[Creating the first Docker containers URL]

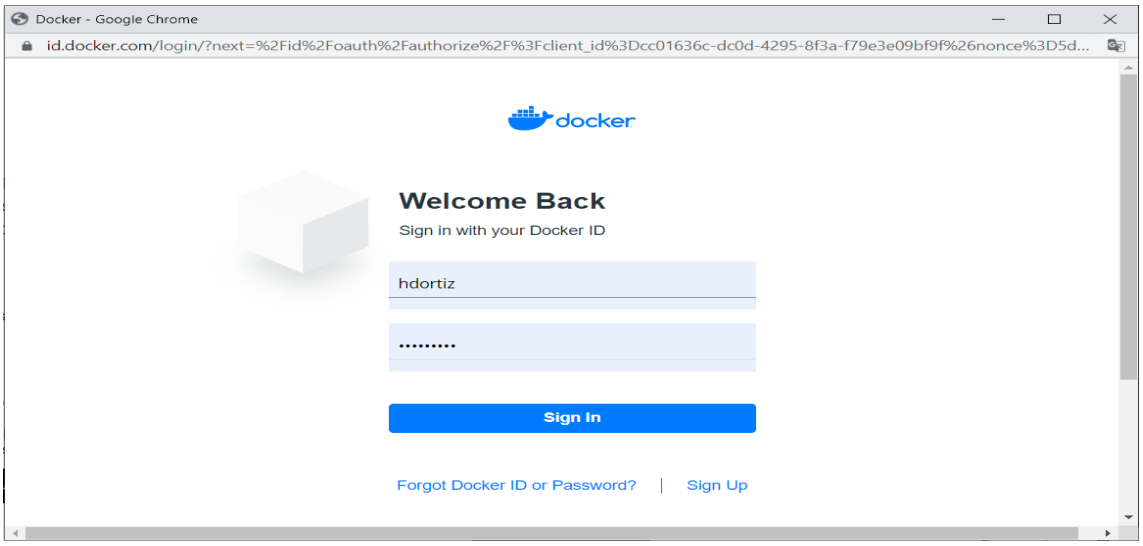
Dokers for beginers

From the page https://training.play-with-docker.com/beginner-linux/ .

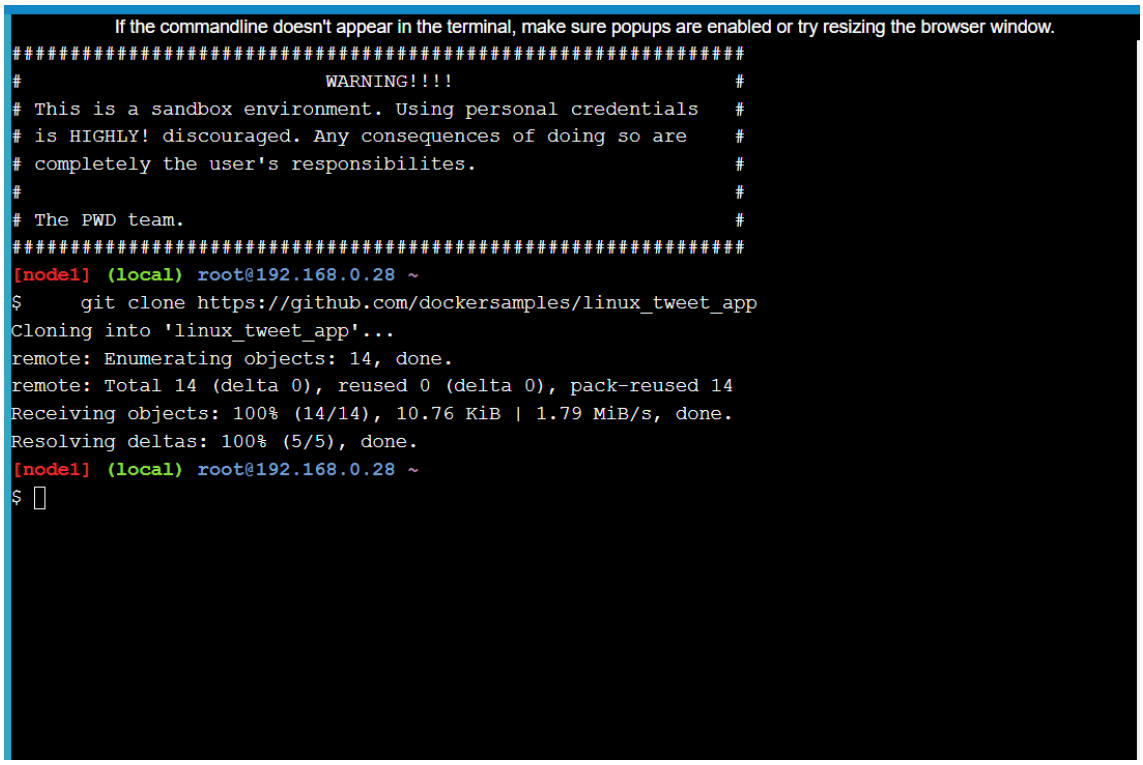
First, we need to make sure meeting with the prerequisites:

- A clone of the lab´s GIT HUB repo.
- A DokerId.

Create a account:



A clone of the lab´s GIT HUB repo:



Creating the first Docker containers URL

TASK 1: Run some simple Docker containers

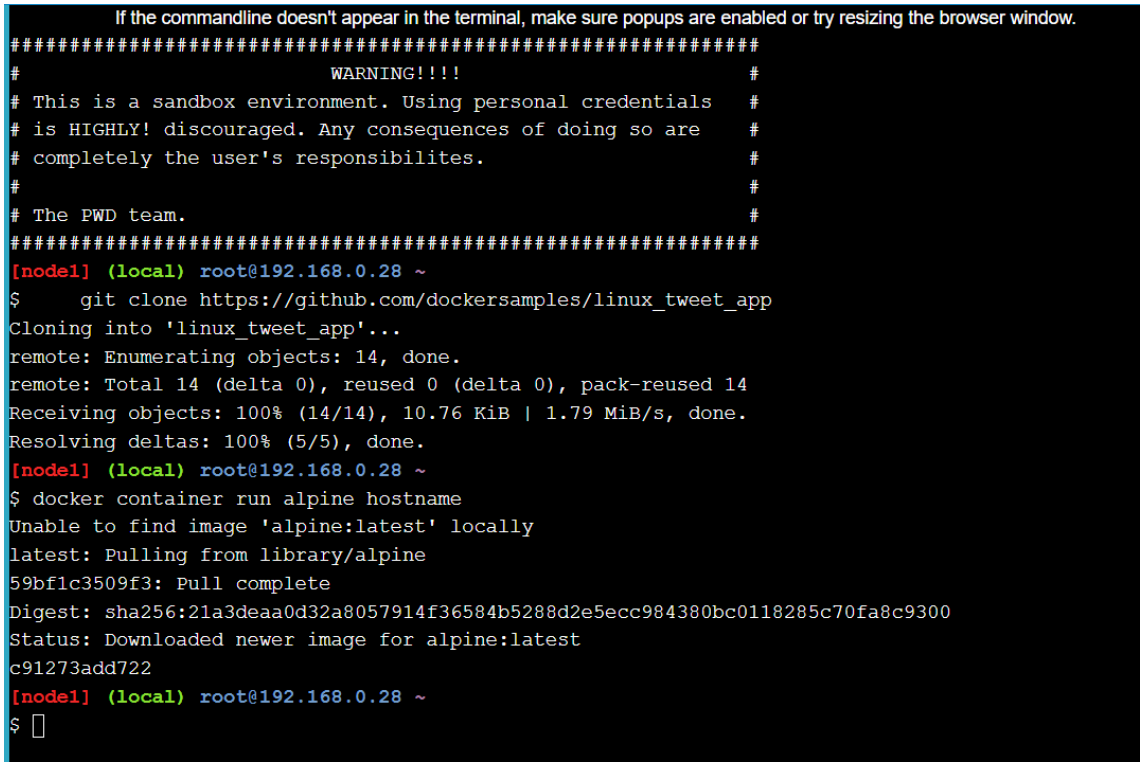
There are different ways to use containers. These include:

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.

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).

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.

[Creating the first Docker containers URL]

```
[node1] (local) root@192.168.0.28 ~
$ docker container ls --all
CONTAINER ID   IMAGE     COMMAND      CREATED         STATUS                     PORTS     NAMES
c91273add722   alpine    "hostname"   3 minutes ago   Exited (0) 3 minutes ago             stupe
fied_visvesvaraya
[node1] (local) root@192.168.0.28 ~
$
```

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
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
7b1a6ab2e44d: Pull complete
Digest: sha256:626ffe58f6e7566e00254b638eb7e0f3b11d4da9675088f4781a50ae288f3322
Status: Downloaded newer image for ubuntu:latest
root@d7afd0fb482f:/#
```

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 20.04.3 LTS.

Creating the first Docker containers URL

```
root@d7afd0fb482f:/# ls /
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@d7afd0fb482f:/#
```

```
root@d7afd0fb482f:/# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   4108  3552 pts/0    Ss   11:54   0:00 bash
root          11  0.0  0.0   5896  2920 pts/0    R+   11:58   0:00 ps aux
root@d7afd0fb482f:/#
```

```
root@d7afd0fb482f:/# cat /etc/issue
Ubuntu 20.04.3 LTS \n \l

root@d7afd0fb482f:/#
```

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

```
root@d7afd0fb482f:/# exit
exit
[node1] (local) root@192.168.0.28 ~
$
```

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
Welcome to Alpine Linux 3.12
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.

[Creating the first Docker containers URL]

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.

```
[node1] (local) root@192.168.0.28 ~
$  docker container run \
>  --detach \
>  --name mydb \
>  -e MYSQL_ROOT_PASSWORD=my-secret-pw \
>  mysql:latest
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
ffbb094f4f9e: Pull complete
df186527fc46: Pull complete
fa362a6aa7bd: Pull complete
5af7cb1a200e: Pull complete
949da226cc6d: Pull complete
bce007079ee9: Pull complete
eab9f076e5a3: Pull complete
8a57a7529e8d: Pull complete
b1ccc6ed6fc7: Pull complete
b4af75e64169: Pull complete
3aed6a9cd681: Pull complete
23390142f76f: Pull complete
Digest: sha256:ff9a288d1ecf4397967989b5d1ec269f7d9042a46fc8bc2c3ae35458c1a26727
Status: Downloaded newer image for mysql:latest
9409c186d1d7ffcde71ca9c843847dfac0018c43d142494f5583129f2030e7d1
[node1] (local) root@192.168.0.28 ~
$
```

- `--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.

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
CONTAINER ID   IMAGE          COMMAND                  CREATED              STATUS
PORTS                 NAMES
9409c186d1d7   mysql:latest   "docker-entrypoint.s…"   About a minute ago   Up About a minute
3306/tcp, 33060/tcp   mydb
[node1] (local) root@192.168.0.28 ~
$
```

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`.

This shows the logs from the MySQL Docker container.

Creating the first Docker containers URL

```
$  docker container logs mydb
2021-12-15 12:08:17+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.27-1debian
10 started.
2021-12-15 12:08:17+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2021-12-15 12:08:17+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.27-1debian
10 started.
2021-12-15 12:08:17+00:00 [Note] [Entrypoint]: Initializing database files
2021-12-15T12:08:17.760777Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 8.0.27) ini
tializing of server in progress as process 44
2021-12-15T12:08:17.783564Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2021-12-15T12:08:18.265376Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2021-12-15T12:08:19.981278Z 0 [Warning] [MY-013746] [Server] A deprecated TLS version TLSv1 is e
nabled for channel mysql_main
2021-12-15T12:08:19.981335Z 0 [Warning] [MY-013746] [Server] A deprecated TLS version TLSv1.1 is
 enabled for channel mysql_main
2021-12-15T12:08:20.065619Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an e
mpty password ! Please consider switching off the --initialize-insecure option.
```

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

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

```
[node1] (local) root@192.168.0.28 ~
$    docker container top mydb
PID                 USER                TIME                COMMAND
9947                999                 0:01                mysqld
[node1] (local) root@192.168.0.28 ~
```

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.

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

```
$  docker exec -it mydb \
>  mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
mysql: [Warning] Using a password on the command line interface can be insecure.
mysql  Ver 8.0.27 for Linux on x86_64 (MySQL Community Server - GPL)
[node1] (local) root@192.168.0.28 ~
```

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.

Creating the first Docker containers URL

```
#  mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
mysql: [Warning] Using a password on the command line interface can be insecure.
mysql  Ver 8.0.27 for Linux on x86_64 (MySQL Community Server - GPL)
#
```

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
sh: 2: cd: can't cd to /root/linux_tweet_app
#
```

2. Display the contents of the Dockerfile.

```
#  cat Dockerfile
cat: Dockerfile: No such file or directory
#
```