Basic Docker commands

Here is a list of ten basic and easiest Docker commands. This will help you in getting started with Docker

- List all running containers

```
$ sudo docker ps
```

- List all running and stopped containers

```
$ sudo docker ps -a
```

- List all Docker images

```
$ sudo docker images
```

- Pull a Docker image

```
$ sudo docker pull alpine
```

- Run a Docker container

```
$ sudo docker run -i -t alpine /bin/sh
```

- Check logs of a Docker container

```
$ sudo docker logs <container_id>
```

- Run a command in a running Docker container

```
$ sudo docker exec -i -t <container_id> /bin/sh
```

- Stop a Docker container

```
$ sudo docker stop <container_id>
```

- Delete a Docker container

```
$ sudo docker rm <container_id>
```

- Delete a Docker image

```
$ sudo docker rmi <image_id>
```

Bonus command

```
$ sudo docker --help
```

Docker Images

Docker images are the read-only templates from which a container is created. In previous sections, we have seen how to pull and list the docker images. Let us learn how to build them.

Building Docker images by using docker commit

A lot of Docker's command line subcommands are inspired by Git, including this one. Docker provides a very simple way to create images. The workflow goes like this:

- Create a container from a Docker image
- Make required changes, like installing a web server or adding a user
- On command line execute the following

```
$ sudo docker commit <container_id> <optional_tag>
```

While this presents a very simple way to create images, this is not a good way to do so. Creating images this way is not very reproducible and hence not recommended.

Building Docker images by using a Dockerfile

Docker provides an easy way to build images from a description file. This file is known as Dockerfile. A simple Dockerfile will look like this:

```
FROM centos
RUN yum -y update && yum clean all
RUN yum -y install httpd
EXPOSE 80
```

```
CMD ["/usr/sbin/httpd", "-D", "FOREGROUND"]
```

Dockerfile supports various commands. I have used a few of them and I am going to describe them below:

- FROM: This defines the base image for the image that would be created
- MAINTAINER: This defines the name and contact of the person that has created the image.
- RUN: This will run the command inside the container.
- EXPOSE: This informs that the specified port is to be bound inside the container.
- CMD: Command to be executed when the container starts.

A comprehensive list and description of all the supported commands can be found in the documentation. Let us create a file with the above lines and build an image from this.

```
$ sudo docker build -f <path_of_dockerfile> .
```

Images created this way document the steps involved clearly and hence using Dockerfile is a good way to build reproducible images. It is also worth nothing that each line in Dockerfile create a new layer in Docker image. So often, we will club statements using and operator (&&) like this:

```
RUN yum -y update && yum clean all
```

Docker Registry

Docker registry is a repository of Docker images. It enables users to share images publicly and privately. Docker registry can be hosted on-premise, and can be password protected which makes it a great tool for any corporate environment where security and privacy are important.

How to setup Docker Registry?

Docker has containerize the registry which makes the installation extremely easy. At the moment, registry:2 is the latest registry. So let us pull the image first

```
$ sudo docker pull registry:2
```

As and when we push images to the registry, the registry will store the data. We want to ensure that our data is safe, even if the container running registry dies. The easiest way to achieve that

is to mount a directory from the host to the container which will store the data. So let us create a directory.

```
$ sudo mkdir /opt/registry-data
```

Now will mount this directory inside the registry container.

```
$ sudo docker run -d -p 5000:5000 -v /opt/registry-data:/var/lib/registry registry:2
```

Let us verify that the container is actually running

```
$ sudo docker ps
```

Once we confirm that the container is running, we should tag an image and try to push it.

```
$ sudo docker tag <image_id> localhost:5000/myimage
$ sudo docker push localhost:5000/myimage
```

Docker Networking

Connecting containers is very important for most of the applications out there. A classic web application consists of at least a web server, an application server and a database server. The web server needs to talk to the application server and the application server would need to talk to the database server. A legacy, but popular way to do so is via passing `--link` flag to `docker run` command.

```
$ sudo docker run -d --name mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw mysql
$ sudo docker run --link mysql -i -t centos /bin/bash
```

Currently, Docker has a very elaborated network suite to cater to a wide variety of use-cases. By default it comes with three network pre-configured:

- **none:** Containers in this network have only local interface. They have no external connectivity. So this is ideal for applications that do not need network but may contain sensitive info
- **host:** This network copies the host network. So the containers running in this network basically have an identical network as the host.
- **bridge:** This the default network in which Docker will boot up containers. This creates a bridge (docker0 interface) with a gateway and other required network components.

Containers running in this network can talk to the other containers in the same network but they cannot talk to any other containers in any other network.

Please take moment to run a container in all these network and run ifconfig or ip addr.

```
$ sudo docker run --net=<type_of_the_network> -i -t centos /bin/bash
```

Docker does not limit us to these three networks only. We can create additional networks as and when required. This helps in maintaining isolation between networks and improves security. Docker lets us create two kinds of user-defined networks:

- **Bridge:** It is similar to default bridge network with minor differences, like --link flag is not available for user-defined networks. Since bridge networks are defined per host, all the containers in this network must reside on the same host. Multiple networks on the same host are isolated from each other, however, a container can be a part of multiple networks and facilitate inter-network communication. Let us create and run a container in a user-defined bridge network.

```
$ sudo docker network create --driver bridge isolated_nw
$ sudo docker run --net=isolated_nw -i -t centos /bin/bash
```

  Try pinging containers running in other networks.

- **Overlay:** Overlay network is Docker's way to simplify multi-host networking. An overlay network can span to several hosts and maintain network level isolation. It uses libnetwork and libkv. To run libkv, we need to install one of the key value stores supported by Docker which are Consul, Etcd, and ZooKeeper. For this session, we will go ahead with etcd. So let us start by installing etcd on all the servers that are going to run the Docker daemon.

```
$ sudo yum -y install etcd
```

  Designate one of the servers as etcd master and configure it to listen to the non-local IP address.

```
ETCD_LISTEN_CLIENT_URLS="http://<ip-addr>:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://<ip-addr>:2379"
```

  Now restart the etcd service.

```
$ sudo systemctl restart etcd
```

Once our etcd service is up, we can start the Docker daemon with some additional parameters.

```
$ nohup sudo docker daemon  --cluster-store=etcd://<ip-addr>:2379 --cluster-advertise=eth0:2376 &
```

Now let us create an overlay network, named overlay-net1 on one of the machines.

```
$ sudo docker network create --driver overlay overlay-net1
```

And then let us check this on the other machine.

```
$ sudo docker network ls
```

This is it! Now any container in the overlay network will be able to communicate to each other regardless of what host it is running on.

Docker Remote Api

Docker provides a rich set of APIs which can be used to do manage containers. This opens up a huge opportunity of creating automation to improve user experience and scalability. For this lab, we will bind the Docker daemon to a TCP port.

```
$ sudo docker daemon -H :1234
```

Let us try doing some basic operations using the Remote API.

- Listing the running containers

```
$ curl http://localhost:1234/containers/json
```

- Pulling a Docker image

```
$ curl -d '' http://localhost:1234/images/create?fromImage=alpine
```

- Creating a container

```
$ curl -H "Content-Type: application/json" -d '{"Image":"alpine", "Cmd":["sleep", "10000"]}'
http://localhost:1234/containers/create
```

- Running a container

```
$ curl -d '' http://localhost:1234/containers/<container_id>/start
```