200.29:2376
jtucloudhost2            digitalocean   Running   tcp://45.55.87.226:2376
```

**2) Go to your home directory and run `ls -la`**

```
docker@52.26.42.249 ~: ls -la
total 11756
drwxr-xr-x 8 docker docker     4096 Jun 16 21:19 .
drwxr-xr-x 4 root   root       4096 Jun  4 01:13 ..
-rw------- 1 docker docker     9758 Jun 16 09:20 .bash_history
-rw-r--r-- 1 docker docker      220 Apr  9  2014 .bash_logout
-rw-r--r-- 1 docker docker     3691 Jun  4 01:13 .bashrc
drwx------ 2 docker docker     4096 Jun  8 00:06 .cache
drwxrwxr-x 3 docker docker     4096 Jun 16 21:17 .docker
...
...
docker@52.26.42.249 ~:
```

**3) Confirm that you can see the hidden `.docker` folder**

**4) Go into the `.docker` folder and inside, check for the "machine" folder and change directory into machine.**

**5) Then check for the machines folder and change directory into it**

**6) What do you notice in the machines folder (/home/<user>/.docker/machine/machines)**

The output of steps 4 to 6 are as follows:

```
docker@52.26.42.249 ~: cd .docker/
docker@52.26.42.249 ~/.docker: cd machine/machines/
docker@52.26.42.249 ~/.docker/machine/machines: ls
jtucloudhost1  jtucloudhost2]
```

You should see a folder for each of the hosts that you have provisioned.

## Exercise 7.6 – Connect client to remote host

**1) Using your Ubuntu AWS instance, connect your Docker client to cloudhost1**
The command syntax is
`eval $(docker-machine env <host name>)`

Remember that our host name was "jtucloudhost1"

```
student@DockerTraining:~$ eval $(docker-machine env jtucloudhost1)
```

To verify that our client is connected, we can run

```
student@DockerTraining:~$ echo $DOCKER_HOST
tcp://159.203.142.192:2376
```

If you compare this to the output of `docker-machine ls`, you will see that the client is pointing at `jtucloudhost1`

**2) Check your active host to make sure the Docker client is connected to cloudhost1**

```
student@DockerTraining:~$ docker-machine active
jtucloudhost1
```

Also, you run docker-machine ls, the active host will be marked on the ACTIVE column

```
student@DockerTraining:~$ docker-machine ls
NAME            ACTIVE   DRIVER         STATE     URL
SWARM
jtucloudhost1   *        digitalocean   Running
tcp://159.203.142.192:2376
jtucloudhost2            digitalocean   Running
tcp://159.203.142.199:2376
```

**3) Run a few containers of your choice on the host. Remember what you have run. You will need it later.**

Let's run an NGINX container in the background. Our new host won't have the NGINX image, so you should see it being pulled

```
docker@52.26.42.249 ~: docker run -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from nginx
64e5325c0d9d: Pull complete
...
...
36f0b0306c21: Pull complete
107c338c1d31: Pull complete
319d2015d149: Already exists
nginx:latest: The image you are pulling has been verified. Important: image
verification is a tech preview feature and should not be relied on to
provide security.
Digest:
sha256:43bc7f2d683c9dd77b41e2d68e3af72ac3df45c3e16c1794f3c4a74e5e1abf61
Status: Downloaded newer image for nginx:latest
c6c5a40232bba23609773091694fbb8f9ca021cd02c1295158e8b7a11a161d28
```

Now lets check our containers. The only container we should see is the NGINX one that we just ran

```
docker@52.26.42.249 ~: docker ps -a
CONTAINER ID     IMAGE            COMMAND                CREATED
STATUS              PORTS            NAMES
c6c5a40232bb     nginx:latest     "nginx -g 'daemon of   About a minute ago
Up About a minute   80/tcp, 443/tcp   focused_hypatia
```

## Exercise 7.7 – Disconnect and connect again

**1) Disconnect your Docker client from the remote daemon**
`eval $(docker-machine env -u)`

At this stage, we are connected to "jtucloudhost1" Once we run the command above, the Docker client will be connected back on the main host

**2) Connect it to the other host you provisioned**
`eval $(docker-machine env <hostname>)`

Recall that in Exercise 15.4, we called our second host "jtucloudhost2". Your host will be prefixed with your own name, so adjust the command accordingly

```
docker@52.26.42.249 ~: eval $(docker-machine env jtucloudhost2)
```

**3) Run `docker ps -a` and notice how there are no previous containers**

This is because we haven't run any container's on our second host yet

```
docker@52.26.42.249 ~: docker ps -a
CONTAINER ID        IMAGE          COMMAND         CREATED         STATUS
PORTS           NAMES
```

**4) Run a few containers**

Let's run a Tomcat container

```
docker@52.26.42.249 ~: docker run -d tomcat
Unable to find image 'tomcat:latest' locally
latest: Pulling from tomcat
61b3964dfa70: Pull complete
f5224fc54ad2: Pull complete
...
...
Status: Downloaded newer image for tomcat:latest
8d278aad7312b0c380139992717a866f5703389dcf017dfbf122a1983cedd014
```

**5) Disconnect the client from the remote daemon. The client should be back on your localhost. Connect back to the first remote host**

First, we disconnect

```
docker@52.26.42.249 ~: eval $(docker-machine env -u)
docker@52.26.42.249 ~: echo $DOCKER_HOST

docker@52.26.42.249 ~:
```

Now to connect back to the first host

```
docker@52.26.42.249 ~: eval $(docker-machine env jtucloudhost1)
```

**6) Run `docker ps -a`. Can you see the containers you ran in exercise 7.6?**

We should see our NGINX container

```
docker@52.26.42.249 ~: docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
...       NAMES
c6c5a40232bb        nginx:latest        "nginx -g 'daemon of   18 minutes
ago    ...       focused_hypatia
```

# Exercise 7.8 – SSH into a host

**Note: If you cannot do this on your PC or MAC, you can use the AWS host allocated to you for the class**

The example below are based on the `docker-machine` binary running on a PC. The terminal is `msysgit`

**1) Using your PC or Mac terminal, connect to one of the VirtualBox VM's you created using SSH**
`docker-machine ssh <host name>`

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ssh testhost1
                        ##         .
                  ## ## ##        ==
               ## ## ## ##       ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
           \______ o          __/
             \    \        __/
              \____\______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
...
docker@testhost1:~$
```

The hosts that we provisioned on VirtualBox are running Boot2Docker

**2) Run a few containers of your choice and then exit the host**

Let's run a busybox  container

```
docker@testhost1:~$ docker run -it -d busybox
Unable to find image 'busybox:latest' locally
Pulling repository busybox
8c2e06607696: Download complete
cf2616975b4a: Download complete
6ce2e90b0bc7: Download complete
Status: Downloaded newer image for busybox:latest
e0059ddc59206a3865f34e5e5210ba8df5eb82b31ef8037ffcb7e29137f05e79
```

We can exit just by typing `exit` on the terminal

```
docker@testhost1:~$ exit
Johnny@JTCOMMANDCENTER ~
$
```

**3) Use the ssh command to check what containers are running on the host you just exited**

`docker-machine ssh <host name> docker ps`

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ssh testhost1 docker ps
CONTAINER ID        IMAGE                         COMMAND
CREATED                   NAMES
e0059ddc5920        busybox:buildroot-2014.02   "/bin/sh"           About a
minute ago         suspicious_ritchie
```

**4) Run `docker-machine ssh <host name> ps -ef`. Notice the error**

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ssh testhost1 ps -ef
Incorrect Usage.
```

**5) Fix the command in 4)**

The error occurred because we tried to specify the `-ef` argument. If we want to specify a flag on a command we run via ssh, we need the flag terminator, which is `--`
The command should be:
`docker-machine ssh <host name> -- ps -ef`

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ssh testhost1 -- ps -ef
PID   USER     COMMAND
    1 root     init
    2 root     [kthreadd]
    3 root     [ksoftirqd/0]
...
...
```

## Exercise 7.9 – Start and stop machines

**This exercise is optional. The following examples were run on Windows with `msysgit`. Commands are the same on MAC and Linux**

**1) Using your PC (msysgit) or Mac terminal, stop one of the VirtualBox hosts you created previously**

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine stop testhost1
```

**2) Run `docker-machine ls` and verify the machine state**

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ls
NAME                  ACTIVE   DRIVER         STATE     URL
SWARM
jtucloudhostfrompc1   *        digitalocean   Running
tcp://45.55.87.215:2376
testhost1                      virtualbox     Stopped
testhost2                      virtualbox     Running
tcp://192.168.99.103:2376
```

**3) Now start the host up**

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ls
NAME                  ACTIVE   DRIVER         STATE     URL
SWARM
jtucloudhostfrompc1   *        digitalocean   Running
tcp://45.55.87.215:2376
testhost1                      virtualbox     Stopped
testhost2                      virtualbox     Running
tcp://192.168.99.103:2376
```

**4) Open VirtualBox and manually stop the other host**

Right click on your `testhost2` in the interface, select the "close" option and then look for "Power Off"

**5) Run `docker-machine ls` and verify the machine state (might take a while for it to get the state information)**

**6) Now start the host again**

Run `docker-machine start testhost2`

# Exercise 7.10 – Delete hosts

**1) Change directory into `/home/<user>/.docker/machine/machines`**

**2) List the folder in this directory and notice how each machine you provisioned has its own folder**

The output of steps 1 and 2 should be as follows

```
docker@52.26.42.249 ~: cd .docker/machine/machines/
docker@52.26.42.249 ~/.docker/machine/machines: ls
jtucloudhost1  jtucloudhost2
```

Notice the two folders, one for each host that was provisioned.

**3) Using your AWS Ubuntu host, delete the two other hosts that you provisioned earlier**
`docker-machine rm <host name>`

You can actually specify multiple host names to delete multiple hosts at once. Once we run the command, you should notice that the two folders are also deleted.

```
docker@52.26.42.249 ~/.docker/machine/machines: docker-machine rm
jtucloundhost1 jtucloundhost2
INFO[0001] The machine was successfully removed.
```

**4) List the files in the directory again and verify that the machines have been deleted. Also verify this by running** `docker-machine ls`

```
docker@52.26.42.249 ~/.docker/machine/machines: docker-machine ls
NAME   ACTIVE   DRIVER   STATE   URL   SWARM
docker@52.26.42.249 ~/.docker/machine/machines: ls
docker@52.26.42.249 ~/.docker/machine/machines:
```

**5) Open VirtualBox and manually delete one of the hosts**

To delete a host in VirtualBox, you need to power it off first. Once the host is powered off, you will be able to remove it.

**6) Open your local PC or Mac terminal (use msysgit for PC) and run `docker-machine ls`. Notice the error message for the host you manually deleted**

In the example below, testhost2 was deleted

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ls
[31mERRO[0m[0000] error getting state for host testhost2: machine does not
exist
[31mERRO[0m[0000] error getting URL for host testhost2: machine does not
exist
NAME                 ACTIVE   DRIVER         STATE     URL
SWARM
jtucloudhostfrompc1   *        digitalocean   Running
tcp://45.55.87.215:2376
testhost1                      virtualbox     Running
tcp://192.168.99.101:2376
testhost2                      virtualbox     Error
```

**7) Delete the local reference to the host machine that is causing the error**

Run `docker-machine rm testhost2`

# Exercise 8.1 – Start swarm manager

**Using main AWS instance**

Remember, you will have been allocated three AWS or cloud instances of Ubuntu to work with.

- Your main or master instance
- Node 1
- Node 2

For this exercise, you will use your main instance

**1) First, create the token that we will use to identify our cluster**
`docker run --rm swarm create`

```
docker@52.26.42.249 ~: docker run --rm swarm create
Unable to find image 'swarm:latest' locally
latest: Pulling from swarm
....
....
73504b2882a3: Already exists
swarm:latest: The image you are pulling has been verified. Important: image
verification is a tech preview feature and should not be relied on to
provide security.
Digest:
sha256:aaaf6c18b8be01a75099cc554b4fb372b8ec677ae81764dcdf85470279a61d6f
Status: Downloaded newer image for swarm:latest
461223b3ab291a4471dd3270aec80527
```

The last line in the output (461223b3ab291a4471dd3270aec80527) is the cluster token

**2) Copy the output from the command in 1) and paste into a file called `swarmtoken`. Do not lose this token you will need it for the next few exercises. Put the `swarmtoken` file on your home directory**

**3) Start the swarm manager using automatic port mapping**
`docker run -d -P swarm manage token://$(cat swarmtoken)`

One reason for saving the cluster token into a file is so we can easily reference it with `cat`

```
docker@52.26.42.249 ~: docker run -d -P swarm manage token://$(cat
swarmtoken)
3b46d05a38c0a2d69eadf1841db4021a5361f181c051dacab2e5e295143d2ba4
```

# Exercise 8.2 – Join nodes to cluster

**1) Switch to your second AWS instance. This instance will become node 1 in the cluster**
In the examples these are our host IP's. You will have your own set of hosts, so be sure to use the correct IP address when following the examples

- Master - 52.26.42.249
- Node 1 - 52.26.37.124
- Node 2 - 52.26.34.74

**2) Configure the `/etc/default/docker` file to have the daemon listening on tcp://0.0.0.0:2375**
`DOCKER_OPTS="-H tcp://0.0.0.0:2375"`

Your `DOCKER_OPTS` variable should something like the following

```
DOCKER_OPTS="-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375"
```

**3) Restart the Docker service**
`sudo service docker restart`

**4) Run `ifconfig` and note down the IP address of the machine**

If you already have the public IP address of your node 1 instance there is no need for this step. In this example, our node 1 public IP is 52.26.37.124

**5) Join the node to the cluster using the cluster token from your main AWS instance. Make sure you specify the IP address of this host**
`docker run –d swarm join --addr=<node ip>:2375 token://<cluster token>`

Many people make the mistake in this step by specifying the IP address of the host where the Swarm manager is running. **This is incorrect. Make sure you specify the IP address of the node you are on.** In this case our node 1 IP is 52.26.37.124

You will need to switch back to your master instance in order to get the cluster token from the swarmtoken file. In our case the token is 461223b3ab291a4471dd3270aec80527

```
docker@52.26.37.124 ~: docker run -d swarm join --addr=52.26.37.124:2375
token://461223b3ab291a4471dd3270aec80527
Unable to find image 'swarm:latest' locally
latest: Pulling from swarm
de939d6ed512: Pull complete
....
....
....
Status: Downloaded newer image for swarm:latest
57347fda045a938f7684d75cd154ee67b31ee3a943ca2e0e53946f4a9e180b0f
```

6) Switch back to your master AWS instance and list the nodes in the cluster
`docker run --rm swarm list token://<cluster_token>`

You should see your Node 1 IP address listed

```
docker@52.26.42.249 ~: docker run --rm swarm list token://$(cat swarmtoken)
52.26.37.124:2375
```

**7) Verify you can see the IP address of your Node 1 AWS instance on the list**

**8) Now repeat the same steps on your Node 2 AWS instance. This instance will become node 2 in the cluster**

Make sure you switch your terminal into your node 2 instance. In this case our node 2 IP address is 52.26.34.74.

First we need to make sure that the Docker daemon on the host is listening on tcp://0.0.0.0:2375. Let's configure the `/etc/default/docker` file

```
DOCKER_OPTS="-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375"
```

Then we restart Docker

`sudo service docker restart`

And finally we run `swarm join`

```
docker@52.26.34.74 ~: docker run -d swarm join --addr=52.26.34.74:2375
token://461223b3ab291a4471dd3270aec80527
```

**9) By the end of this exercise you should have your Swarm master managing two nodes**

If you switch back to your master instance. We can run the `swarm list` command again to confirm this.

```
docker@52.26.42.249 ~: docker run --rm swarm list token://$(cat swarmtoken)
52.26.37.124:2375
52.26.34.74:2375
```

# Exercise 8.3 – Connect client

**1) Make sure you are using your main AWS instance**

Following on from previous exercises, we will on be our master instance.

**2) Run docker ps to find your Swarm container and check the port mapping**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                ...
PORTS                     NAMES
3b46d05a38c0        swarm:latest        "/swarm manage token   ...
0.0.0.0:32769->2375/tcp   modest_davinci
```

From the output above, you can see that our Swarm container exposes its port 2375 and this is mapped to port 32769 on our host. The name is ' modest_davinci'. Note that your values may be different.

**3) Set the `DOCKER_HOST` variable to the IP of 127.0.01 and specify the port from question 2).**
`export DOCKER_HOST=<ip>:<port>`

```
docker@52.26.42.249 ~: export DOCKER_HOST=10.0.0.2:32769
```

**4) Run `docker version` and verify that Swarm is listed as the server version**

```
docker@52.26.42.249 ~: docker version
Client:
 Version:      1.11.1
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   5604cbe
 Built:        Tue Apr 26 23:30:23 2016
 OS/Arch:      linux/amd64
Server:
 Version:      swarm/1.2.3
 API version:  1.22
 Go version:   go1.5.4
 Git commit:   eaa53c7
 Built:        Fri May 27 17:25:03 UTC 2016
 OS/Arch:      linux/amd64
```

**5) Run `docker info` to check the status of your connected nodes**

```
docker@52.26.42.249 ~: docker info
Containers: 3
 Running: 2
 Paused: 0
 Stopped: 1
Images: 2
Server Version: swarm/1.2.3
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint
Nodes: 2
 node-1: 10.0.18.36:2375
   ID: GIAS:R7OB:CERX:32F2:D4A7:ZHAG:42QJ:XUS4:L25U:7IPJ:R6NM:QJ2C
   Status: Healthy
   Containers: 2
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 3.859 GiB
   Labels: executiondriver=, kernelversion=4.2.0-23-generic,
operatingsystem=Ubuntu 14.04.4 LTS, storagedriver=aufs
   UpdatedAt: 2016-06-07T05:12:11Z
   ServerVersion: 1.11.1
 node-2: 10.0.32.57:2375
   ID: HRWT:XHQZ:DPJM:KDWG:DF6I:QJUW:YSQM:FMBQ:TGIR:XO5V:H7QV:Z7SW
   Status: Healthy
   Containers: 1
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 3.859 GiB
   Labels: executiondriver=, kernelversion=4.2.0-23-generic,
operatingsystem=Ubuntu 14.04.4 LTS, storagedriver=aufs
   UpdatedAt: 2016-06-07T05:12:00Z
   ServerVersion: 1.11.1
Plugins:
 Volume:
 Network:
Kernel Version: 4.2.0-23-generic
Operating System: linux
Architecture: amd64
CPUs: 2
Total Memory: 7.718 GiB
Name: 677ba9c521d4
Docker Root Dir:
Debug mode (client): false
Debug mode (server): false
WARNING: No kernel memory limit support
```

Notice the two nodes. At the moment they each have one container. This is because we ran our Swarm container's in detached mode. The Swarm containers on our nodes don't actually have to stay running for our nodes to stay connected to the cluster

# Exercise 8.4 – Run containers in the cluster

Using your master AWS instance

**1) Run an NGINX container in detached mode**
`docker run -d nginx`

```
docker@52.26.42.249 ~: docker run -d nginx
547b5b7ac56e6783b65c7dcb2c8b61747f37e394447278c322170518431d593a
```

**2) What node did Swarm run the container on? (Use `docker ps` to find out)**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  ...
NAMES
547b5b7ac56e        nginx:latest        "nginx -g 'daemon of   ...
52.26.34.74/lonely_archimedes
```

Notice how you can see the IP address of the node as part of the container name

**3) Run a Tomcat container in detached mode**
`docker run -d tomcat`

```
docker@52.26.42.249 ~: docker run -d tomcat
96790c062cc3b00fb881aee71ce76a27129fd2d99a966cb1befdf83db9ea4c0f
```

**4) What node is the container running in?**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  ...
NAMES
96790c062cc3        tomcat:latest       "catalina.sh run"        ...
52.26.37.124/hopeful_almeida
547b5b7ac56e        nginx:latest        "nginx -g 'daemon of    ...
52.26.34.74/lonely_archimedes
```

Notice how both containers are on different nodes.

# Exercise 8.5 – Spread strategy

**Using your master AWS instance**

**1) Run `docker ps` to check the number of containers you have and what node they are running on. At this stage there should be 1 container on each node**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  ...     NAMES
96790c062cc3        tomcat:latest       "catalina.sh run"        ...
52.26.37.124/hopeful_almeida
547b5b7ac56e        nginx:latest        "nginx -g 'daemon of  ...
52.26.34.74/lonely_archimedes
```

Right now there is 1 container on each node

**2) Run two more NGINX containers in detached mode and observe which node they are scheduled on**

```
docker@52.26.42.249 ~: docker run -d nginx
^[[A^[[B0bdbf153b1b479fbd0a3d4f0f55bdcc05b2194e8dcadff7b1a35be7bab800369
docker@52.26.42.249 ~: docker run -d nginx
3f58973fef0675246c36f8bc92f23b4739dbbc0c53d3791024451b8fcf8248d4
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  ...     NAMES
3f58973fef06        nginx:latest        "nginx -g 'daemon of   ...
52.26.34.74/stupefied_wilson
0bdbf153b1b4        nginx:latest        "nginx -g 'daemon of   ...
52.26.37.124/sick_yalow
96790c062cc3        tomcat:latest       "catalina.sh run"        ...
52.26.37.124/hopeful_almeida
547b5b7ac56e        nginx:latest        "nginx -g 'daemon of   ...
52.26.34.74/lonely_archimedes
```

From the output, you can see there are now two containers on each node based on the IP address on the container names.

**3) Switch over to the terminal on node 1**

**You should now be on Node 1**

**4) Connect the Docker client to the daemon via TCP**

`export DOCKER_HOST="tcp://localhost:2375"`

5) Run two more NGINX containers in detached mode. There should now be 4 containers on the host machine. (5 if we include the Swarm container we used to run the join command)

```
docker@52.26.37.124 ~: docker run -d nginx
7c2623dff0f09b5f328978c43a0d3795d02850585cf7c36321d2fe209207fc00
docker@52.26.37.124 ~: docker run -d nginx
2e9b6197e668c92661cfafba6b8df85fbc7c9615d7d2971336c4dcdf23aba967
docker@52.26.37.124 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
...     NAMES
2e9b6197e668        nginx:latest        "nginx -g 'daemon of   16 seconds
ago   ...     jovial_lovelace
7c2623dff0f0        nginx:latest        "nginx -g 'daemon of   18 seconds
ago   ...     modest_curie
0bdbf153b1b4        nginx:latest        "nginx -g 'daemon of   18 minutes
ago   ...     sick_yalow
96790c062cc3        tomcat:latest       "catalina.sh run"      22 minutes
ago   ...     hopeful_almeida
57347fda045a        swarm:latest        "/swarm join --addr=   2 hours ago
...     tender_einstein
```

**6) Switch back to the master AWS instance**

**Using your master AWS instance**

**7) Run docker info and to check the number of containers on each node. You should observe four containers on one of your nodes and two on the other**

```
docker@52.26.42.249 ~: docker info
Containers: 8
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 3
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 5
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

In our case, there are 5 containers on one node and 3 on the other because our Swarm container that we used to run the join command is still actually running.

**8) Run two more NGINX containers in detached mode. What node were they scheduled on?**

```
docker@52.26.42.249 ~: docker run -d nginx
7d0b38e55c02ab5d3449f6447878c185e861dd07866bed66f97e54c73d0a7180
docker@52.26.42.249 ~: docker run -d nginx
36ab8162b3ca0013b750a8691e067c48656379bfe33b67b7e7db1343c0c0331c
```

To check the nodes, we could just use `docker info`

```
docker@52.26.42.249 ~: docker info
Containers: 10
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 5
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 5
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

You will now notice that both nodes have 5 containers. The spread strategy ensures that containers are run evenly across hosts. So if one host has a lot more containers running, Swarm will schedule containers for the other hosts until they all have the same number of containers.

**9) Quick clean up: Stop all containers and then delete them**

```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
```

`We can combined these two commands together by running docker rm -f $(docker ps -aq)`

```
docker@52.26.42.249 ~: docker rm -f $(docker ps -sq)
36ab8162b3ca
7d0b38e55c02
2e9b6197e668
7c2623dff0f0
3f58973fef06
0bdbf153b1b4
96790c062cc3
547b5b7ac56e
6489481535c0
57347fda045a
```

## Exercise 8.6 – Switch to binpack strategy

**Using master AWS instance**

**1) Disconnect your Docker client from the Swarm manager**

```
docker@52.26.42.249 ~: unset DOCKER_HOST
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                     NAMES
3b46d05a38c0        swarm:latest        "/swarm manage token   4 hours ago
Up 4 hours          0.0.0.0:32769->2375/tcp   modest_davinci
```

Once you disconnect your Docker client from Swarm, you should be able to see your Swarm manage container. This is because the container is running on the Docker Daemon on our master host, which is not part of the cluster.

**2) Stop the container which is running Swarm**

```
docker@52.26.42.249 ~: docker stop modest_davinci
modest_davinci
```

**3) Start another container running the swarm manage process but specify the binpack strategy. Remember to use the same cluster token you created earlier and to specify the port mapping**
`docker run -d -P swarm manage token://$(cat swarmtoken) --strategy binpack`

```
docker@52.26.42.249 ~: docker run -d -P swarm manage token://$(cat
swarmtoken) --strategy binpack
810fc71d4db3c42535c28f96673cf052ea4d67a8c504f90e1d84126b599f369b
```

**4) Run `docker ps` to find out what host port has been mapped to the TCP port 2375 on the container**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                ...    PORTS
NAMES
810fc71d4db3        swarm:latest        "/swarm manage token  ...
0.0.0.0:32770->2375/tcp   clever_shockley
```

Port 2375 has been mapped to 32770

**5) Set the `DOCKER_HOST` variable to connect the Docker client to Swarm**
`export DOCKER_HOST=127.0.0.1:<port>`
When you run `docker info` afterwards, you should see the binpack strategy indicated

```
docker@52.26.42.249 ~: export DOCKER_HOST=127.0.0.1:32770
docker@52.26.42.249 ~: docker info
Containers: 3
 Running: 2
 Paused: 0
 Stopped: 1
Images: 5
Server Version: swarm/1.2.3
Role: primary
Strategy: binpack
Filters: health, port, containerslots, dependency, affinity, constraint
Nodes: 2
 node-1: 10.0.18.36:2375
   ID: GIAS:R7OB:CERX:32F2:D4A7:ZHAG:42QJ:XUS4:L25U:7IPJ:R6NM:QJ2C
   Status: Healthy
   Containers: 2
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 3.859 GiB
   Labels: executiondriver=, kernelversion=4.2.0-23-generic,
operatingsystem=Ubuntu 14.04.4 LTS, storagedriver=aufs
   UpdatedAt: 2016-06-07T05:44:07Z
   ServerVersion: 1.11.1
 node-2: 10.0.32.57:2375
   ID: HRWT:XHQZ:DPJM:KDWG:DF6I:QJUW:YSQM:FMBQ:TGIR:XO5V:H7QV:Z7SW
   Status: Healthy
   Containers: 1
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 3.859 GiB
   Labels: executiondriver=, kernelversion=4.2.0-23-generic,
operatingsystem=Ubuntu 14.04.4 LTS, storagedriver=aufs
   UpdatedAt: 2016-06-07T05:43:41Z
   ServerVersion: 1.11.1
Plugins:
 Volume:
 Network:
Kernel Version: 4.2.0-23-generic
Operating System: linux
Architecture: amd64
CPUs: 2
Total Memory: 7.718 GiB
Name: 327acad29e41
Docker Root Dir:
Debug mode (client): false
Debug mode (server): false
WARNING: No kernel memory limit support
```

# Exercise 8.7 – Use binpack strategy

**Using master AWS instance**

**1) Run `docker info` to check the number of containers on each node. There should zero containers at the moment**

```
docker@52.26.42.249 ~: docker info
Containers: 0
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

**2) Run two NGINX containers in detached mode**

```
docker@52.26.42.249 ~: docker run -d nginx
299a2c2d90675f05beb16a240f1c78c2b929bffff767590cad458841308a8668
docker@52.26.42.249 ~: docker run -d nginx
62d489654f965aa1f0939e7911b94156ea77f14dd84dcab06cd4f1fbbb931ca3
```

**3) Use docker info again to check where the containers have been scheduled. Notice they are on the same node**

```
docker@52.26.42.249 ~: docker info
Containers: 2
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 2
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

**4) Run three more NGINX containers and check which node they are scheduled on. You should observe that all your containers are on the same node**

```
docker@52.26.42.249 ~: docker run -d nginx
d00959575d998ab61e5923db3788ac31d75b47db27aa54343d66ee7b8bb13daf
docker@52.26.42.249 ~: docker run -d nginx
b19073a6bf2f1f12812347b996627f632e8fea0eebd81793197386828697dcae
docker@52.26.42.249 ~: docker run -d nginx
c2244c4736d7ea675499103f55bf6c97df34756e395a5fba1df081c163221343
docker@52.26.42.249 ~: docker info
Containers: 5
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 5
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

**5) Stop all containers and then delete them**

```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
```

We will use the shortcut method

```
docker@52.26.42.249 ~: docker rm -f $(docker ps -aq)
c2244c4736d7
b19073a6bf2f
d00959575d99
62d489654f96
299a2c2d9067
```

**6) Now run an NGINX container but specify a memory limit of 200mb**

```
docker run -d -m 200mb nginx
```

In our case we will actually specify 1500mb since our hosts have around 3.8 GB or RAM. Your instructor can advise you on how much memory to specify

```
docker@52.26.42.249 ~: docker run -d -m 1500mb nginx
6be73a1f7b64625a7c1ec2247562208702393e2fbf8ee28ac895b55ab9ea288f
```

**7) Check which node the container has been scheduled on**

```
docker@52.26.42.249 ~: docker info
Containers: 1
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 1.465 GiB / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

Notice the container running on 52.26.34.74 and notice the reserved memory being used.

**8) Repeat 6) and check that the container has been scheduled on the same node as 7)**

```
docker@52.26.42.249 ~: docker run -d -m 1500mb nginx
b9e303fe269ee7b6c65cd2efc9d9f95a8bc7c22b92fcd19d0d9186e4850379b6
docker@52.26.42.249 ~: docker info
Containers: 2
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 2
   Reserved CPUs: 0 / 2
   Reserved Memory: 2.93 GiB / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

You can now see two containers on 52.26.34.74 and also notice that it is almost out of reserved memory.

**9) Repeat 6) again and verify that this new container is running on your other node**

```
docker@52.26.42.249 ~: docker run -d -m 1500mb nginx
d501c71130162033e5bd0901cdc7c94724eb491f34595964ccf1b4a6390ed7b5
docker@52.26.42.249 ~: docker info
Containers: 3
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 2
   Reserved CPUs: 0 / 2
   Reserved Memory: 2.93 GiB / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 1.465 GiB / 3.86 GiB
```

This time, our container is scheduled on 52.26.37.124 because our first host is out of reserved memory.

**10) Clean up: Stop and delete all containers from the cluster**

```
docker@52.26.42.249 ~: docker rm -f $(docker ps -aq)
```

# Exercise 8.8 – Label the Docker daemon

**Using node 1**

**1) Open the `/etc/default/docker` file and add the `--label` flag to `DOCKER_OPTS`. Specify a key value of `region=us`**

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 -H tcp://0.0.0.0:2375 --label region=us”
```

**2) Restart the Docker service**

Run

```
sudo service docker restart
```

**3) Run `docker info` to confirm the label is present**

```
docker@52.26.37.124 ~: docker info
Containers: 0
Images: 136
Storage Driver: aufs
 Root Dir: /mnt/docker/aufs
 Backing Filesystem: extfs
 Dirs: 136
 Dirperm1 Supported: false
Execution Driver: native-0.2
Kernel Version: 3.13.0-40-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 2
Total Memory: 3.676 GiB
Name: 52.26.37.124
ID: KMV2:V43S:ATQN:6VZ7:NSGK:ZJOQ:U6EL:5QHD:EKVU:SNQT:EL7D:3A42
WARNING: No swap limit support
Labels:
 region=us
docker@52.26.37.124 ~:
```

## Exercise 8.9 – Specify constraint filter

**Using master AWS instance**

**1) Disconnect the Docker client from Swarm**

```
docker@52.26.42.249 ~: unset DOCKER_HOST
```

**2) Stop the container which is running the Swarm manage process**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                ...    NAMES
810fc71d4db3        swarm:latest        "/swarm manage token   ...
clever_shockley
docker@52.26.42.249 ~: docker stop clever_shockley
clever_shockley
```

**3) Start a new container to run the Swarm manage process and use the spread scheduling strategy.**

`docker run -d -P swarm manage token://$(cat swarmtoken)`

```
docker@52.26.42.249 ~: docker run -d -P swarm manage token://$(cat
swarmtoken)
d93e686936f49498e673aa4bd1bb9e3e5998fc106fb84b307600c15d65959180
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED
STATUS              PORTS                     NAMES
d93e686936f4        swarm:latest        "/swarm manage token   1 seconds
ago       Up 1 seconds        0.0.0.0:32771->2375/tcp   clever_mestorf
```

**4) Run an NGINX container and specify the constraint filter of region=us**

`docker run -d -e constraint:region==us nginx`

First we need to connect the Docker client back to the Swarm manager

```
docker@52.26.42.249 ~: export DOCKER_HOST=127.0.0.1:32771
```

Now we can run the container

```
docker@52.26.42.249 ~: docker run -d -e constraint:region==us nginx
9d6c76d7ca588a7138e68ad05604da54c84a89b48597d5a6bc58be72af5c00d0
```

**5) Verify that the container runs on node 1**

If we run `docker info`, we should see the container on node 1, which is our node with the IP of 52.26.37.124

```
docker@52.26.42.249 ~: docker info
Containers: 1
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

**6) Repeat step 4 and once again confirm that the container runs on node 1**

```
docker@52.26.42.249 ~: docker run -d -e constraint:region==us nginx
3427f47b903d7f045b0e44dc2427c2ee3f55df10c68fec23abf7cef9a71ed9a6
docker@52.26.42.249 ~: docker info
Containers: 2
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 2
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

As you can see from the output, both containers run on the node with IP 52.26.37.124. Normally with the spread strategy, the containers would be evenly distributed. But because we specify the constraint, it can only run on the node where the Docker daemon has our specified label.

**7) Clean up: Stop and delete all containers**

```
docker@52.26.42.249 ~: docker rm -f $(docker ps -aq)
3427f47b903d
9d6c76d7ca58
```

## Exercise 8.10 – Container affinity

**Using master AWS instance**

**1) Run a Tomcat container and give it the name of "appserver"**

```
docker@52.26.42.249 ~: docker run -d --name appserver tomcat
e4b74b95f226e369940232e440a7d3ea8475a339719e115c61687439cab88357
```

**2) Check which node the container is scheduled on**

```
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND             ...
NAMES
e4b74b95f226        tomcat:latest       "catalina.sh run"    ...
52.26.34.74/appserver
```

The container has been placed on the node with IP 52.26.34.74

**3) Now run an NGINX container and specify a container affinity so that it is scheduled on the same node as the "appserver" container**
`docker run -d -e affinity:container==appserver nginx`

```
docker@52.26.42.249 ~: docker run -d -e affinity:container==appserver nginx
92e797a027750691edd69132d72bad6a5849369adb3ee657d4ab2d639fb324ff
docker@52.26.42.249 ~:
docker@52.26.42.249 ~: docker ps
CONTAINER ID        IMAGE               COMMAND                  ....
NAMES
92e797a02775        nginx:latest        "nginx -g 'daemon of     ....
52.26.34.74/backstabbing_almeida
e4b74b95f226        tomcat:latest       "catalina.sh run"        ....
52.26.34.74/appserver
```

You can see how our new container is running on the Node with IP of 52.26.34.74, which is the same as our Tomcat container

**4)Verify that the NGINX container is running on the same node as the Tomcat container**

See output for 3)

**5) Clean up: stop and delete all containers**

Run
`docker rm -f $(docker ps -aq)`

# Exercise 8.11 – Port filtering

**1) Run an NGINX container and map the container port 80 to port 80 on the host**

```
docker@52.26.42.249 ~: docker run -d -p 80:80 nginx
b5383047bf6744701917387244d1069db061530073cf604898de84c282aea9b8
```

**2) Check which node the container is scheduled on**

We will use `docker info` to see how many containers are on each node

```
docker@52.26.42.249 ~: docker info
Containers: 1
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 0
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

You can see how the container is running on 52.26.37.124

**3) Repeat step 1). This time you should see that the container is running on the other node, because port 80 on the first node has been taken.**

```
docker@52.26.42.249 ~: docker run -d -p 80:80 nginx
92e0005de06b7e95c3b2777e65322199f5a0d4e44692275550c5edc287669d8a
docker@52.26.42.249 ~: docker info
Containers: 2
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
 52.26.34.74: 52.26.34.74:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
 52.26.37.124: 52.26.37.124:2375
   Containers: 1
   Reserved CPUs: 0 / 2
   Reserved Memory: 0 B / 3.86 GiB
```

We now have 1 container on each node.

**4) Repeat 1) again. This time you should notice that Swarm refuses to schedule the container. Why is that so?**

```
docker@52.26.42.249 ~: docker run -d -p 80:80 nginx
FATA[0000] Error response from daemon: unable to find a node with port 80
available
```

Neither of our nodes have a port 80 available

**5) Run another NGINX container but this time, map the container port 80 to port 8080 on the host. Swarm will schedule this because we have not used up port 8080 on either of our nodes**

`docker run -d -p 8080:80 nginx`

```
docker@52.26.42.249 ~: docker run -d -p 8080:80 nginx
04b02c90c50ff1af7ab196fd573943edb285a3c4d23752d344da5907410a2c9a
```

Port 8080 is still available on either Node, so Swarm is able to schedule the container.

**6) Clean up: Stop and delete all containers**

Run
`docker rm -f $(docker ps -aq)`

# Exercise 9.1 - Install Node.js and sample code

**1) Install Node.js on your AWS instance**
`sudo apt-get install nodejs`

**2)Install the Node.js package manager**
`sudo apt-get install npm`

**3) In your home folder, check out the source code of our application**
`git clone https://github.com/johnny-tu/inventory-service.git`

Install git if it is not on your host machine
`sudo apt-get install git`

Then clone the repository. When the command finishes you should have a folder titled "inventory-server"

```
docker@52.26.42.249 ~: git clone
https://github.com/johnny-tu/inventory-service.git
Cloning into 'inventory-service'...
remote: Counting objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Unpacking objects: 100% (7/7), done.
Checking connectivity... done.
docker@52.26.42.249 ~: ls
docker-ca  docker-machine  inventory-service  javahelloworld  myimage
swarmtoken
docker@52.26.42.249 ~:
```

**4) Change directory into the inventory-service folder**

**5) Install the node_redis package using the Node package manager**
The command to run is:
`npm install redis`

Make sure you run this in your inventory-service folder

```
docker@52.26.42.249 ~: cd inventory-service/
docker@52.26.42.249 ~/inventory-service: npm install redis
npm http GET https://registry.npmjs.org/redis
npm http 200 https://registry.npmjs.org/redis
npm http GET https://registry.npmjs.org/redis/-/redis-0.12.1.tgz
npm http 200 https://registry.npmjs.org/redis/-/redis-0.12.1.tgz
redis@0.12.1 node_modules/redis
```

# Exercise 9.2 - Run the code

**1) Run a Redis server inside a container. Use the redis image and map the container port 6379 to host port 6379. Remember to run in detached mode. Give the container the name "redisdb"**

`docker run -d -p 6379:6379 --name redisdb redis`

```
 docker@52.26.42.249 ~/inventory-service: docker run -d -p 6379:6379 --name
redisdb redis
a2b8bceb3bcf30d45f75081dd73bd3837abc9da72a4a6f2b77762915107043ea
```

**2) Run the Node.js application**

The command is:

`nodejs inventoryService.js`

**3) Verify that you see the following output**

```
docker@52.26.42.249 ~/inventory-service: nodejs inventoryService.js
connected
Reply: OK
books_count = 123
```

Hit CTRL + C to end the process

# Exercise 9.3 – Dockerize the application

**1) Build an image from the given Dockerfile. Name your image repository inventory-service**

```
docker@52.26.42.249 ~/inventory-service: docker build -t inventory-service
.
```

**2) Create a new bridge network called nodeapp**

`$ docker network create nodeapp`

```
student@master:~$ docker network create nodeapp
3ff408c2c66c48d57ca38958fe0cc9300e3fd4a247ab9e0767385269635a3820
student@master:~$ docker network ls
NETWORK ID          NAME                DRIVER
9702b9ca5256        bridge              bridge
68e0020c5089        none                null
ef713a6eeb42        host                host
3ff408c2c66c        nodeapp             bridge
```

**3) Stop the existing redis container and start a new one on the nodeapp network**

```
student@master:~$ docker stop redisdb
redisdb
student@master:~$ docker rm redisdb
redisdb
student@master:~$ docker run -d -p 6379:6379 --name redisdb --net nodeapp
redis
cb7028f64d4c386fbf2501a1da94f662eca8d4e74abb634a53f68433ed612263
```

**4) Run a container from your built image and link it to the Redis container**

The command to run is:
`docker run --link redisdb:redisdb inventory-service`

```
student@master:~$ docker run --net nodeapp inventory-service
events.js:85
      throw er; // Unhandled 'error' event
            ^
Error: Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
    at exports._errnoException (util.js:746:11)
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1010:19)
```

**5) Notice the error message saying that it can't connect to 127.0.0.1:6379. This is because our code doesn't take into account the link we established.**

See output for 4)

**6) Open the inventoryService.js file and change the line**
`var client = redis.createClient();`
into
`var client = redis.createClient(6379, "redisdb");`

Notice how we are specifying a connection to Redis, using the same name as our redis container, which is `redisdb`. Your inventoryService.js file should look like this at the end

```
var redis = require('redis');
var client = redis.createClient(6379, "redisdb");
client.on('connect', function() {
    console.log('connected');
});
client.set("books_count", "123", redis.print);
client.get("books_count", function (err, reply) {
    console.log('books_count = ' + reply.toString());
});
```

**7) Build the image again**

```
student@master:~/inventory-service$ docker build -t inventory-service .
```

**8) Repeat step 2. Run a container from your built image and link it to the Redis container.**

```
student@master:~/inventory-service$ docker run --net nodeapp
inventory-service
connected
Reply: OK
books_count = 123
```

This time the container works properly because we have established a link with our Redis server. The connection details we specified in our code matched with the alias name we used in the container link

# Exercise 10.1 – Install Docker Compose

**1) Download the Docker Compose Linux binary from https://github.com/docker/compose/releases**

```
wget
https://github.com/docker/compose/releases/download/1.2.0/docker-compose-L
inux-x86_64
```

**2) Rename the binary file to docker-compose and place it into your path at /usr/local/bin**

```
student@master:~$ mv docker-compose-Linux-x86_64 docker-compose
student@master:~$ mv docker-compose /usr/local/bin
```

**2) Set the correct file permission**

```
sudo chmod +x /usr/local/bin/docker-compose
```

**4) Verify the installation by running `docker-compose --version`**

```
student@master:~$ docker-compose --version
docker-compose version 1.5.2, build 7240ff3
student@master:~$
```

## Exercise 10.2 – Try the example manually

**1) In your home directory, download the sample code**
`git clone https://github.com/johnny-tu/HelloRedis.git`

student@master:~$

```
student@master:~$ git clone https://github.com/johnny-tu/HelloRedis.git
Cloning into 'HelloRedis'...
remote: Counting objects: 42, done.
remote: Total 42 (delta 0), reused 0 (delta 0), pack-reused 42
Unpacking objects: 100% (42/42), done.
Checking connectivity... done.
student@master:~$ ls
docker-ca  HelloRedis  inventory-service  javahelloworld  myimage
swarmtoken  tmp
```

**2) Build the image using the Dockerfile. Name your image "helloredis"**
`docker build -t helloredis .`

We need to be in the HelloRedis folder to do this

```
student@master:~/HelloRedis$ docker build -t helloredis .
```

**3) Create a new bridge network called java_app**
`$ docker network create –d bridge java_app`

```
student@master:~/HelloRedis$ docker network create -d bridge java_app
598463c23f349db067f5c7e0c1cf2a915fe349d89f2347e110de92043dbe817a
student@master:~/HelloRedis$ docker network ls
NETWORK ID          NAME                DRIVER
3ff408c2c66c        nodeapp             bridge
598463c23f34        java_app            bridge
9702b9ca5256        bridge              bridge
68e0020c5089        none                null
ef713a6eeb42        host                host
student@master:~/HelloRedis$
```

**4) Run a Redis container called redisdb on your java_app network**

`$ docker run -d --name redisdb --net java_app redis`

```
student@master:~/HelloRedis$ docker run -d --name redisdb --net java_app
redis
491823de17e6395270e2f570e77dff62b62ced7a6075026d72b0f9dbb69fee3e
```

**5) Run your helloredis image on the java_app network container. Verify that the command line output tells you are connected to Redis. Remember to specify a name for the container**

`$ docker run –-name client --net java_app helloredis`

```
student@master:~/HelloRedis$ docker run --name client --net java_app
helloredis
Server is running: PONG
books_count = null
Server is running: PONG
books_count = null
```

## Exercise 10.3 – Compose our application

**1) Shut down your existing redis container**

```
docker@52.26.42.249 ~: docker stop redisdb
```

**2) Run the example application**

The `docker-compose.yml` has already been written. All we have to do is run it. Make sure you are in the HelloRedis folder.

```
docker@52.26.42.249 ~/HelloRedis: docker-compose up
Creating helloredis_redis_1...
Creating helloredis_javaclient_1...
Building javaclient...
....
....
Successfully built 9df932011321
Attaching to helloredis_redis_1, helloredis_javaclient_1
redis_1      | 1:C 18 Jun 06:37:01.286 # 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 3.0.1
(00000000/0) 64 bit
redis_1      |   .-`` .-```.  ```\/    _.,_ ''-._
redis_1      |  (    '      ,       .-`  | `,    )     Running in
standalone mode
redis_1      |  |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1      |  |    `-._   `._    /     _.-'    |     PID: 1
redis_1      |   `-._    `-._  `-./  _.-'    _.-'
redis_1      |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1      |  |    `-._`-._        _.-'_.-'    |
http://redis.io
redis_1      |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1      |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1      |  |    `-._`-._        _.-'_.-'    |
redis_1      |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1      |       `-._    `-.__.-'    _.-'
redis_1      |           `-._        _.-'
redis_1      |               `-.__.-'
redis_1      |
redis_1      | 1:M 18 Jun 06:37:01.288 # Server started, Redis version
3.0.1
redis_1      | 1:M 18 Jun 06:37:01.288 # WARNING overcommit_memory is set
to 0! Background save may fail under low memory condition. To fix this
issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or
run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1      | 1:M 18 Jun 06:37:01.288 # WARNING you have Transparent Huge
Pages (THP) support enabled in your kernel. This will create latency and
memory usage issues with Redis. To fix this issue run the command 'echo
never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to
your /etc/rc.local in order to retain the setting after a reboot. Redis
must be restarted after THP is disabled.
redis_1      | 1:M 18 Jun 06:37:01.288 # WARNING: The TCP backlog setting
of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to
the lower value of 128.
redis_1      | 1:M 18 Jun 06:37:01.288 * The server is now ready to accept
connections on port 6379
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
```

**3) Check the output on your terminal. You should see the output of the Redis server starting up, followed by the output of our Java client**

**4) Hit CTRL + C to terminate the services**

## Exercise 10.4 – Running in detached mode

**1) Remove the two services from the previous exercise**
docker-compose rm

```
docker@52.26.42.249 ~/HelloRedis: docker-compose rm
Going to remove helloredis_javaclient_1, helloredis_redis_1
Are you sure? [yN] y
Removing helloredis_redis_1...
Removing helloredis_javaclient_1...
docker@52.26.42.249 ~/HelloRedis:
```

**2) Start the example application again but this time, run in detached mode**
`docker-compose up -d`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose up -d
Creating helloredis_redis_1...
Creating helloredis_javaclient_1...
docker@52.26.42.249 ~/HelloRedis:
```

**3) Run docker-compose ps and check that both the javaclient and redis service are running**

```
docker@52.26.42.249 ~/HelloRedis: docker-compose ps
         Name                       Command             State    Ports
-------------------------------------------------------------------------
helloredis_javaclient_1   java HelloRedis                Up
helloredis_redis_1        /entrypoint.sh redis-server    Up      6379/tcp
```

**4) Check the log for the javaclient**
`docker-compose logs javaclient`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose logs javaclient
Attaching to helloredis_javaclient_1
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
....
....
```

**5) Stop the javaclient service**

`docker-compose stop javaclient`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose stop javaclient
Stopping helloredis_javaclient_1...
```

**6) Start the javaclient service again**
`docker-compose start`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose start
Starting helloredis_javaclient_1...
```

## Exercise 10.5 – Scaling services

**1) Scale the javaclient service to run 5 containers**
`docker-compose scale javaclient=5`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose scale javaclient=5
Creating helloredis_javaclient_2...
Creating helloredis_javaclient_3...
Creating helloredis_javaclient_4...
Creating helloredis_javaclient_5...
Starting helloredis_javaclient_2...
Starting helloredis_javaclient_3...
Starting helloredis_javaclient_4...
Starting helloredis_javaclient_5...
```

**2) Check your containers with `docker-compose ps`**

```
docker@52.26.42.249 ~/HelloRedis: docker-compose ps
         Name                       Command             State    Ports
-------------------------------------------------------------------------
helloredis_javaclient_1   java HelloRedis                Up
helloredis_javaclient_2   java HelloRedis                Up
helloredis_javaclient_3   java HelloRedis                Up
helloredis_javaclient_4   java HelloRedis                Up
helloredis_javaclient_5   java HelloRedis                Up
helloredis_redis_1        /entrypoint.sh redis-server   Up      6379/tcp
```

**3) View the aggregated log of your javaclient service**

This command will display the logs for all the javaclient containers
`docker-compose logs javaclient`

```
docker@52.26.42.249 ~/HelloRedis: docker-compose logs javaclient
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
javaclient_2 | Server is running: PONG
javaclient_2 | books_count = null
javaclient_3 | Server is running: PONG
javaclient_3 | books_count = null
javaclient_4 | Server is running: PONG
javaclient_4 | books_count = null
javaclient_5 | Server is running: PONG
javaclient_5 | books_count = null
....
....
```

**4) Pick any one of the javaclient containers and display the log output for just that container**

To view the log for an individual container, we have to reference the container name or ID. We also have to use the Docker client, not Docker Compose

```
docker@52.26.42.249 ~/HelloRedis: docker logs helloredis_javaclient_5
Server is running: PONG
books_count = null
Server is running: PONG
books_count = null
```

# Exercise 10.6 – Compose and Networking

**1) In the HelloRedis example, modify the docker-compose.yml file.**
**Delete the "`links`" instruction from the `javaclient` service**

**2) In the `redisdb` service, add a `container_name` instruction and name the container `redisdb`**
`container_name: redisdb`

**3) Your docker-compose.yml file should look like this:**

```
version: '2'
    services:
      javaclient:
        build: .
      redis:
        image: redis
        container_name: redisdb
```

**4) Run `docker-compose up -d`**

```
student@master:~/HelloRedis$ docker-compose up -d
Creating redisdb
Building javaclient
Step 1 : FROM java:7
 ---> 4f896d7ad04f
Step 2 : COPY /src /HelloRedis/src
 ---> a7eec470e34c
Removing intermediate container 1d3b9b7c26a6
Step 3 : COPY /lib /HelloRedis/lib
 ---> 0dce8ec8e4d0
Removing intermediate container 65e2d1fbd190
Step 4 : WORKDIR /HelloRedis
 ---> Running in ef19ac5677c3
 ---> 364d50835666
Removing intermediate container ef19ac5677c3
Step 5 : RUN javac -cp lib/jedis-2.1.0-sources.jar -d . src/HelloRedis.java
 ---> Running in 468cce0c7f74
 ---> 53764a85a81f
Removing intermediate container 468cce0c7f74
Step 6 : CMD java HelloRedis
 ---> Running in d39fb66c2755
 ---> b1a1b2bc2004
Removing intermediate container d39fb66c2755
Successfully built b1a1b2bc2004
Creating helloredis_javaclient_1
```

**5) Run `docker network ls` and verify that you can see a bridge network named `helloredis_default`**

```
student@master:~/HelloRedis$ docker network ls
NETWORK ID          NAME                DRIVER
1d2614eafd68        none                null
5b54402b06fe        host                host
cb865585602f        helloredis_default  bridge
3ff408c2c66c        nodeapp             bridge
598463c23f34        java_app            bridge
a33a22ecb70a        bridge              bridge
```

**6) Inspect the newly created containers and verify that they are on the helloredis network**

```
$ docker inspect helloredis_javaclient_1
$ docker inspect redisdb
```

For both containers, look for the "`Networks`" Field in the output.

```
student@master:~/HelloRedis$ docker inspect helloredis_javaclient_1
[
{
    "Id":
"3024393b0ae29f4a99014879ba245f524201448b782999301b728a3852b7047b",
    "Created": "2016-01-25T03:45:47.585959266Z",
.....
.....
        "Networks": {
            "helloredis_default": {
                "EndpointID":
"90d0c039d1899f56879a3f51efca69dec736ac63ce1ec01622b6c7e868e13f01",
                "Gateway": "172.20.0.1",
                "IPAddress": "172.20.0.3",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:14:00:03"
            }
        }
    }
}
]
student@master:~/HelloRedis$
student@master:~/HelloRedis$ docker inspect redisdb
[
{
    "Id":
"72001e20fe27be5afeedea764881c506abb010f60833de672df29f7d1b84575e",
    "Created": "2016-01-25T03:45:39.57340742Z",
....
....
        "Networks": {
            "helloredis_default": {
                "EndpointID":
"c8bcf849bbcc93a645621dfa873e9232b9f0bcbee9f5fb625bd9c46085f47c42",
                "Gateway": "172.20.0.1",
                "IPAddress": "172.20.0.2",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:14:00:02"
            }
        }
    }
}
]
student@master:~/HelloRedis$
```

**7) Check the javaclient log and make sure it is able to ping the redisdb container. The output should say:**
`Server is running: PONG`
`books_count = null`

```
student@master:~/HelloRedis$ docker logs helloredis_javaclient_1
Server is running: PONG
books_count = null
Server is running: PONG
books_count = null
Server is running: PONG
books_count = null
....
....
```

For clean up at the end of this exercise you should stop and remove both services

```
student@master:~/HelloRedis$ docker-compose stop
Stopping helloredis_javaclient_1 ... done
Stopping redisdb ... done
student@master:~/HelloRedis$ docker-compose rm
Going to remove helloredis_javaclient_1, redisdb
Are you sure? [yN] y
Removing helloredis_javaclient_1 ... done
Removing redisdb ... done
student@master:~/HelloRedis$
```

# Exercise 10.7 – Compose on Swarm

> You need to have successfully completed exercise 3.1 from the Multi host networking module and exercise 8.1 and 8.2 from the Swarm module in order for the examples in this exercise to work properly

**1) Build the image defined in the Dockerfile of the HelloRedis application folder. Tag the image with your Docker Hub account name**
`$ docker build -t <docker hub login>/hello-redis:1.0 .`

**2) Push the image to Docker Hub**
`docker push <docker hub login>/hello-redis:1.0`

**3) Open the docker-compose.yml file.**

**4) Modify the javaclient service by replacing the build parameter with an image parameter and specify the image you've just built and pushed**
`image: <docker hub login>/hello-redis:1.0`

In this example, we are using the `trainingteam` Docker Hub login. The docker-compose.yml file should now look like:

```
javaclient:
  image: trainingteam/hello-redis:1.0
redis:
  image: redis
  container_name: redisdb
```

5) Connect the Docker client to Swarm

On your master node, your Swarm manager container should still be up an running. Let's do a quick check on the container. (If your Swarm manager is not up and running, you will need to setup your Swarm manager and nodes again by following the steps in Exercise 8.3)

```
student@master:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                           NAMES
d4fb6a22c845        swarm               "/swarm manage token:"   9 minutes
ago       Up 9 minutes        0.0.0.0:32768->2375/tcp        elated_keller

student@master:~$ docker inspect elated_keller
[
{
…
 "Networks": {
            "my_network": {
                "EndpointID":
"9914ad736bfb38c41fc0678345c8421cc14ca55e9d1e8eafef09e890d8d4f890",
                "Gateway": "",
                "IPAddress": "10.0.0.2",
                ...
```

Our Swarm manager container is mapped to port 32768. To connect our client we set the DOCKER_HOST variable to <ip>:<port> for our Swarm managaer container, elated_keller.

```
$ export DOCKER_HOST=10.0.0.2:32768
```

**6) Now run `docker-compose up -d`**

Remember to change back into the HelloRedis folder when running this command.

**Note:** For this exercise to work, you must have the same number of containers on each node before running the command. That is because we are using the Swarm spread strategy. If one node has more containers than the other, then Compose will end up spinning both services into the same Node, which is not the result we are after.

```
student@master:~/HelloRedis$ docker-compose up -d
Pulling redis (redis:latest)...
node-2: Pulling redis:latest... : downloaded
node-1: Pulling redis:latest... : downloaded
Creating redisdb
Pulling javaclient (trainingteam/hello-redis:1.0)...
node-2: Pulling trainingteam/hello-redis:1.0... : downloaded
node-1: Pulling trainingteam/hello-redis:1.0... : downloaded
Creating helloredis_javaclient_1
student@master:~/HelloRedis$
```

**7) Run `docker ps` and check that your two containers are on different nodes**

```
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS              PORTS                NAMES
7bf02ce01201        trainingteam/hello-redis:1.0   "java HelloRedis"
2 seconds ago       Up 2 seconds
node-1/helloredis_javaclient_1
d7ecc9d8965c        redis                              "/entrypoint.sh redis"
3 seconds ago       Up 2 seconds        6379/tcp             node-2/redisdb
```

**8) Check the logs to your `helloredis_javaclient_1` container. You will notice that it fails to connect to `redisdb` because the `redis` container is on a different node**

```
$ docker logs helloredis_javaclient_1
```

```
student@master:~/HelloRedis$ docker logs helloredis_javaclient_1
java.net.UnknownHostException: redisdb
java.net.UnknownHostException: redisdb
java.net.UnknownHostException: redisdb
java.net.UnknownHostException: redisdb
.....
.....
```

**9) Stop and remove the services**

```
$ docker-compose stop
$ docker-compose rm -f
```

**10) Run the application again but this time, use the compose networking option. It will default to the overlay driver if you have the requisite key value store available.**

```
$ docker-compose up -d
```

```
student@master:~/HelloRedis$ docker-compose up -d
Creating network "helloredis_default" with driver "overlay"
Creating redisdb
Creating helloredis_javaclient_1
student@master:~/HelloRedis$
```

**11) Run `docker network ls` and check that you have an overlay network named `helloredis_default`**

```
student@master:~/HelloRedis$ docker network ls
NETWORK ID          NAME                      DRIVER
9a5fa1018bd3        node-2/docker_gwbridge    bridge
c46c6f92f295        node-2/bridge             bridge
2b4f7a3fa403        node-1/bridge             bridge
01200ab2df58        node-1/host               host
978448adec82        node-1/docker_gwbridge    bridge
2435db48aeaf        multinet                  overlay
6323880c3e6d        node-2/none               null
06ea9f238b31        node-1/none               null
493023585f6c        test-overlay              overlay
5ec21a0a6c47        helloredis_default        overlay
a49d0b1eebd8        node-2/host               host
student@master:~/HelloRedis$
```

**12) Inspect the `javaclient` container and verify that it is running on the `helloredis_default` overlay network**

```
$ docker inspect helloredis_javaclient_1
```

```
student@master:~/HelloRedis$ docker inspect helloredis_javaclient_1
[
{
    "Id":
"0a1d5b741b5b083e74feb3fae043ca38ef523eba736aa464da95da9a69dd7a6b",
    "Created": "2016-01-25T05:53:21.032657825Z",
....
....
        "Networks": {
            "helloredis_default": {
                "EndpointID":
"8b784e8dacef7de0f14ffa6d99375e738f2e25bbe2dd8a7a9dbba8e3cd4c81b8",
                "Gateway": "",
                "IPAddress": "10.0.1.3",
                "IPPrefixLen": 24,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:0a:00:01:03"
            }
        }
    }
}
]
```

**13) Check the javaclient container log and verify that it can reach the redisdb container.**

```
$ docker logs helloredis_javaclient_1
```

**14) The logs output should be the following two lines repeated constantly**

```
Server is running: PONG
books_count = null
```

**15) Now let's scale our `javaclient` service to run 5 containers**

```
$ docker-compose scale javaclient=5
```

```
student@master:~/HelloRedis$ docker-compose scale javaclient=5
Creating and starting 2 ... done
Creating and starting 3 ... done
Creating and starting 4 ... done
Creating and starting 5 ... done
student@master:~/HelloRedis$
```

**16) Run `docker ps` to check the containers and verify that the `javaclient` containers are spread across both Swarm nodes**

```
student@master:~/HelloRedis$ docker ps
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS              PORTS               NAMES
c23f44bcc45e        trainingteam/hello-redis:1.0   "java HelloRedis"
34 seconds ago      Up 33 seconds
node-2/helloredis_javaclient_5
0b2f7abdf70b        trainingteam/hello-redis:1.0   "java HelloRedis"
34 seconds ago      Up 33 seconds
node-2/helloredis_javaclient_3
2c69a7566338        trainingteam/hello-redis:1.0   "java HelloRedis"
34 seconds ago      Up 33 seconds
node-1/helloredis_javaclient_4
97ac1cb492ad        trainingteam/hello-redis:1.0   "java HelloRedis"
34 seconds ago      Up 33 seconds
node-1/helloredis_javaclient_2
0a1d5b741b5b        trainingteam/hello-redis:1.0   "java HelloRedis"
5 minutes ago       Up 5 minutes
node-2/helloredis_javaclient_1
36444ad12877        redis                              "/entrypoint.sh redis"
5 minutes ago       Up 5 minutes        6379/tcp            node-1/redisdb
student@master:~/HelloRedis$
```

Note: At the end of step 16, the newly created javaclient containers will not be connected to the “helloredis” multihost network by default. Therefore if you decide to check the container log, there will be an exception message thrown in that the client cannot reach the redisdb service. To fix this, you must manually connect the javaclient service containers to the helloredis network.

Eg.

```
$ docker network connect helloredis helloredis_javaclient_2
$ docker network connect helloredis helloredis_javaclient_3
```