Docker for Beginners - Linux

Task 0: Prerequisites

Clone the Lab's GitHub Repo

```
git clone https://github.com/dockersamples/linux_tweet_app
```

Task 1: Run some simple Docker containers

Run a single task in an Alpine Linux container

Start a new container and tell it to run the **hostname** command. The container will start, execute will start, execute the **hostname** command, then exit.

```
docker container run alpine hostname
```

Docker keeps a container running as long as the process it started inside the container is still running. In this case the **hostname** process exits as soon as the output is written. This means the container stops. However, Docker doesn't delete resources by default, so the container still exists in the **Exited** state.

List all containers:

```
docker container ls --all
```

Containers which do one task and then exit can be very useful. You could build a Docker image that executes a script to configure something. Anyone can execute that task just bby running the container -they don't need the actual scripts or configuration information.

Run an interactive Ubuntu container

Run a Docker container and access its shell

```
docker container run --interactive --tty --rm ubuntu bash

--interactive: says you want an interactive session
--tty: allocates a pseudo-tty
--rm: tells Docker to go ahead and remove the container when it's d
```

Run the following commands in the container:

```
ls /
ps aux
cat /etc//issue
```

Type **exit** to leave the shell session. This will terminate the **bash** process, causing the container to exit.

```
exit

As we used the --rm flag when we started the container, Docker remo
```

For fun, let's check the version of our host VM:

```
cat /etc/issue
```

The distribution of Linux inside the container does not need to match the distribution of Linux running on the Docker host.

Linux containers require the Docker host to be running a Linux kernel.

Run a background MySQL container

Background containers are how you'll run most applications. Here's a simple example using MySQL.

Run a new MySQL container with the following command:

```
docker container run \
--detach \
--name mydb \
-e MYSQL_ROOT_RASSWORD=my-secrect-pw \
mysql:latest

--detach will run the container in the background
--name will name it mydb
--e will use an environment variable to specify the root password
```

List running containers:

```
docker container ls
```

Check what's happening in your containers by using a couple of build-in Docker commands: **docker container logs** and **docker container top**:

```
docker container logs mydb
```

Look at the processes runing inside the container:

```
docker container top mydb
```

List the MySQL version using **docker container exec**

docker container exec allows you to run a command inside a container.

```
docker exec -it mydb \
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

You can also use **docker container exec** to connect to a new shell process inside an already-running container. Executing the command below will give you an interactive shell inside your MySQL container.

```
docker exec -it mydbb sh
```

Check the version number by running the same command:

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

Exit the interactive shell

```
exit
```

Task 2: Package and run a custom app

using Docker

To package your own apps as Docker images.

Build a simple website image

Make sure you're in the **linux*tweet*app** directory.

```
cd ~/linux_tweet_app
```

Display the content of the Dockerfile.

```
cat Dockerfile
```

In order to make the following commands more copy/paste friendly, export an environment variable containing your DockerID

```
export DOCKERID=<your docker id>
```

Echo the value of the variable back to the terminal to ensure it was stored correctly

```
echo $DOCKERID
```

Use the **docker image build** command to create a new Docker image using the instructions in the Dockerfile.

```
docker image build --tag $DOCKERID/linux_tweet_app:1.0 .
```

Use the **docker container run** command to start a new container from the image you created:

```
docker container run \
--detach \
--publish 80:80 \
--name linux_tweet_app \
$DOCKERID/linux_tweet_app:1.0
```

Once you've accessed your website, shut it down and remove it.

```
docker container rm --force linux_tweet_app
```

Task 3: Modify a running website

Start our web app with a bind mount

Start the web app and mount the current directory into the container

```
docker container run \
--detach \
--publish 80:80 \
--name linux_tweet_app \
--mount type=bind, source="$(pwd)", target=/usr/share/nginx/html \
$DOCKERID/linux_tweet_app:1.0
```

Modify the running website

Copy a new **index.html** into the container

```
cp index-new.html index.html
```

Go the running webstei and refresh the page. Notice that the site has changed.

Even though we've modeifed the **index.html** local filesystem and seen it reflected in the running container, we've not actually changed the Docker image that the container was started from.

Stop and remove the currently running container.

```
docker rm --force linux_tweet_app
```

Rerun the current version without a bind mount

```
docker container run \
--detach \
--publish 80:80 \
--name linux_tweet_app \
$DOCKERID/linux_tweet_app:1.0
```

Notice the website is back to the original version

Stop and remove the current container

```
docker rm --force linux_tweet_app
```

Update the image

To persisit the changes you made to the **index.html** file into the image, you need to build a new version of the image

Build a new image and tag it as **2.0**

```
docker image build --tag $DOCKERID/linux_tweet_app:2.0 .
```

Look at the images on the system

```
docker image ls
```

Test the new version

Run a new container from the new version of the image

```
docker container run \
--detach \
--publish 80:80 \
--name linux_tweet_app \
$DOCKERID/linux_tweet_app:2.0
```

Check the new version of the website

Run another new container, this time from the old version of the image.

```
docker container run \
--detach \
--publish 8080:80 \
--name old_linux_tweet_app \
$DOCKERID/linux_tweet_app:1.0
```

View the old version of the website

Push your images to Docker Hub

List the images on your Docker host

```
docker image ls -f reference="$DOCERID/*"
```

Before you can push your images, you will need to log into Docker Hub:

```
docker login
```

Push version **1.0** of you web app using **docker image push**

```
docker image push $DOCKERID/linux_tweet_app:1.0
```

Now push version **2.0**

```
docker image push $DOCKERID/linux_tweet_app:2.0
```