Docker for Beginners - Linux

> **Tasks**:
>
> - Task 0: Prerequisites
> - Task 1: Run some simple Docker containers
> - Task 2: Package and run a custom app using Docker
> - Task 3: Modify a Running Website

Task 0: Prerequisites

You will need all of the following to complete this lab:

- A clone of the lab's GitHub repo.
- A DockerID.

Clone the Lab's GitHub Repo

Use the following command to clone the lab's repo from GitHub (you can click the command or manually type it). This will make a copy of the lab's repo in a new sub-directory called `linux_tweet_app`.

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

Make sure you have a DockerID

If you do not have a DockerID (a free login used to access Docker Cloud, Docker Store, and Docker Hub), please visit Docker Cloud and register for one. You will need this for later steps.

Task 1: Run some simple Docker containers

There are different ways to use containers. These include:

1. **To run a single task:** This could be a shell script or a custom app.
2. **Interactively:** This connects you to the container similar to the way you SSH into a remote server.
3. **In the background:** For long-running services like websites and databases.

In this section you'll try each of those options and see how Docker manages the workload.

Run a single task in an Alpine Linux container

In this step we're going to start a new container and tell it to run the `hostname` command. The container will start, execute the `hostname` command, then exit.

1. Run the following command in your Linux console.

   ```
   docker container run alpine hostname
   ```

   The output below shows that the `alpine:latest` image could not be found locally. When this happens, Docker automatically *pulls* it from Docker Hub.

   After the image is pulled, the container's hostname is displayed (`888e89a3b36b` in the example below).

   ```
   Unable to find image 'alpine:latest' locally
   latest: Pulling from library/alpine
   88286f41530e: Pull complete
   Digest: sha256:f006ecbb824d87947d0b51ab8488634bf69fe4094959d935c0c103f4820a41
   Status: Downloaded newer image for alpine:latest
   888e89a3b36b
   ```

2. 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
   ```

   Notice that your Alpine Linux container is in the `Exited` state.

   ```
   CONTAINER ID        IMAGE               COMMAND             CREATED
   888e89a3b36b        alpine              "hostname"          50 seconds ago
   ```

   > **Note:** The container ID *is* the hostname that the container displayed. In the example above it's

> `888e89a3b36b`.

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 by running the container - they don't need the actual scripts or configuration information.

Run an interactive Ubuntu container

You can run a container based on a different version of Linux than is running on your Docker host.

In the next example, we are going to run an Ubuntu Linux container on top of an Alpine Linux Docker host (Play With Docker uses Alpine Linux for its nodes).

1. Run a Docker container and access its shell.

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

   In this example, we're giving Docker three parameters:

   - `--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 done executing.

   The first two parameters allow you to interact with the Docker container.

   We're also telling the container to run `bash` as its main process (PID 1).

   When the container starts you'll drop into the bash shell with the default prompt `root@<container id>:/#`. Docker has attached to the shell in the container, relaying input and output between your local session and the shell session in the container.

2. Run the following commands in the container.

   `ls /` will list the contents of the root director in the container, `ps aux` will show running processes in the container, `cat /etc/issue` will show which Linux distro the container is running, in this case Ubuntu 16.04.4 LTS.

   ```
   ls /
   ```

   ```
   ps aux
   ```

```
cat /etc/issue
```

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

   ```
   exit
   ```

   > **Note:** As we used the `--rm` flag when we started the container, Docker removed the container when it stopped. This means if you run another `docker container ls --all` you won't see the Ubuntu container.

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

   ```
   cat /etc/issue
   ```

   You should see:

   ```
   Welcome to Alpine Linux 3.6
   Kernel \r on an \m (\l)
   ```

Notice that our host VM is running Alpine Linux, yet we were able to run an Ubuntu container. As previously mentioned, the distribution of Linux inside the container does not need to match the distribution of Linux running on the Docker host.

However, Linux containers require the Docker host to be running a Linux kernel. For example, Linux containers cannot run directly on Windows Docker hosts. The same is true of Windows containers - they need to run on a Docker host with a Windows kernel.

Interactive containers are useful when you are putting together your own image. You can run a container and verify all the steps you need to deploy your app, and capture them in a Dockerfile.

> You *can* commit a container to make an image from it - but you should avoid that wherever possible. It's much better to use a repeatable Dockerfile to build your image. You'll see that shortly.

Run a background MySQL container

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

1. Run a new MySQL container with the following command.

```
docker container run \
--detach \
--name mydb \
-e MYSQL_ROOT_PASSWORD=my-secret-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 (NOTE: This should never be done in production).

As the MySQL image was not available locally, Docker automatically pulled it from Docker Hub.

```
Unable to find image 'mysql:latest' locallylatest: Pulling from library/mysql
aa18ad1a0d33: Pull complete
fdb8d83dece3: Pull complete
75b6ce7b50d3: Pull complete
ed1d0a3a64e4: Pull complete
8eb36a82c85b: Pull complete
41be6f1a1c40: Pull complete
0e1b414eac71: Pull complete
914c28654a91: Pull complete
587693eb988c: Pull complete
b183c3585729: Pull complete
315e21657aa4: Pull complete
Digest: sha256:0dc3dacb751ef46a6647234abdec2d47400f0dfbe77ab490b02bffdae57846
Status: Downloaded newer image for mysql:latest
41d6157c9f7d1529a6c922acb8167ca66f167119df0fe3d86964db6c0d7ba4e0
```

As long as the MySQL process is running, Docker will keep the container running in the background.

2. List the running containers.

```
docker container ls
```

Notice your container is running.

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
3f4e8da0caf7        mysql:latest        "docker-entrypoint..."   52 seconds a
```

3. You can check what's happening in your containers by using a couple of built-in Docker commands: `docker container logs` and `docker container top`.

```
docker container logs mydb
```

This shows the logs from the MySQL Docker container.

```
  <output truncated>
  2017-09-29T16:02:58.605004Z 0 [Note] Executing 'SELECT * FROM INFORMATION_S
  2017-09-29T16:02:58.605026Z 0 [Note] Beginning of list of non-natively part
  2017-09-29T16:02:58.616575Z 0 [Note] End of list of non-natively partitione
```

Let's look at the processes running inside the container.

```
  docker container top mydb
```

You should see the MySQL daemon (`mysqld`) is running in the container.

```
PID                 USER                TIME                COMMAND
2876                999                 0:00                mysqld
```

Although MySQL is running, it is isolated within the container because no network ports have been published to the host. Network traffic cannot reach containers from the host unless ports are explicitly published.

4. List the MySQL version using `docker container exec`.

`docker container exec` allows you to run a command inside a container. In this example, we'll use `docker container exec` to run the command-line equivalent of `mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version` inside our MySQL container.

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

You will see the MySQL version number, as well as a handy warning.

```
mysql: [Warning] Using a password on the command line interface can be insecu
mysql  Ver 14.14 Distrib 5.7.19, for Linux (x86_64) using  EditLine wrapper
```

5. 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 (`sh`) inside your MySQL container.

   ```
   docker exec -it mydb sh
   ```

   Notice that your shell prompt has changed. This is because your shell is now connected to the `sh` process running inside of your container.

6. Let's check the version number by running the same command again, only this time from within the new shell session in the container.

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

   Notice the output is the same as before.

7. Type `exit` to leave the interactive shell session.

   ```
   exit
   ```

Task 2: Package and run a custom app using Docker

In this step you'll learn how to package your own apps as Docker images using a Dockerfile.

The Dockerfile syntax is straightforward. In this task, we're going to create a simple NGINX website from a Dockerfile.

Build a simple website image

Let’s have a look at the Dockerfile we’ll be using, which builds a simple website that allows you to send a tweet.

1. Make sure you’re in the `linux_tweet_app` directory.

```
cd ~/linux_tweet_app
```

2. Display the contents of the Dockerfile.

```
cat Dockerfile
```

```
FROM nginx:latest

COPY index.html /usr/share/nginx/html
COPY linux.png /usr/share/nginx/html

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

   Let’s see what each of these lines in the Dockerfile do.

   - FROM specifies the base image to use as the starting point for this new image you’re creating. For this example we’re starting from `nginx:latest`.
   - COPY copies files from the Docker host into the image, at a known location. In this example, `COPY` is used to copy two files into the image: `index.html`. and a graphic that will be used on our webpage.
   - EXPOSE documents which ports the application uses.
   - CMD specifies what command to run when a container is started from the image. Notice that we can specify the command, as well as run-time arguments.

3. In order to make the following commands more copy/paste friendly, export an environment variable containing your DockerID (if you don’t have a DockerID you can get one for free via Docker Cloud).

   You will have to manually type this command as it requires your unique DockerID.

   `export DOCKERID=<your docker id>`

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

```
echo $DOCKERID
```

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

    - `--tag` allows us to give the image a custom name. In this case it's comprised of our DockerID, the application name, and a version. Having the Docker ID attached to the name will allow us to store it on Docker Hub in a later step
    - `.` tells Docker to use the current directory as the build context

    Be sure to include period (`.`) at the end of the command.

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

    The output below shows the Docker daemon executing each line in the Dockerfile

    ```
    Sending build context to Docker daemon  32.77kB
    Step 1/5 : FROM nginx:latest
    latest: Pulling from library/nginx
    afeb2bfd31c0: Pull complete
    7ff5d10493db: Pull complete
    d2562f1ae1d0: Pull complete
    Digest: sha256:af32e714a9cc3157157374e68c818b05ebe9e0737aac06b55a09da374209a8
    Status: Downloaded newer image for nginx:latest
    ---> da5939581ac8
    Step 2/5 : COPY index.html /usr/share/nginx/html
    ---> eba2eec2bea9
    Step 3/5 : COPY linux.png /usr/share/nginx/html
    ---> 4d080f499b53
    Step 4/5 : EXPOSE 80 443
    ---> Running in 47232cb5699f
    ---> 74c968a9165f
    Removing intermediate container 47232cb5699f
    Step 5/5 : CMD nginx -g daemon off;
    ---> Running in 4623761274ac
    ---> 12045a0df899
    Removing intermediate container 4623761274ac
    Successfully built 12045a0df899
    Successfully tagged <your docker ID>/linux_tweet_app:latest
    ```

6. Use the `docker container run` command to start a new container from the image you created.

    As this container will be running an NGINX web server, we'll use the `--publish` flag to publish port 80 inside the container onto port 80 on the host. This will allow traffic coming in to the Docker host on port 80 to be directed to port 80 in the container. The format of the `--publish` flag is `host_port`:`container_port`.

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

Any external traffic coming into the server on port 80 will now be directed into the container on port 80.

In a later step you will see how to map traffic from two different ports - this is necessary when two containers use the same port to communicate since you can only expose the port once on the host.

7. Click here to load the website which should be running.

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

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

> **Note:** We used the `--force` parameter to remove the running container without shutting it down. This will ungracefully shutdown the container and permanently remove it from the Docker host.
>
> In a production environment you may want to use `docker container stop` to gracefully stop the container and leave it on the host. You can then use `docker container rm` to permanently remove it.

Task 3: Modify a running website

When you're actively working on an application it is inconvenient to have to stop the container, rebuild the image, and run a new version every time you make a change to your source code.

One way to streamline this process is to mount the source code directory on the local machine into the running container. This will allow any changes made to the files on the host to be immediately reflected in the container.

We do this using something called a bind mount.

When you use a bind mount, a file or directory on the host machine is mounted into a container running on the same host.

Start our web app with a bind mount

1. Let's start the web app and mount the current directory into the container.

   In this example we'll use the `--mount` flag to mount the current directory on the host into `/usr/share/nginx/html` inside the container.

   Be sure to run this command from within the `linux_tweet_app` directory on your Docker host.

   ```
   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
   ```

   > Remember from the Dockerfile, `usr/share/nginx/html` is where the html files are stored for the web app.

2. The [website] should be running.

Modify the running website

Bind mounts mean that any changes made to the local file system are immediately reflected in the running container.

1. Copy a new `index.html` into the container.

   The Git repo that you pulled earlier contains several different versions of an index.html file. You can manually run an `ls` command from within the `~/linux_tweet_app` directory to see a list of them. In this step we'll replace `index.html` with `index-new.html`.

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

2. Go to the running [website] and **refresh the page**. Notice that the site has changed.

   > If you are comfortable with `vi` you can use it to load the local `index.html` file and make additional changes. Those too would be reflected when you reload the webpage. If you are really adventurous, why not try using `exec` to access the running container and modify the files stored there.

Even though we've modified 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.

To show this, stop the current container and re-run the `1.0` image without a bind mount.

1. Stop and remove the currently running container.

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

2. 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
```

3. Notice the website is back to the original version.

4. Stop and remove the current container

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

Update the image

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

1. Build a new image and tag it as `2.0`

   Remember that you previously modified the `index.html` file on the Docker hosts local filesystem. This means that running another `docker image build` command will build a new image with the updated `index.html`

   Be sure to include the period (`.`) at the end of the command.

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

   Notice how fast that built! This is because Docker only modified the portion of the image that changed vs. rebuilding the whole image.

2. Let's look at the images on the system.

```
docker image ls
```

You now have both versions of the web app on your host.

```
REPOSITORY                     TAG                 IMAGE ID            CREATE
<docker id>/linux_tweet_app    2.0                 01612e05312b        16 sec
<docker id>/linux_tweet_app    1.0                 bb32b5783cd3        4 minu
mysql                          latest              b4e78b89bcf3        2 week
ubuntu                         latest              2d696327ab2e        2 week
nginx                          latest              da5939581ac8        3 week
alpine                         latest              76da55c8019d        3 week
```

Test the new version

1. 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
```

2. Check the new version of the website (**You may need to refresh your browser to get the new version to load**).

   The web page will have an orange background.

   We can run both versions side by side. The only thing we need to be aware of is that we cannot have two containers using port 80 on the same host.

   As we're already using port 80 for the container running from the `2.0` version of the image, we will start a new container and publish it on port 8080. Additionally, we need to give our container a unique name (`old_linux_tweet_app`)

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

   Notice that this command maps the new container to port 8080 on the host. This is because two containers cannot map to the same port on a single Docker host.

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

4. View the old version of the website.

Push your images to Docker Hub

1. List the images on your Docker host.

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

You will see that you now have two `linux_tweet_app` images - one tagged as `1.0` and the other as `2.0`.

```
REPOSITORY                      TAG                 IMAGE ID            CREATE
<docker id>/linux_tweet_app     2.0                 01612e05312b        3 minu
<docker id>/linux_tweet_app     1.0                 bb32b5783cd3        7 minu
```

These images are only stored in your Docker hosts local repository. Your Docker host will be deleted after the workshop. In this step we'll push the images to a public repository so you can run them from any Linux machine with Docker.

Distribution is built into the Docker platform. You can build images locally and push them to a public or private registry, making them available to other users. Anyone with access can pull that image and run a container from it. The behavior of the app in the container will be the same for everyone, because the image contains the fully-configured app - the only requirements to run it are Linux and Docker.

Docker Hub is the default public registry for Docker images.

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

```
docker login
```

You will need to supply your Docker ID credentials when prompted.

```
Username: <your docker id>
Password: <your docker id password>
Login Succeeded
```

3. Push version `1.0` of your web app using `docker image push`.

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

You'll see the progress as the image is pushed up to Docker Hub.

```
The push refers to a repository [docker.io/<your docker id>/linux_tweet_app]
910e84bcef7a: Pushed
1dee161c8ba4: Pushed
110566462efa: Pushed
305e2b6ef454: Pushed
24e065a5f328: Pushed
1.0: digest: sha256:51e937ec18c7757879722f15fa1044cbfbf2f6b7eaeeb578c7c352bab
```

4. Now push version `2.0`.

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

Notice that several lines of the output say `Layer already exists`. This is because Docker will leverage read-only layers that are the same as any previously uploaded image layers.

```
The push refers to a repository [docker.io/<your docker id>/linux_tweet_app]
0b171f8fbe22: Pushed
70d38c767c00: Pushed
110566462efa: Layer already exists
305e2b6ef454: Layer already exists
24e065a5f328: Layer already exists
2.0: digest: sha256:7c51f77f90b81e5a598a13f129c95543172bae8f5850537225eae0c78
```

You can browse to `https://hub.docker.com/r/<your docker id>/` and see your newly-pushed Docker images. These are public repositories, so anyone can pull the image - you don't even need a Docker ID to pull public images. Docker Hub also supports private repositories.