First Steps with Docker

You should either have access to VM with Docker installed, or use http://play-with-docker.com for this exercise.

To begin with, type:

```
$ docker run debian echo hello world
```

(Don't type the $, it's intended to represent your command prompt).

You should get output similar to:

```
Unable to find image 'debian:latest' locally
latest: Pulling from library/debian
693502eb7dfb: Pull complete
Digest: sha256:52af198afd8c264f1035206ca66a5c48e602afb32dc912eb...
Status: Downloaded newer image for debian:latest
hello world
```

Run the exact same command again and the output should be a lot cleaner and faster:

```
$ docker run debian echo hello world
hello world
```

So what's happened here? Docker has spun up a new container based on the image we specified (`debian`) and asked it to execute the command `echo hello world`. Once the container had executed the command, Docker stopped the container. A Docker container will always stop when the main process (PID 1) completes, even if other processes are still running.

The first time around, Docker didn't have a copy of the `debian` image, so it had to download a copy.

It's worth considering for a moment how much longer this would have taken if we were using traditional VMs - spinning up a new VM would likely have taken a minute or two and downloading a new image could easily have taken much longer. We'll look at how these speed benefits are realised later.

For now, let's look at some more useful examples. Try running:

```
$ docker run -it debian bash
```

This time we've specified the flags `-i` and `-t` which will give us an interactive terminal inside the container. This should look something like:

```
root@14efb4f27237:/#
```

Try running simple commands such as `ls`, `uname`, `ps`, `hostname` and `touch`.

Note that changes you make in a container do not affect the host system, in the same way as a VM. Try:

```
root@14efb4f27237:/# echo "in a container" > /testfile
root@14efb4f27237:/# cat /testfile
in a container
```

Now let's leave the container. The easiest way is just to type exit:

```
root@14efb4f27237:/# exit
```

Note that the file we created does not exist on the host:

```
$ cat /testfile
cat: can't open '/testfile': No such file or directory
```

Try starting a new container and looking for the file:

```
$ docker run -it debian bash
root@68cb1624d4cb:/# cat /testfile
cat: /testfile: No such file or directory
```

Again, it's not there — each container gets it own filesystem that is completely separate from other containers and the host. Later on, we'll see how we can share files between containers and the host.

Let's have a look at something a bit more useful. Exit from the last container and try running the following command:

```
root@68cb1624d4cb:/# exit
$ docker run --name web -d -p 8000:80 nginx
...
```

This time we've used the `nginx` image — a webserver — and the `-d` flag to start the container in the background rather than taking over our terminal. We've also taken the opportunity to give the container a name — "web". We can see information on all the running containers on our host by using the `docker ps` or `docker container ls` commands (they are synonymns for each other):

```
$ docker container ls
CONTAINER ID  IMAGE  COMMAND                  CREATED...
a31dae86c263  nginx  "nginx -g 'daemon ..."   2 hours ago
```

You can also see stopped containers by providing the `-a` flag:

```
$ docker container ls -a
CONTAINER ID  IMAGE   COMMAND                  CREATED...
a31dae86c263  nginx   "nginx -g 'daemon ..."   2 hours ago
68cb1624d4cb  debian  "bash"                   2 hours ago
5e7818911284  debian  "bash"                   3 hours ago
```

Now try running:

```
$ curl localhost:8000
<!DOCTYPE html>
<html>
```

```
...
```

If you know the IP address of your Docker host, you should also be able to visit this page in a webbrowser (if you're using Play With Docker there should be a link at the top of the page with the number 8000).

This example shows how we can run a simple application in Docker - in this case a webserver - and have it serve traffic to clients by exposing ports using the `-p` flag. Our example exposed port 80 inside the container, which Nginx is listening on, to port 8000 on the host.

We can see what's happening in the container with `docker logs`:

```
$ docker logs web
172.17.0.1 - - [06/Mar/2017:13:40:03 +0000] "GET / HTTP/1.1" 200...
```

If you need to get an interactive terminal inside a container running in the background (for example for debugging purposes), use the `docker exec` command:

```
$ docker exec -it web bash
root@9a141d0a2663:/# ps aux
USER   PID %CPU %MEM    VSZ   RSS TTY  STAT START   TIME COMMAND
root     1  0.0  0.0  31760  4948 ?    Ss   12:19   0:00 nginx: m
nginx    7  0.0  0.0  32152  2964 ?    S    12:19   0:00 nginx: w
root     8  0.4  0.0  20244  3184 ?    Ss   12:21   0:00 bash
root    15  0.0  0.0  17500  2016 ?    R+   12:21   0:00 ps aux
root@9a141d0a2663:/# exit
```

Finally, we can stop the running container with the `stop` command:

```
$ docker stop web
```

And we can remove containers with the `rm` command:

```
$ docker rm web
```

Bonus Task

See if you can get a Redis database working in a container (if you haven't heard of it, Redis is a simple in-memory key-value store). Bonus points for successfully managing to use the `redis-cli` within the container to insert/return data.

Building an Image with Commit

In this exercise, we'll have a look at how we can build our own image and use it to create containers. We can see which images are available to the daemon with the `docker images` command:

```
$ docker images
REPOSITORY  TAG     IMAGE ID      CREATED     SIZE
nginx       latest  6b914bbcb89e  7 days ago  182 MB
debian      latest  978d85d02b87  8 days ago  123 MB
```

Alternatively we could have used `docker image ls`. New images are retrieved implicitly by `docker run` if they are not available locally. We can also pull images explicitly with the `docker pull` command:

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
627beaf3eaaf: Pull complete
Digest: sha256:58e1a1bb75db1b5a24a462dd5e2915277ea06438c3f1051...
Status: Downloaded newer image for alpine:latest
```

For the rest of this exercise, we'll build an image with the famous "cowsay" program:

Start by launching a new `debian` container:

```
$ docker run -it --name cowsay debian bash
```

Now update the packager manager inside the container and install the cowsay and fortune utilities:

```
root@49a02c8575e6:/# apt-get update
...
root@49a02c8575e6:/# apt-get install -y cowsay fortune
...
```

And try it out:

```
root@49a02c8575e6:/# /usr/games/cowsay $(/usr/games/fortune)
...
```

OK, so now we have a container with our software installed. The next step is to turn it into an image. In this example, we'll take the easy route of using the docker commit command. Exit from the container and run the commit command with the container name and the name we want to give our image:

```
root@49a02c8575e6:/# exit
exit
$ docker commit cowsay cowmage
sha256:807f3c4c1eb0da3975a1f61642a3131bc4ec752ad644b5545509334...
```

Docker responds by giving us a content based hash that can be used to reference our image. It doesn't matter that the container was stopped when we ran the commit command - the filesystem still exists on disk until the container is removed.

We can now run our image and execute our application directly:

```
$ docker run cowmage /usr/games/cowsay Moo
...
```

Building an Image with a Dockerfile

Let's create our `cowsay` image the proper way. Start by creating a new directory for our code and our empty `Dockerfile`:

```
$ mkdir cowsay
$ cd cowsay
$ touch Dockerfile
```

Put the following contents into the `Dockerfile` using whichever editor you're most comfortable with (try nano if you don't have much experience with vi or emacs):

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y cowsay fortune
```

The `FROM` statement defines the *base image* for the new image we are creating. In this case we are using the `debian` image as before, although we've been a little more specific and chosen the "jessie" version.

The `RUN` statements define the "steps" in creating our new image.

Try building the image from the same directory as the Dockerfile:

```
$ ls
Dockerfile
$ docker build -t cowmage2 .
...
```

Note that the "." at the end is important (it specifies the "context" of the build to be the current directory). We used the -t flag to specify a name (`cowmage2`) for the image.

We can test out the new image as before:

```
$ docker run cowmage2 /usr/games/cowsay boo
...
```

So, this has achieved the same effect as `docker commit`, but in a repeatable manner which is easy to build on. The next thing we can do is set a `CMD` statement that calls cowsay automatically when a container is started. Edit the `Dockerfile` so it contains the following:

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y cowsay fortune
CMD /usr/games/cowsay $(/usr/games/fortune)
```

Repeat the build:

```
$ docker build -t cowmage2 .
...
```

And this time try running with no command specified:

```
$ docker run cowmage2
...
```

Nice, this is a bit simpler. But we can actually do a bit better - currently if we want to specify new text we have to overwrite the whole command rather than just provide the new text. We can use the `ENTRYPOINT` statement to achieve this.

Update the Dockerfile to:

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y cowsay fortune
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Moo", "to", "you"]
```

Build it and run it:

```
$ docker build -t cowmage2 .
...
$ docker run cowmage2
...
$ docker run cowmage2 "boo"
...
```

Effectively `ENTRYPOINT` specifies the "command" to run and `CMD` specifies any arguments to the `ENTRYPOINT`. `CMD` is directly analogous to the command specified at the end of `docker run` CLI calls and can be overridden by the CLI as shown above. The `ENTRYPOINT` command can also be overridden by passing the `--entrypoint` flag to `docker run`.

Note that this time we've used the exec syntax (the square brackets) to execute the `cowsay` binary directly, rather than calling through a shell (which is the default for all `RUN`, `ENTRYPOINT` and `CMD` commands). A side-effect of this is that we are no longer able to use the fortune program as a default input. In order to do something like that, we have to get a bit more organized and write a small script to help us out. Let's try that now. Start by creating a file in the cowsay directory called `entrypoint.sh` with the following contents:

```
#!/bin/bash
if [ $# -eq 0 ]; then
  /usr/games/cowsay $(/usr/games/fortune)
else
  /usr/games/cowsay "$@"
fi
```

Make the file executable:

```
$ ls
Dockerfile  entrypoint.sh
$ chmod +x entrypoint.sh
```

Now update the `Dockerfile` so that it looks like:

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y cowsay fortune

COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]
```

The `COPY` command will copy our script from the directory into the Docker image, and the new `ENTRYPOINT` command will execute it by default. Build and run it once more:

```
$ docker build -t cowmage2 .
...
$ docker run cowmage2
...
$ docker run cowmage2 moo
...
```

Great, this is about as far as we want to go with the venerable cowsay application…

Bonus Task

Cowsay can use different graphics rather than just the cow - see if you can modify the `Dockerfile` to use a different animal by default.

Docker Volumes

Docker volumes can be used in a lot of different ways. First let's look at using the `-v` flag to `docker run`.

Try the following:

```
$ echo "this is a file on the host" > /tmp/hostfile
$ docker run -v /tmp/hostfile:/containerfile \
         debian cat /containerfile
this is a file on the host
```

What's happened here is we've mounted the file `/tmp/hostfile` on the host as the file `/containerfile` inside the container. We can do the same with directories and binaries, so you can make anything on the host accessible to the container (but be careful with file permissions and dynamic libraries). The path must always be fully qualified, so you will often see files and directories in local directories prefixed with `$PWD` or `$(pwd)`.

Let's try adding a directory. Docker will create it for us if it doesn't exist:

```
$ docker run -v /tmp/hostdir:/condir debian \
    sh -c 'echo "hello from containerland" > /condir/containerfile'
$ cat /tmp/hostdir/containerfile
hello from containerland
```

So we can create files in the container and they will appear immediately on the host. This is because it is in fact exactly the same file - Docker is not copying or adding indirection here - it is simply exposing the file directly.

Rather than specify a directory or file on the host, we can let Docker manage the volume itself, under a directory it controls. For example:

```
$ docker run -v /data debian \
  sh -c 'echo "hello from container $HOSTNAME" > /data/file'
```

Now if we use the `volume` subcommand, we can see our new volume:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               1ec7b9124f932f36931de615e2985355c126112adbf...
```

Docker has assigned a unique, but rather large and unwieldy name to our volume. We can find more information about the volume with inspect:

```
$ docker volume inspect \
  1ec7b9124f932f36931de615e2985355c126112adbf8c66b78e6440d6ab5f202
[
    {
        "Driver": "local",
        "Labels": null,
```

```
        "Mountpoint": "/graph/volumes/1ec7b9124f932f36931de615...
        "Name": "1ec7b9124f932f36931de615e2985355c126112adbf8c...
        "Options": {},
        "Scope": "local"
    }
]
```

Amongst other things, this tells us the mountpoint, or where Docker has placed our volume. If you have root access to the Docker host, you can directly access the path (note this won't work if you are using Docker for Mac or Windows):

```
$ cat /graph/volumes/1ec7b9124f932f36931de615e29853.../\_data/file
hello from container 7ceb66d53323
```

Of course, you wouldn't normally access files like this - Docker managed volumes are typically only accessed via containers.

Try also creating a volume with the `docker volume` command:

```
$ docker volume create my_vol
my_vol
```

You can run `docker volume ls` to verify this worked. To attach it to a container, use the `-v` flag with the name of the volume and the mountpoint in the container:

```
$ docker run -v my_vol:/data debian \
  sh -c 'echo "hi from $HOSTNAME" > /data/file'
```

Now we can read it from a different container:

```
$ docker run -v my_vol:/data debian cat /data/file
hi from 6470a9d242d6
```

The other option is declare a `VOLUME` in a `Dockerfile`. For example, the `Dockerfile` for the Redis official image includes the line:

```
VOLUME /data
```

When a container is created, Docker will create a new volume for each `VOLUME` statement. In this case, it means any data written to disk by the Redis container will be persisted and not lost when the container is removed. Volumes are only deleted when the `-v` flag is passed to `docker rm` or with the `docker volume rm command`. Volumes will never be deleted if they are in use by a container.

To finish this section, let's see how you might typically interact with a volume during development. We'll start with an Nginx container:

```
$ docker run -d -p 8000:80 --name web nginx
...
$ curl localhost:8000
<!DOCTYPE html>
<html>
<head>
```

```
<title>Welcome to nginx!</title>
...
```

Now, let's try to serve our own HTML page. Create a directory called `html` and put a simple HTML file called `index.html` inside with the following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test title</title>
  </head>
  <body>
    Test body
  </body>
</html>
```

The `nginx` image is configured to look for files in `/usr/share/nginx/html`, so we mount our directory to that path:

```
$ docker stop web
...
$ docker rm web
...
$ ls html
html        index.html
$ docker run -v $PWD/html:/usr/share/nginx/html \
        -d -p 8000:80 --name web nginx
```

And try curling the file again. You should see your new HTML page!

Now try editing the `index.html` file to read:

```
...
  <body>
     Updated body
  </body>
...
```

If you now access the webserver again, you should find your changes appear immediately.

In this manner, we can iteratively and quickly develop software and assets. Once you're happy with what you've built, you can then bake the results into a new image.

Try creating a new `Dockerfile` with the contents:

```
FROM nginx:1.11

COPY html /usr/share/nginx/html
```

Build it:

```
$ docker build -t myweb .
...
```

And now we can run it again without the mount (we need to stop the previous container first)

```
$ docker stop web
web
$ docker run -d -p 8000:80 --name web2 myweb
9317adbef8ab5e6b69fe98e98ba8fd22638dbb6ae4fb8c19306cc4deac65d45a
$ curl localhost:8000
...
```

Note that you can still mount a volume on top of this image and continue developing, then re-run the build command to create the final image.

Basic Docker Networking

In this worksheet we will see the basics of how containers on the same host can communicate.

Start by creating a new Docker bridge network:

```
$ docker network create -d bridge mynet
e5788917f54b6c0ed6d9b9811511051c96b64fe8c66d400c3baf89a2d237831e
```

We can see what networks are availabe with the `network ls` command:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
636450ff0145        bridge              bridge              local
ff885138f910        host                host                local
e5788917f54b        mynet               bridge              local
21bfb77d0e38        none                null                local
```

Start a Redis container and attach it to the network with the `--net` flag:

```
$ docker run -d --name redis --net mynet redis
28da10ac27d1f13144365369adc0ef6bb6bb283b9e50781fee16f8259ac9499e
```

Note that we don't need to use the `-p` command to expose ports as we won't be connecting from the host.

Now let's start another container and try to connect to our redis server:

```
$ docker run --net mynet debian ping -c 4 redis
PING redis (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: icmp_seq=0 ttl=64 time=0.593 ms
64 bytes from 172.19.0.2: icmp_seq=1 ttl=64 time=0.075 ms
64 bytes from 172.19.0.2: icmp_seq=2 ttl=64 time=0.089 ms
64 bytes from 172.19.0.2: icmp_seq=3 ttl=64 time=0.071 ms
--- redis ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.071/0.207/0.593/0.223 ms
```

Great. We can see that our new container was able to reach the redis container by name, which was resolved to an internal IP address.

Now let's connect to the actual database using the `redis-cli` in another container:

```
$ docker run --net mynet -it redis redis-cli -h redis
redis:6379>
```

Here we've started an interactive connection to the remote redis container. The argument `-h redis` told the client to connect to the remote server called `redis`.

Now we can run some Redis commands:

```
redis:6379> ping
PONG
redis:6379> set foo bar
OK
redis:6379> get foo
"bar"
redis:6379> exit
```

This is the basic way to connect containers in Docker. Things can get a lot more complex when we consider multi-host networking and load-balancing, but the principle of connecting via known names on shared networks remains the same.

Docker Compose

In this exercise we'll briefly explore the Docker Compose tool.

Create a file `docker-compose.yml` with the following contents:

```
version: '2'
services:
  identidock:
    image: amouat/identidock
    ports:
      - "5000:5000"
    environment:
      ENV: DEV

  dnmonster:
    image: amouat/dnmonster:1.0

  redis:
    image: redis:3.0
```

This file defines three containers, identidock, dnmonster and redis. They all have associated images and the identidock container also exposes a port to the host and sets an environment variable.

Try starting the service:

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

You should now be able to access the application at http://localhost:5000

Try some of the other Compose commands that you can list with `docker-copmose --help` e.g. `logs` and `ps`.