# Docker Containers

## Lab 3 – Controlling Containers

In this lab you will get a chance to manage Docker containers with a range of Docker CLI features.

### 1. Run Container as a Service

When processes are run in the background they are often referred to as daemons. You can run containers in the background
as daemons with the `docker container run -d` command. The *-d* switch stands for "detached" and runs the container
detached from your shell's input and output.

Imagine we have a service and that its job is to log data to STDOUT every 10 seconds. We can run this service in the
background with the `-d` switch. Execute the following Docker command:

```
user@ubuntu:~$ docker container run -d ubuntu /bin/sh \
-c "while true; do echo 'hello Docker'; sleep 10; done"

5d1c49639807f769b610e7e997e144624435aedeb0cf3a0acde3a2e2db56dc4b

user@ubuntu:~$
```

When the container starts, display the running containers. You should see the new container running in the background.

The `docker container logs` subcommand can display the output of a background container. Display the STDOUT log for the
container you started above.

```
user@ubuntu:~$ docker container logs 5d1c

hello Docker
hello Docker
hello Docker

user@ubuntu:~$
```

- Get help on the `docker container logs` subcommand
- Display the container log data with timestamps added
- Use the “follow” option to continuously display the log output of the container

Next let’s run an actual service within a container. Nginx is a popular web server with an available image on Docker
Hub. Run an Nginx container instance:

```
user@ubuntu:~$ docker container run -d nginx:1.11

Unable to find image 'nginx:1.11' locally
1.11: Pulling from library/nginx
6d827a3ef358: Pull complete
f8f2e0556751: Pull complete
5c9972dca3fd: Pull complete
451b9524cb06: Pull complete
Digest: sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c59582
Status: Downloaded newer image for nginx:1.11
69d580ac6e17f97271c17184598e2f15aeacafe0dd29576bf08675eaf5494e04

user@ubuntu:~$
```

Use the `docker container top` subcommand to display the processes running within the container:

```
user@ubuntu:~$ docker container top 69d5

UID          PID         PPID        C       STIME       TTY       TIME          CM
root         3887        3840        0       20:32       ?         00:00:00      ng
syslog       3908        3887        0       20:32       ?         00:00:00      ng

user@ubuntu:~$
```

Imagine you are a BSD Unix fan and prefer an "aux" style `ps` output to the default `top` output. You can have `top`
display any `ps` style you like by simply including the *ps* options after the container ID/Name. For example:

```
user@ubuntu:~$ docker container top 69d5 aux

USER      PID      %CPU      %MEM      VSZ        RSS      TTY      STAT      START     T
```

```
root     3887    0.0       0.1      31872      5260    ?        Ss       20:32    0
syslog   3908    0.0       0.0      32292      2952    ?        S        20:32    0

user@ubuntu:~$
```

- Run the top command with the `-t` ps switch
- Run the top command with the `-w` ps switch
- Try your own favorite ps switches with `docker container top`
- Learn more here https://docs.docker.com/engine/reference/commandline/top/
- And even more here `man ps` (especially when to use '-' or not)

The concept of a “container” is implemented via kernel namespaces and cgroups. Many processes can run within the
isolation/constraints of the same “container”. Notice that the previous Nginx container has no shell running within it.
This is typical of service/microservice containers. You can still launch a shell within the container to perform
diagnostics using the `docker container exec` subcommand.

Run a new interactive shell within the Nginx container:

```
user@ubuntu:~$ docker container exec -it 69d5 /bin/sh

#
```

Now list all of the processes running within the container:

```
# ps -ef

UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 04:32 ?        00:00:00 nginx: master process nginx -g da
nginx          8       1  0 04:32 ?        00:00:00 nginx: worker process
root           9       0  0 04:36 ?        00:00:00 /bin/sh
root          14       9  0 04:36 ?        00:00:00 ps -ef

#
```

In this listing we can see the two processes launched with the container and the shell we have launched within the
container along with the `ps` command running under the shell.

Also note that the `top` command run from the host shows the host relative process ids (the main nginx process id was
3887 in the above example). Within the container the main nginx process has process id 1. This is the manifestation of
the container's process namespace. Processes within a container have one id in the container and another id on the host.

The first process launched within each container is always process id 1 within the container.

Display the IP addresses within the container:

```
# ip a show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default q
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP gr
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
       valid_lft forever preferred_lft forever

#
```

Now exit the container and see if you can request the root web page from the nginx service from the host with `wget`:

```
user@ubuntu:~$ wget -qO- 172.17.0.3 | head

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }

user@ubuntu:~$
```

## 2. Creating and Copying From Containers

In this step we're going to use the BusyBox container image. BusyBox combines tiny versions of many common UNIX
utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU
fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins;
however, the options that are included provide the expected functionality and behave very much like their GNU
counterparts. BusyBox provides a fairly complete environment for any small or embedded system.

Due to its small size and fairly complete feature set, BusyBox makes for a good test container image.

Imagine you need the `uuencode` tool to encode some strings for the web. Unfortunately Ubuntu 16.04 doesn't come with
`uuencode`. To avoid installing a big Ubuntu package we could just grab a copy of the small `uuencode` tool from
BusyBox. You don't need to run a BusyBox container to copy files out of the image. We can simply create a container
based on busybox and `docker container cp` the files we need. Try it:

```
user@ubuntu:~$ docker container create -it --name file-donor busybox

Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
4b0bc1c4050b: Pull complete
Digest: sha256:817a12c32a39bbe394944ba49de563e085f1d3c5266eb8e9723256bc4448680e
Status: Downloaded newer image for busybox:latest
eda7b1019f28320e48dda6ce4f68b06e0a0630f1ffa9da3e40b68bea2701df51

user@ubuntu:~$
```

Now copy out the binary you need and use it to encode your string:

```
user@ubuntu:~$ which uuencode

user@ubuntu:~$ docker container cp file-donor:/bin/uuencode ./uuencode

user@ubuntu:~$ ls -l uuencode
```

```
-rwxr-xr-x 1 user user 1028368 Jan 12 10:10 uuencode

user@ubuntu:~$ echo "diphthong-test: How now brown cow?" | ./uuencode -

begin 664 -
C9&EP:'1H;VYG+71E<W0Z($AO=R!N;W<@8G)O=VX@8V]W/PH`
`
end

user@ubuntu:~$
```

Perfect!

## 3. Starting and Stopping Containers

We can stop and restart containers using the `docker container stop` and `docker container start` subcommands. Let's try
this by stopping and restarting the file-donor container created (but not run) above. First start the file-donor
container:

```
user@ubuntu:~$ docker container start -i file-donor

/ #
```

Explore busybox for a minute if you like (take a look in the `bin` directory), then detach from the shell with `^p ^q`.

To stop a container, use `docker container stop` and pass the command the ID of the container (you can use the entire ID
or just a prefix, as long as you supply enough characters to make the prefix unique on the local Docker host.) For
example, use `docker container ls` subcommand to display your file-donor container and then use the `docker container stop` subcommand to stop it:

```
user@ubuntu:~$ docker container ls

CONTAINER ID     IMAGE      COMMAND                  CREATED           STATUS
eda7b1019f28     busybox    "sh"                     2 minutes ago     Up 43 second
69d580ac6e17     nginx      "nginx -g 'daemon ..."   9 minutes ago     Up 9 minutes
5d1c49639807     ubuntu     "/bin/sh -c 'while..."   10 minutes ago    Up 10 minute

user@ubuntu:~$ docker container stop file-donor
```

```
  file-donor

  user@ubuntu:~$
```

Run the `docker container ls` subcommand again.

- How many containers are running?

Containers are not destroyed unless you explicitly remove them. You can delete a container with the `docker container rm` command. Run the `docker container ls` subcommand with the *-a* all switch.

```
user@ubuntu:~$ docker container ls -a
```

- How many total containers are present on your Docker host?
- How many containers are stopped on your Docker host?

Rerun the file-donor container again using the `docker container start` subcommand, something like this:

```
user@ubuntu:~$ docker container start -i file-donor

/ #
```

The `start` subcommand reruns the existing container, re-executing the container's command ( `/bin/bash` in this case).
The `-i` switch connects the input stream of you shell to the container.

## 4. Running Command Containers

In class we discussed how containers can be used as services, machines, and commands. Think back to your use of
`uuencode` earlier in the lab. Rather than copying the `uuencode` program to our host and running it, we could have just
ran it in the busybox container. In fact this would not only be easier but more reliable. After all, our host could have
had some incompatible libraries installed or missing configuration, etc.

Try running the `uuencode` program directly within the busybox container:

```
user@ubuntu:~$ docker container run busybox /bin/uuencode

BusyBox v1.26.2 (2017-06-15 18:09:33 UTC) multi-call binary.

Usage: uuencode [-m] [FILE] STORED_FILENAME

Uuencode FILE (or stdin) to stdout

    -m  Use base64 encoding per RFC1521

user@ubuntu:~$
```

Ok, so far so good. We now know we can run any program in the container filesystem within the container. The help tells
us that the `uuencode` program will accept input from stdin. Most Unix programs allow you to send data to them via STDIN
either by default or by passing them the dash ("-") argument.

So if we echo our encoding string to the `uuencode -` we should get our encoded string. The only catch is that we will
not be running the `uuencode` program directly, we will be running Docker. We need to tell Docker to connect the STDIN
of `uuencode` to our shell. This is easy enough, we have been using the `-i` switch for this. However, in this case we
do not want an interactive tty session (-t), we simply want to connect our output to the `uuencode` program in the
container, letting its output flow back to our terminal.

Try it:

```
user@ubuntu:~$ echo "diphthong-test: How now brown cow?" \
| docker container run -i busybox /bin/uuencode -

begin 644 -
C9&EP:'1H;VYG+71E<W0Z($AO=R!N;W<@8G)O=VX@8V]W/PH`
`
end

user@ubuntu:~$
```

Excellent. Now we have seen containers in the role of machines (our Ubuntu containers), in the role of services (our
nginx container) and commands (our uuencode busybox).

## 5. Stats

Run a new nginx container as a daemon:

```
user@ubuntu:~$ docker container run -d --name=web87 --label="svc=web" nginx:1.11

9ec466e5aa7ed85f67d6888698b4b153ad7b0233510cb66cf6497be6b1818747

user@ubuntu:~$
```

Run the `docker container stats` subcommand.

- How many containers are running?
- How many process IDs are present in each container?
- Other than the container ID, what type of data is stats showing you?

Type `control C` to exit stats.

Rerun `docker container stats` with the *web87* argument:

```
user@ubuntu:~$ docker container stats web87
```

- What is displayed?

Stop stats and rerun stats with the `-a` switch:

```
user@ubuntu:~$ docker container stats -a
```

- What is displayed?

Exit stats.

## 6. Cleanup

After you have finished exploring, stop, and remove all of the containers you started/created in this lab. Remember that
you can refer to containers by name or ID.

Congratulations you have completed the controlling containers lab!