1 Contents

2 Images Exercise 1: Finding Images on DockerHub

1) Create a Docker Hub Account

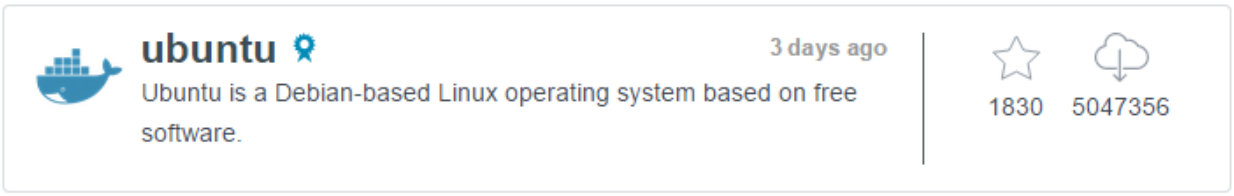
If you have an existing Docker Hub account, please feel free to use it for this exercise and skip this step.

You can sign up for an account at https://hub.docker.com/account/signup/. You will need a valid email address.

2) Search for the official Ubuntu image

Access https://registry.hub.docker.com/ to search for public images.

We can see a tile for the Ubuntu image near the top of the official repositories list; but let's use the text search to check out the interface. Search for 'ubuntu'. The top result in the search should look as follows:



Note the badge next to the name, informing us that it is an official image. We can also see that it was last updated 3 days ago, has been starred 1830 times, and pulled a little over 5 million times.

3) Pull the image

In the previous step, we found that the name of the official Ubuntu image is simply 'ubuntu' via the web search. Note that we can also search from the command line:

```
$ docker search ubuntu
NAME    DESCRIPTION
STARS        OFFICIAL
ubuntu     Ubuntu is a Debian-based Linux operating s...    1830
[OK]
```

Finally, let's pull the image:

```
$ docker pull ubuntu
```

3 Images Exercise 2: Running a Container

1) Run the Ubuntu image as a container

Start by looking at the list of images available locally:

```
$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED       VIRTUAL SIZE
ubuntu              latest    6d4946999d4f   4 days ago    188.3 MB
```

Now, run the container using the following command:

```
$ sudo docker run -t -i ubuntu
```

Lets break that command down:

Component	Function
`docker run <imagename>`	The bare minimum required to run an container. Will start a new container running in the background; behaviour will be dependent on its dockerfile.
`docker run <imagename> -it`	Starts a new container, allocates a tty and connects stdin. The combination of i and t is needed to link the container stdin/stdout to your terminal. Feel free to experiment and try and run a container using none, or just one of the options to see the impact.
`-t`	Allocates a new psuedo-tty to display output. Needed if you intend to use the terminal.

Component	Function
`-i`	Keeps stdin open even if not attached. Also necessary to properly use the terminal.

After running the command, you should see a shell for the Ubuntu instance. Feel free to try some standard commands to inspect the state of the machine.

2) Install curl

You may have noticed that some common commands are not present, as the Ubuntu image is kept as small as possible by default.

Install curl. You will need to respond to some prompts during the install process.

```
# apt-get update
# apt-get install curl
```

3) Stop the container

Use 'exit' to exit the bash prompt for your container and return to your host OS.

Note that once the bash prompt is closed, the container will stop, as it's main process (PID 1) has terminated, We will be exploring the container lifecycle in further detail in the Containers session.

4 Images Exercise 3: Tag and Push to DockerHub

1) Create a new image

Now, we can turn our modified Ubuntu with curl into a new image. Recall that the installation of curl on our container will not record any changes to our local Ubuntu image.

Identify the container that we made the change to using the following command. This will show all containers, running or stopped.

```
$ docker ps -a

CONTAINER ID    IMAGE           COMMAND        CREATED
STATUS                        PORTS        NAMES
942e06e20ab1    ubuntu:latest    "/bin/bash"    5 minutes ago
Exited (0) 2 minutes ago                 condescending_wozniak
```

Based on this output, we will want to work with the container with id 942e06e20ab1.

We can create an image inside a local repository from this stopped container using the following command:

```
$ docker commit 942e06e20ab1 YourDockerHubUser/ubuntucurl:1.0
```

The DockerHubUser/ubuntucurl:1.0 parameters 'tags' the image with relevant details and prepares for upload to Docker Hub. In this case, the image is tagged as ubuntucurl version 1.0.

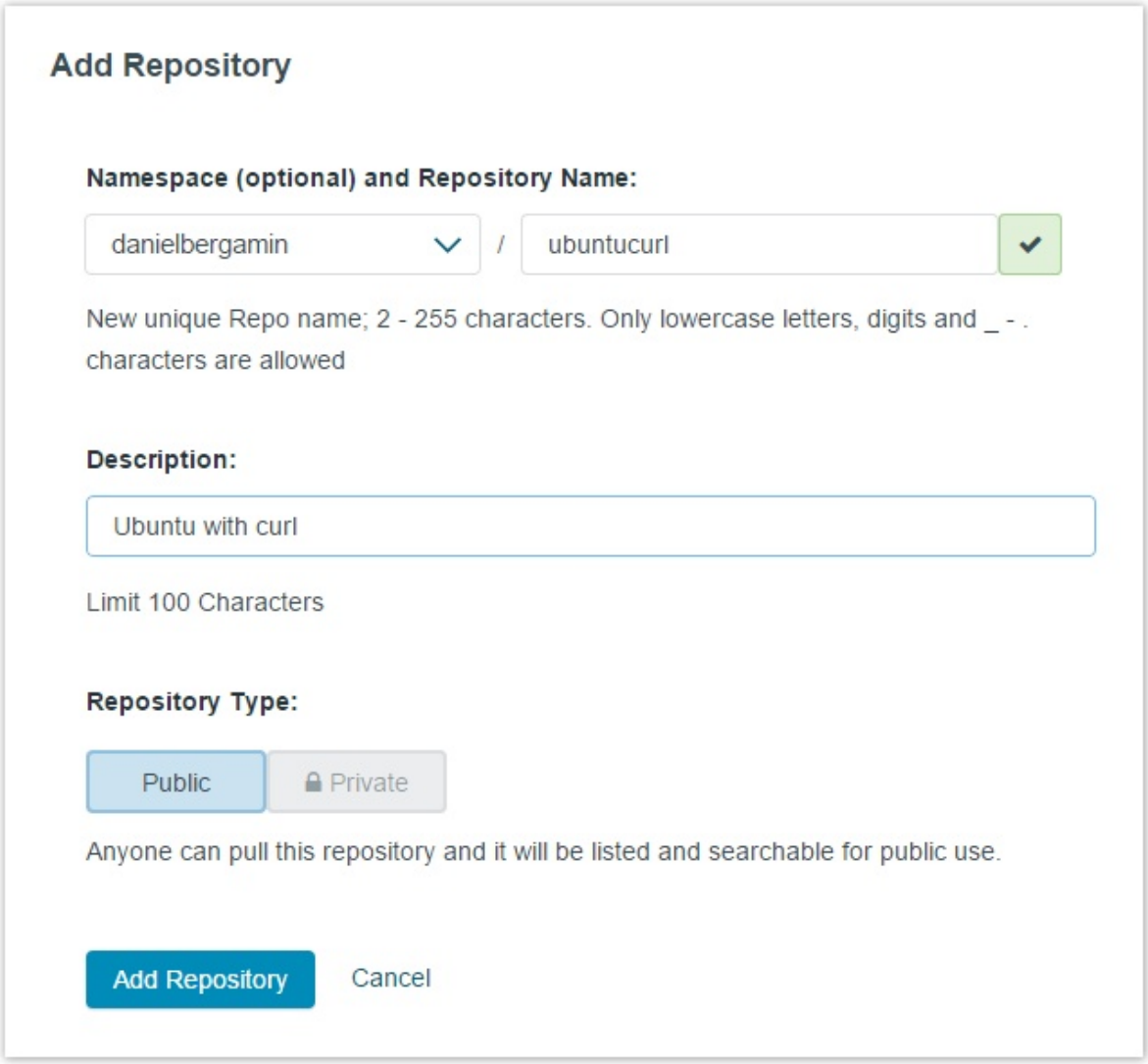
2) Verify by listing your images

Verify that the image was tagged correctly by getting docker to list your images

```
$ docker images
REPOSITORY                        TAG       IMAGE ID
CREATED           VIRTUAL SIZE
YourDockerHubUser/ubuntucurl    1.0        3e8b4d4eb571    2
minutes ago    220.8 MB
ubuntu                            latest  6d4946999d4f    4 days
ago      188.3 MB
```

3) Push the local repository to a public repository on Docker Hub

First, you will need to set up a public repository. Login to Docker Hub and click 'Add Repository'.



Now, use the following command to push to your repository:

```
docker push YourDockerHubUser/ubuntucurl
```

Verify that the push was successful by returning to the DockerHub site and looking at your repository in the browser.

4) [Optional] Share your image

If you have time, attempt to pull and run the image created by by a person next to you - good luck!

5 Dockerfiles Exercise 1: Creating a Sample Application

1) Create a sample application

We are going to write a small Java application that will let us explore some of the functionality available in the Dockerfile.

First, install java 7 on your VM:

```
$ sudo apt-get install openjdk-7-jdk
```

Now, create a folder called javahelloworld in your home directory. In it, create a file called HelloWorld.java:

```
$ mkdir ~/javahelloworld
$ cd ~/javahelloworld/
$ ~/javahelloworld: touch HelloWorld.java
$ vi HelloWorld.java
```

Write the java file as follows:

```
public class HelloWorld {
    public static void main (String [] args) {
        System.out.println("Hello World!");
    }
}
```

2) Compile and run the test program Working in the javahelloworld folder, compile HelloWorld.java:

```
$ javac HelloWorld.java
```

Now, run the program:

```
$ java HelloWorld
Hello World!
```

It should output the text "Hello World!".

6 Dockerfiles Exercise 2: Dockerize your Application

1) Create your Dockerfile

Create a Dockerfile:

```
$ touch Dockerfile
$ vi Dockerfile
```

Add the following instructions. You may skip the comments denoted by a # at the start of the line but be sure to digest them, they provide valuable context:

To start inserting text in vi, press the 'i' key to enter insert mode:

```
FROM java:7
#Use official java 7 image as a starting point - it has a Linux-
based OS with Java7 pre-installed.

COPY HelloWorld.java /
RUN javac HelloWorld.java
#Copy the HelloWorld.java file into the container.
#Upon starting the container, run the shell command to compile
HelloWorld.java.
#The RUN command can be used to run any shell command
ENTRYPOINT ["java", "HelloWorld"]
#ENTRYPOINT commands are given in the form ["executable", "param1"
, "param2"]
#Use ENTRYPOINT to set the default executable, and, optionally,
recommended minimum parameters that your application is intended
to be run with.
```

When you have finished adding the instructions, press ESC to exit insert mode, then type ':wq' and press return to save the file.

2) **Build the Dockerfile**

Now, build the Dockerfile:

```
$ ~/javahelloworld: docker build -t javahelloworld:1.0 .
Sending build context to Docker daemon  5.12 kB
Sending build context to Docker daemon
Step 0 : FROM java:7
7: Pulling from java
a2703ed272d7: Pull complete
...
...
Successfully built 5eeca326d8a7
docker@52.26.42.249 ~/javahelloworld:
```

Run the container and observe its output:

```
$ docker run javahelloworld:1.0
Hello World
```

3) Have a look inside the container

At the moment, our container will launch straight into running the Java application. Sometimes, it may be necessary to head to a terminal instead to troubleshoot.

Docker allows us to override the native entrypoint for the image (default process run on startup) if needed:

```
$ docker run -it --entrypoint bash javahelloworld:1.0
root@7c0ee2270e83:/#
```

Find out where the copied source and compiled class are being stored:

```
root@7c0ee2270e83:/# ls
HelloWorld.class  HelloWorld.java  bin  boot  dev  etc  home  lib
lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr
var
```

You will notice that the files are in the root folder. Not too bad for our two file application, but we might want to get a bit more organised if we were to run anything more complex.

Type exit to terminate the container:

```
root@7c0ee2270e83:/# exit
```

7 Dockerfiles Exercise 3: Restructure the Application

1) Restructure our Application

In your javahelloworld folder, create two folders called src and bin. Move your HelloWorld.java source file into src.

```
$ ~/javahelloworld: mkdir src
$ ~/javahelloworld: mkdir bin
$ ~/javahelloworld: mv HelloWorld.java src/
```

Compile your code and place the compiled class into the bin folder

```
$ ~/javahelloworld: javac -d bin src/HelloWorld.java
```

Now, run your application to test:

```
$ ~/javahelloworld: java -cp bin HelloWorld
Hello World!
```

2) Tweak the Dockerfile accordingly

Let's update the Dockerfile to reflect our new structure.

```
$ ~/javahelloworld: vi Dockerfile
```

Enter insert mode and update the file accordingly:

```
FROM java:7
COPY src /home/root/javahelloworld/src
#Note that we have updated the path

RUN javac -d bin src/HelloWorld.java
#Note that we have updated the path to use the src/bin structure

ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
#Note that we are now dropping the compiled class into a 'bin'
directory in the javac statement above
#The entrypoint has been updated to refer to this new location
```

Now, rebuild the dockerfile using:

```
$ docker build -t javahelloworld:1.1 .
Sending build context to Docker daemon 6.144 kB
Sending build context to Docker daemon
Step 0 : FROM java:7
 ---> beabd1eef902
Step 1 : COPY src /home/root/javahelloworld/src
 ---> Using cache
 ---> b16369f2a3f0
Step 2 : WORKDIR /home/root/javahelloworld
 ---> Using cache
 ---> 50d4e26e04ab
Step 3 : RUN javac -d bin src/HelloWorld.java
 ---> Running in c9bd4cd0cafd
javac: directory not found: bin
Usage: javac <options> <source files>
use -help for a list of possible options
INFO[0000] The command [/bin/sh -c javac -d bin src/HelloWorld.
java] returned a non-zero code: 2
```

Unfortunately, while our application has moved to a more sensible path, Docker is still looking for it in the root directory, /.

To fix this, we will need to use a new instruction, WORKDIR. We can also see that we need to create the bin directory.

3) Add a WORKDIR

Let's update the Dockerfile again to tell it where it should be running the commands:

```
FROM java:7
COPY src /home/root/javahelloworld/src

WORKDIR /home/root/javahelloworld
#Specify a working directory in which the following commands
should be executed

RUN mkdir bin
# Make the bin directory
RUN javac -d bin src/HelloWorld.java
ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
```

Build your image again. This time it should work without issues:

```
$ docker build -t javahelloworld:1.1 .
Sending build context to Docker daemon 5.632 kB
Sending build context to Docker daemon
Step 0 : FROM java:7
 ---> 39b678444b33
Step 1 : COPY src /home/root/javahelloworld/src
 ---> Using cache
 ---> 72d09169279a
Step 2 : WORKDIR /home/root/javahelloworld
 ---> Running in 1f0063837f8b
 ---> fcfd2f3e2738
Removing intermediate container 1f0063837f8b
Step 3 : RUN mkdir bin
 ---> Running in 95b850ad98bf
 ---> 211429de9147
Removing intermediate container 95b850ad98bf
Step 4 : RUN javac -d bin src/HelloWorld.java
 ---> Running in 488112451ca2
 ---> fcc168817e02
Removing intermediate container 488112451ca2
Step 5 : ENTRYPOINT java -cp bin HelloWorld
 ---> Running in 72d294cf9f22
 ---> 7615c837f865
Removing intermediate container 72d294cf9f22
Successfully built 7615c837f865
```

4) Test the updated container

Run your new image:

```
$ ~/javahelloworld: docker run javahelloworld:1.1
Hello World!
```

It should work! Now, override the entrypoint to inspect the folder structure again and verify the changes.

```
docker run -it --entrypoint bash javahelloworld:1.1
```

When you run this, you will notice that it takes you straight into the working directory. You should see the `src` and `bin` folders here

```
$ ~/javahelloworld: docker run -it --entrypoint bash
javahelloworld:1.0
root@066cdc55d1ce:/home/root/javahelloworld#
root@066cdc55d1ce:/home/root/javahelloworld# ls
bin  src
root@066cdc55d1ce:/home/root/javahelloworld# exit
```

Finally, exit the container using exit.

5) View the image history

For modified base images that have been built from a Dockerfile (rather than modifying the container and then using docker commit), Docker is able to view the component layers.

Run the following command to see the different image layers than make up our javahelloworldimage:

```
$ ~/javahelloworld: docker history javahelloworld:1.1
```

8 Containers Exercise 1: Container Lifecycles

1) Run a simple container

Run the following command to pull the latest Ubuntu image and run a command

```
$ docker run ubuntu echo "Hello world!"
Hello world!
```

You should see "Hello world!" printed. Now, check your list of active containers:

```
$ docker ps
```

Notice that the container is no longer running.

2) Run an interactive terminal

Run the following command to start a container and access a terminal:

```
$ docker run -it ubuntu
```

Examine the running processes:

```
root@83cd2e37440b:/# ps
PID    TTY    TIME        CMD
1    ?    00:00:00    bash
14    ?    00:00:00    ps
```

We can see that bash is pid 1. Exit the container terminal:

```
root@83cd2e37440b:/# exit
```

You can use *'docker ps'* and observe that since we quit bash, the container has stopped.

A container will stop when the process that it is initialized with (PID 1) completes or is terminated.

3) Try adding some users to a container

Run a new container:

```
$ docker run -it ubuntu
```

To add users run the following command. Just use return to accept default values at all the prompts:

```
root@6343d96babfc:/# adduser testuser
Adding user `testuser' ...
Adding new group `testuser' (1000) ...
Adding new user `testuser' (1000) with group `testuser' ...
Creating home directory `/home/testuser' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for testuser
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
root@6343d96babfc:/#
```

Exit the container:

```
root@6343d96babfc:/# exit
exit
```

As we know, this will have stopped the container.

4) Once again run the container

```
$ docker run -it ubuntu
```

Now, try and find your user. We can do this my checking the etc/passwd file.

```
root@7e502b04f05e:/# cat /etc/passwd | grep testuser
root@7e502b04f05e:/#
root@7e502b04f05e:/# su testuser
No passwd entry for user 'testuser'
root@7e502b04f05e:/#
```

Notice that it no longer exists. This is because everytime we use the `docker run` command, we are creating and starting a new container from the ubuntu image.

If we wanted to preserve the changes, we would need to make a new image based on the Ubuntu image via either docker commit (details in Session 1 exercises), or producing a Dockerfile (details in Session 2 exercises).

9 Containers Exercise 2: Detached Mode

1) Run a container in the background using Detached Mode

When you ran the container in previous exercises, it has commandeered the input and/or output of your terminal. In Docker terms, we can say that we were 'attached' to the container.

However, you may wish to run a container as a daemon-like processes on a single host – for example, when using a container that runs a web server. To do this, we can take advantage of 'detached mode'.

Let's start a container running in detached mode. It's job will be to ping localhost 99 times.

```
$ docker run -d ubuntu ping 127.0.0.1 -c 99
c606991501576e72eb769fbff1c88d54160db4e9f9276a4f34728a8316c9ee1c
```

You will see the container ID printed after the command.

2) List active containers

```
$ docker ps
CONTAINER ID    IMAGE            COMMAND
CREATED           STATUS           PORTS    NAMES
c60699150157    ubuntu:latest    "ping 127.0.0.1" -c 9    4
seconds ago    Up 3 seconds           compassionate_bartik
```

You should see the container with the 'ping' container running.

3) Issue a command to the detached container via exec

Sometimes, we may want to execute commands against a detached container. Let's take a look at the processes running on the container that we just started:

```
$ docker exec c606 ps
UID        PID        PPID    C    STIME    TTY
TIME        CMD
root    1        0        1    00:35    ?        00:00:00    ping
127.0.0.1 -c 50
root    6        0        0    00:36    ?        00:00:00    ps -
ef
```

Notice that we only need to specify the first few characters of the container ID. This is a consistent feature when identifying containers by id with the docker API.

Also note that our PID 1 is the ping command.

4) View container log

Lets see how our ping results are progressing.

Run the following command to examine the standard output of the container – it will work on both active and stopped containers:

```
$ docker logs c60
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.154ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.110ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.095ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.212ms
...
```

5) List active containers again

```
$ docker ps
CONTAINER ID    IMAGE            COMMAND
CREATED            STATUS            PORTS    NAMES
```

The ping container has completed it's set of 99 pings and has stopped once complete. Containers run in detached mode will still terminate when their PID 1 is complete, detached mode will not prevent that.

10 Containers Exercise 3: Managing Containers

1) Find the container we added a user

We can view old, stopped containers using the -a switch on docker ps:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS
PORTS               NAMES
3b228dfccc03        ubuntu:latest       "ping 127.0.0.1 -c 5   2
minutes ago     Exited (0) About a minute
ago                     serene_euclid
0d761ff45f06        ubuntu:latest       "/bin/bash"            2
minutes ago     Exited (127) 2 minutes
ago                         insane_turing
01ba2f6dbaa2        ubuntu:latest       "/bin/bash"            2
minutes ago     Exited (0) 2 minutes
ago                         jolly_pike
f9b9ddbbb856        ubuntu:latest       "/bin/bash"            3
minutes ago     Exited (0) 3 minutes
ago                         sad_tesla
4ac7fafa6135        ubuntu:latest       "echo 'Hello World'"   3
minutes ago     Exited (0) 3 minutes
ago                         reverent_wozniak
```

Find the right container by performing a diff. Docker diff will show the changes made to the container as compared to the image it was started off.

Have a look for the telltale changes to /etc/group and /etc/passwd to help find the correct one. In my case, it was container 01ba.

```
$ docker diff 01ba
C /etc
C /etc/group
A /etc/group-
C /etc/gshadow
A /etc/gshadow-
C /etc/passwd
A /etc/passwd-
...
```

2) Start and attach to a stopped container

Let's start it again and see if we can find our user:

```
$ docker start 01ba
```

Our container is started, but we don't have our command prompt like usual – it is running detached.

Luckily, we can attach to it using docker attach – type a random character if you don't get the root@01ba... prompt after running the command.

```
$ docker attach 01ba
root@01ba2f6dbaa2:/# cat /etc/passwd | grep testuser
testuser:x:1000:1000:,,,:/home/testuser:/bin/bash
```

We have found our user! Now we can exit and stop this container again:

```
root@01ba2f6dbaa2:/# exit
```

3) Clean up our stopped containers

First, let's try and list only our stopped containers. We can use a filter on status instead of just `docker ps -a`, because the `ps -a` command will list all containers including the ones that are still running:

```
$ docker ps --filter='status=exited'
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                          ...         NAMES
3b228dfccc03        ubuntu:latest       "ping 127.0.0.1 -c 5   30
minutes ago         Exited (0) 29 minutes ago
serene_euclid
0d761ff45f06        ubuntu:latest       "/bin/bash"             31
minutes ago         Exited (127) 31 minutes ago
insane_turing
01ba2f6dbaa2        ubuntu:latest       "/bin/bash"             31
minutes ago         Exited (0) 7 minutes ago            jolly_pike
f9b9ddbbb856        ubuntu:latest       "/bin/bash"             31
minutes ago         Exited (0) 31 minutes ago            sad_tesla
4ac7fafa6135        ubuntu:latest       "echo 'Hello World'"   31
minutes ago         Exited (0) 31 minutes ago
reverent_wozniak
```

Now, lets remove one of the containers:

```
$ docker rm 3b22
3b22
```

However, it is going to take a while to remove them one by one. Let's speed things up a bit...

We can list only the ID's by adding the -q switch to docker ps. Note that 3b22 has disappeared:

```
$ docker ps --filter='status=exited' -q
0d761ff45f06
01ba2f6dbaa2
f9b9ddbbb856
4ac7fafa6135
```

Perfect! We can use the output of this command and bash array variables to quickly remove the lot:

```
$ docker rm $(docker ps --filter='status=exited' -q)
0d761ff45f06
01ba2f6dbaa2
f9b9ddbbb856
4ac7fafa6135
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS
NAMES
```

As you can see, our containers have all been removed.

11 Compose Exercise 1: Multi-container Applications Manually

We have pre-installed Compose on the VM's to help speed things up today, but please note that it is not usually bundled with Docker. If you want to install Compose on your own instance later on, instructions are available at http://bit.ly/1N4rXhf.

1) Building a sample Compose application

In your home directory, download the sample application. For this exercise, we are leveraging a sample application developed for the full-length Docker training course by ServiceRocker trainer Johnny Tu.

```
$ 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.
docker@52.26.42.249 ~: ls
docker-ca  HelloRedis  inventory-service  javahelloworld
myimage  swarmtoken  tmp
```

2) Build the image using the Dockerfile. Name your image “helloredis”

Before doing the build, feel free to inspect the Dockfile for your own curiosity. Instructors will happily answer any queries you may have about it's construction.

```
$ cd HelloRedis
$ ~/HelloRedis: cat Dockerfile
FROM java:7
COPY /src /HelloRedis/src
COPY /lib /HelloRedis/lib
WORKDIR /HelloRedis
RUN javac -cp lib/jedis-2.1.0-sources.jar -d . src/HelloRedis.java
CMD ["java", "HelloRedis"]

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

3) Start our application

First, we start a redis container, specifying the name redisdb

```
docker run -d --name redisdb redis
```

Now, run your helloredis image and link it to the redis container. The link command can be used to connect two containers.

```
 $ ~/HelloRedis: docker run --link redisdb:redisdb helloredis
Server is running: PONG
books_count = null
Server is running: PONG
books_count = null
```

Note that the books_count variable seems to be null. Don't worry, this is expected.

4) Exit our application

Press Ctrl-C to terminate the helloredis (our java application) container. Use docker stop to terminate the redis container.

```
 $ ~/HelloRedis: docker stop redisdb
redisdb
```

Note that it took several commands to start and stop our 2 container application with only one link. Consider the amount of commands we would have to run if we added just a web server and multiple application nodes.

With some applications built using the microservices architecture pattern, we can easily be looking at 10 or more application containers. Let's see how Docker Compose can make managing this tangled web a bit easier.

12 Compose Exercise 2: Multi-container Applications with Compose

1) Examine the docker-compose.yml

The Compose yml format is documented at: https://docs.docker.com/compose/yml/

```
$ ~/HelloRedis: cat docker-compose.yml
javaclient:
  build: .
  links:
    - redis:redisdb
redis:
  image: redis
```

Breaking down our small sample file:

Command	Function
javaclient:	Defines a new service called javaclient
build: .	Path to a directory containing a Dockerfile. When the value supplied is a relative path, it is interpreted as relative to the location of the yml file itself. In this case, the period will cause it to look for the dockerfile in the same path as the yml.
links:	Specifies which services to link the defined parent service to.
- redis:redisdb	Specifies the service to link to in the form *service:alias*.
redis:	Defines a new service called redis
image: redis	Specifies which image should be used for the service. Can use either image: or build: depending on whether you want to use a prebuilt image or a dockerfile.

Note the usage of the terminology 'service'. A service may consist of multiple instances when scaled, each one being one container.

2) Start the application using Compose

Start the application:

```
$ ~/HelloRedis: docker-compose up
Creating helloredis_redis_1...
Creating helloredis_javaclient_1...
Building javaclient...
....
....
Successfully built 9df932011321
....
```

Check the output on your terminal. You should see the output of the Redis server starting up, followed by the output of the java client 'PONGing'

Press Ctrl-C to terminate the application and the containers.

13 Compose Exercise 3: Managing a Compose Application

1) Start the Compose application up in detached mode

```
$ ~/HelloRedis: docker compose up -d
Recreating helloredis_redis_1...
Recreating helloredis_javaclient_1...
```

2) Check the status of each container serving the application

Let's use some of the docker client commands we covered in the Containers exercise.

Check what applications are running:

```
$ docker-compose ps
         Name                       Command             State
Ports
-------------------------------------------------------------------
------
helloredis_javaclient_1   java HelloRedis                Up
helloredis_redis_1        /entrypoint.sh redis-server    Up      63
79/tcp
```

Notice that the result only shows the containers that are defined in the docker-compose.yml found in the working directory.

Move up one directory level and run docker-compose ps to see the error output.

```
$ ~/HelloRedis: cd ..
$ ~: docker-compose ps
Can't find a suitable configuration file. Are you in the right
directory?
Supported filenames: docker-compose.yml, docker-compose.yaml, fig.
yml, fig.yaml
$ ~: cd HelloRedis/
$ ~/HelloRedis:
```

3) VIew logs from the application

Let's see how our application is going:

```
$ ~/HelloRedis: docker-compose logs
Attaching to helloredis_javaclient_1, helloredis_redis_1
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
javaclient_1 | Server is running: PONG
redis_1      | 1:C 22 Jun 08:35:19.618 # Warning: no config file
specified, using the default config. In order to specify a config
file use redis-server /path/to/redis.conf
redis_1      |                 _._
redis_1      |            _.-``__ ''-._
redis_1      |       _.-``    `.  `_.  ''-._           Redis 3.0.2
 (00000000/0) 64 bit
redis_1      |   .-`` .-```.  ```\/    _.,_ ''-._
javaclient_1 | books_count = null
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 22 Jun 08:35:19.620 # Server started, Redis
version 3.0.2
redis_1      | 1:M 22 Jun 08:35:19.620 # 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 22 Jun 08:35:19.620 # WARNING you have
Transparent Huge Pages (THP) support enabled in your kernel. This
will create latency and memory usage issues with Redis. To fix thi
s 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 22 Jun 08:35:19.620 # 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 22 Jun 08:35:19.620 * DB loaded from disk: 0.00
0 seconds
redis_1      | 1:M 22 Jun 08:35:19.620 * The server is now ready
to accept connections on port 6379
```

```
javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
...
```

Use Ctrl-C to exit.

4) Attach to the application

Let's try and attach to our application. Why do you think this doesn't work?

```
$  ~/HelloRedis: docker-compose attach
No such command: attach
Commands:
  build     Build or rebuild services
  help      Get help on a command
  kill      Kill containers
  logs      View output from containers
  port      Print the public port for a port binding
  ps        List containers
  pull      Pulls service images
  rm        Remove stopped containers
  run       Run a one-off command
  scale     Set number of containers for a service
  start     Start services
  stop      Stop services
  restart   Restart services
  up        Create and start containers
```

The docker-compose application potentially has multiple containers running. The attach command — which connects to the stdin/out of a container — would be ambiguous here, and as such, it is not supported.

5) Terminate the application

Stop the two containers serving the application with one command:

```
$ ~/HelloRedis: docker-compose stop
Stopping helloredis_javaclient_1...
Stopping helloredis_redis_1...
```