Master of Engineering in Internetworking

INWK 6312

Programming for Internetworking

Lab 1

Container based virtualization technology using Docker

Contents

Introduction

In the simplest sense, containers are just a way of isolating running processes or code without using what we know as virtual machines (VMs) or full virtualisation. Containers allow developers to build, package, share, and deploy applications. Containers provide a way to:

- Package applications and dependencies.
- Guarantee portability and consistency of execution.
- Keep an application isolated.
- Take advantage of the isolation offered by a VM without the overhead.

Imagine being able to guarantee that App A and App B cannot see or interfere with each other on the same host (each does not even know that the other exists) without having to spin up a full VM and operating system for each. Containers give us this functionality.

How do they work?

Containers achieve this isolation through features built into the operating system. Just as a hypervisor is responsible for creating and managing VMs, Linux and most other modern OSes can now create isolated containers. In Linux, this technology is called CGroups. CGroups allow you to portion off resources from the system.

CGroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

Both Windows and macOS have similar isolation features for containerizing apps and processes. It's important to note is that there is only one kernel, with possibly many containers, all running on one operating system that manages the isolation of the containers. In contrast, virtualization requires installing a whole operating system for each virtual machine.

What Containers ARE NOT:

- Microservices: Containers are often confused with microservices. Microservices are lightweight system services, which can benefit from the low overhead provided by containers, especially when splitting up multiple tasks that would previously have been handled within an older, more complex application, but they are not the same thing as containers. Any application can be packaged using containers; it doesn't have to be a microservice. Even a badly written legacy application can be packaged in a container. Using a container doesn't magically improve an application.

- Virtual Machines: Containers are not VMs. Containers run entirely in user space. If an application requires kernel extensions, kernel modules, or a custom kernel then a container is probably not the right solution for it.

- Magic: A container is a tool like other tools. It has its own limitations and nuances and its own deployment considerations, just like any other tool.

Objectives

This Lab introduces Docker, a tool suite for building, sharing, and deploying containers. You will learn how to build, deploy, and optionally share your first Docker container. The knowledge gained from this lab will propel you to do well as a DevOps Engineer, Application developer, System Architect/Engineer, or a member of an IT team addressing developer needs for Docker and Containers.

Lab Environment and Preparation

The Lab virtual machine is already installed with Docker, and you are urged to run all your tasks in it. However, you can also use an in-browser Docker playground called *play-with-docker.com* (Provided by the Docker team). The *play-with-docker.com* website provides access to a full VM running Docker directly in a web browser, making it easy to work with Docker from any device. This link is below:

http://labs.play-with-docker.com/

If you are using the Play-with-docker.com environment, click start, create an account, and start a VM instance. Once you are done, you can skip the rest of this section and proceed to Task 1.

If you are using the Lab environment, then follow the steps below:

Connect to the VM Machine.

The Linux VM Address is in the format 10.0.134.1**YY**. Where **YY** is the POD number. For example, POD 1 is 10.0.134.1**01** and Pod 56 is 10.0.134.1**56**.

Username: student
Password: Meilab123

1. Connect to the Linux VM for your Pod and create a new directory on your Desktop.

```
$ ssh student@10.0.134.1YY

student@6312vm-csr00$ cd Desktop
student@6312vm-csr00$ mkdir docker-lab && cd docker-lab
```

2. Confirm Docker is installed.

```
student@6312vm-csr00$ docker -v
Docker version 20.10.6, build 370c289
```

Task 0 – Getting and Running a Docker Container

The simplest way to use Docker is to run an existing public image that's available from Docker Hub.

Docker Hub is a public exchange for sharing Docker containers. Other Docker sharing sites are available, but we'll take advantage of the fact that Docker's command-line interface searches Docker Hub by default.

To run a publicly available Docker Container, follow these steps:

1. In the browser terminal, execute the following command to find the "hello-world" image:

   ```
   student@6312vm-csr00$ docker search hello-world
   ```

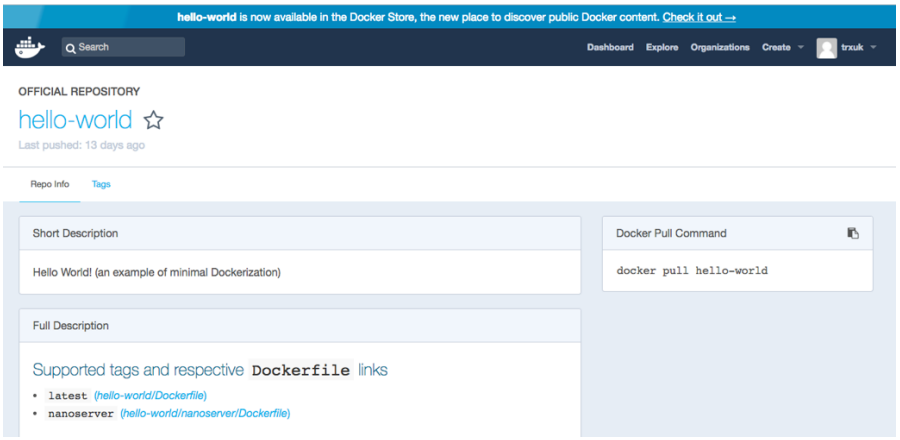
Docker searches the public Docker Hub repositories and finds the "hello-world" image. You can see that Docker has found the "`hello-world`" image. Let's run it.

2. Execute the following command to run the hello-world:

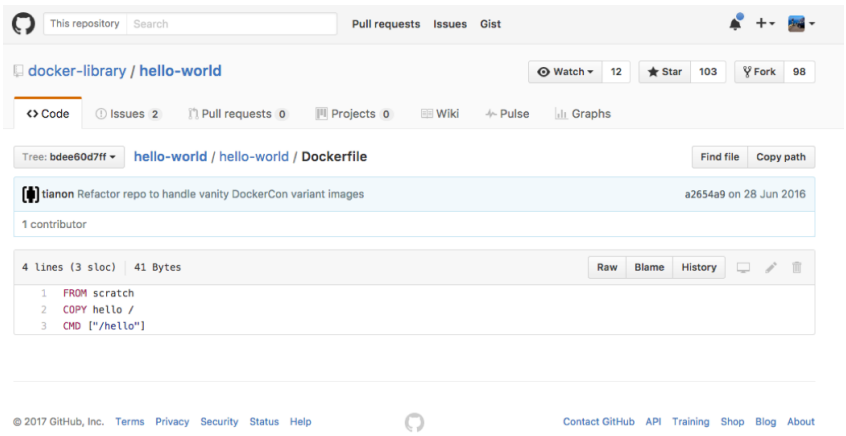
   ```
   student@6312vm-csr00$ docker run hello-world
   ```

Docker first checks to see whether the "hello-world" image is available locally. If not, Docker automatically downloads it from Docker Hub. Docker sets up the container to run locally, ensuring its isolation from other processes. Once the preparations are made, Docker runs the image.

How did the author of "hello-world" create the container image? Because it's available on the public Docker Hub repository, we can find out by inspecting the files. Browse to the "hello world" web page to see details describing the Docker image → https://hub.docker.com/_/hello-world/



Among those details is a link to the Dockerfile, the file that defines the image. (You may need to login and click that link a second time.)



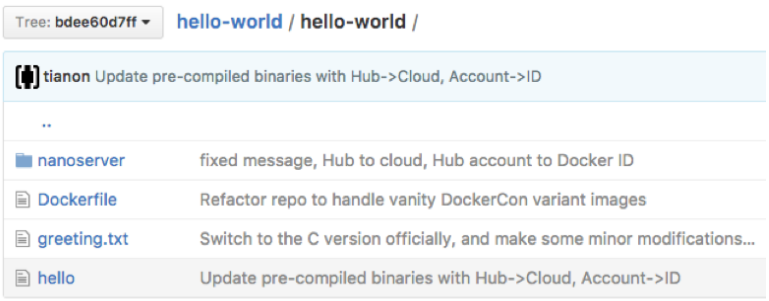
The Dockerfile

Every Docker image, from a simple one like "hello-world" to a complex application, is defined by a Dockerfile. The Dockerfile is a text file that lists the programs and resources that become part of the Docker image, and which instructs Docker what to do when the container runs. An author creates a Docker image by assembling application source code and other resource files and writing a Dockerfile that turns them into a working container image.

To create the container image, the author executes the docker build command from the directory that contains the Dockerfile. Docker reads the Dockerfile and follows the instructions in it to build the container image. Once the Docker image is built, the author can push it to the public repository on Docker Hub, making it available to everyone.

Most container authors store their Dockerfiles in a version-control repository along with the files used in building their images. Storing a Dockerfile along with the files that it uses to build a container simplifies both the Dockerfile itself and the process of building the container image.

The "hello-world" repository illustrates this practice:



All of the files used in this very simple container are stored in the repository together with the Dockerfile.

The "hello-world" Dockerfile

This is the hello-world Dockerfile:

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

Here's what it tells `docker build` to do:

`FROM scratch`

"FROM scratch" means that the new Docker image starts with a fresh, empty container. It's also possible to build an image based on one that already exists. To do that, change "scratch" to the name of the starting container.

`COPY hello /`

Copy the local file 'hello' into the root of the container image. Then reason for storing Dockerfiles together with related application files is so that the Dockerfile can conveniently refer to those files and direct Docker to add them to a container.

`CMD ["/hello"]`

Run the command "hello" when the container image starts. When someone uses `docker run` to execute this container image, Docker automatically runs the program identified in this line.

OS Userspace Containers

A container image can be built with a complete Linux userspace inside it. You can, for example, build a Docker container image that contains all of the parts of the Ubuntu system needed to run NGINX.

On the one hand, building a whole userspace into a container means that the container image may be quite large—perhaps hundreds of megabytes.

On the other hand, it means that even a complex application with many dependencies can be delivered in the form of a container image. The container can even include operating-system package-management software like APT or YUM for use in configuring optional dependencies.

Suppose you want to build a container image that includes the whole Ubuntu system. Begin by searching Docker Hub for a suitable container to start with:

```
student@6312vm-csr00$ docker search ubuntu
```

We're in luck; Docker Hub has an existing Ubuntu container.

Run the container:

```
student@6312vm-csr00$ docker run -ti ubuntu
```

The example command downloads and runs the Ubuntu container from Docker Hub. The command-line flags "-ti" tell Docker that we want to interact with a shell in the running image. As soon as the image finishes launching it dutifully presents us with a shell:

```
root@dec585fddd93:/#
```

Testing Isolation

The processes that run in a Docker container are completely isolated from the outside world. Files and processes on the host machine are invisible to the application in the container. The container application can't create or interact with any files or other resources outside the container.

For example, suppose you use the shell in the container to create a new file at the root of the filesystem:

```
root@dec585fddd93:/# touch /hello
root@dec585fddd93:/# ls /
```

The new file, hello, appears in the root of the filesystem. Now leave the container and examine the host filesystem:

```
root@dec585fddd93:/# exit
exit
student@6312vm-csr00$ ls /
```

File hello isn't there. Why not? Because the host filesystem and the container filesystem are **completely separate**. Creating the "hello" file in the root of the container had no effect on the filesystem of the host.

In this case, typing `exit` in the container kills the container and its application. We can see which containers are running on the host using this command: `docker container ls` OR `docker ps`

```
student@6312vm-csr00$ docker container ls
```

In this case no containers are running. We started only the "hello-world" container, and our exit command stopped it.

To view all containers, both running and exited, run:

```
student@6312vm-csr00$ docker container ls -a
```

If we run `docker run -ti ubuntu` again it creates a brand new instance of the Ubuntu container. Because the container instance is freshly created, the /hello file won't be there. That file is something that we created by interacting with the container, not a resource that was built into the container image. Because it's not part of the image, it's not present when the container runs.

It's important to remember that changes to a running Docker container do not survive after the container is killed. If you want to add a resource to a Docker container and make it available in all future sessions, then you must modify the Dockerfile to include the resource and then rebuild the image.

Task 1: Running a Python3 Script in a Container

In this task, you are asked to:

1. Search for a minimal Docker image based on Alpine Linux.
2. Run the image as an interactive container and interact with the shell.
3. Install the **VIM** text editor. You need to research the package manager for the Alpine Linux and use it to download / install the program. Update your package manager first!
4. Create a new directory called "`app`" and inside this directory, create a new file called "`hello.py`" and type the following:

```
print("hello world from alpine container")
```

5. Run the Python script: "`python3 hello.py`". Did you get any Error? Try to fix it!

Task 2: Running a Python Script in a Container-2

In this task, you are asked to:

1. Visit the Docker Hub and search for "Python" image. Browse through the official Python Image. Under "Simple Tags", notice the different tags that is associated with this Image.
2. Run the Python image with `tag 3.6` as an interactive container.
3. After step 2, you should be in a Python shell. Run any Python3 syntax and exit.
4. Run the same Python Image again as an interactive container but starting from its terminal `bash` shell.
5. Install the **VIM** text editor. You need to research the package manager for this Linux distribution and use it to download / install the program. Update your package manager first!
6. Create a new director called "`app`" and inside this directory, create a new file called "`hello.py`" and type the following:

```
print("hello world from Python3 container")
```

7. Run the Python script: "`python3 hello.py`".

Delete all containers and images on your Linux machine by using the commands:

To delete all containers:

```
docker container rm $(docker container ls -a -q)
```

To delete all Images:

```
docker image rm $(docker image ls -a -q)
```

Task 3: Building a Docker Image

Next, we will learn how to build our own Ubuntu Docker image instead of using the one from Docker Hub.

1. Download the Ubuntu Dockerfile and its dependencies.

```
student@6312vm-csr00$ git clone --single-branch --branch dist-amd64
https://github.com/tianon/docker-brew-ubuntu-core.git
```

2. Change directory to the cloned repository and ensure that the Dockerfile is present.

```
student@6312vm-csr00$ cd docker-brew-ubuntu-core/xenial
student@6312vm-csr00$ ls .
```

3. Build a new image using Docker and the Dockerfile.

```
student@6312vm-csr00$ docker image build .
```

Docker prints progress messages as it builds the Ubuntu image. Once it finishes building the image, Docker assigns it a randomly chosen name.

4. Run the command:

```
student@6312vm-csr00$ docker images
```

Docker lists all the images on your system, including both the newly built one and any that you previously downloaded. You can tell which image is the one you just created by examining the list; it's the only image that isn't associated with a repository.

5. Run the newly created Docker image by giving the randomly-chosen ID to Docker:

```
student@6312vm-csr00$ docker run -ti <your image ID>
```

The newly-created image behaves exactly the same way as the Ubuntu image from Docker Hub, because it is built from the same Dockerfile. You now have a bash root prompt.

6. Run the following command to confirm standard Ubuntu directories are installed:

```
student@6312vm-csr00$ ls
```

7. Exit and return to your Home Linux terminal by typing **exit**

Task 4: Building a Custom Image from a Dockerfile

We will create a Dockerfile that contains instructions required to build the custom image that will display the following message: "*Hello from Internetworking Class*".

To make the image display the above message we will:

- Create a new Python script.
- Edit the Dockerfile to include that script in the container build.
- Change the Dockerfile to install Python in the container (remember, all dependencies must be present in the container).
- Build and test the new container.

1. Navigate to your starting directory of this lab and make a new directory called `custom-container`

```
student@6312vm-csr00$ cd ~/Desktop/docker-lab
student@6312vm-csr00$ mkdir custom-container && cd custom-container
```

2. Create a new file: "`helloinwk.py`" to contain the following Python code:

```
#!/usr/bin/env python3
print("Hello from Internetworking Class!")
```

3. In the same directory, create another file called: "`Dockerfile`" to contain the following instructions:

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install python3
COPY helloinwk.py /helloinwk.py
RUN ["chmod", "+x", "/helloinwk.py"]
CMD ["/helloinwk.py"]
```

This Dockerfile says:

`FROM ubuntu`

Extend the existing ubuntu public Docker image. Our previous examples built an image from scratch. In this case we begin with a previously built container image and extend with our own customisations.

`RUN apt-get update`

Ensure the package-management tools in the base Ubuntu container are updated to use the latest software.

`RUN apt-get -y install python`

Use apt-get to install Python and all its dependencies in the container. The reason to build this container on an existing Ubuntu image is that we could use apt-get to install needed software.

`COPY helloinwk.py /helloinwk.py`

Copy the Python program from the local directory into the container as /hellodevnet.py.

`RUN ["chmod", "+x", "/helloinwk.py"]`

Grant permission to execute the /hellodevnet.py file

`CMD ["/helloinwk.py"]`

Run the Python program when the container starts up.

4. Build the Docker image using the following command:

```
student@6312vm-csr00$ docker image build .
```

Docker prints progress messages as it builds the Ubuntu image. Once it finishes building the image, Docker displays: *Succesfully built <CONTAINER ID> message*. Make note of the newly built <CONTAINER ID> so that you can use it in the next step.

5. Run the new container using the following command. Use the <CONTAINER ID> that you received from the `docker build .` command output.

```
student@6312vm-csr00$ docker container run <container-id>
```

The "Hello from Internetworking!" message is displayed in the container terminal. You can also use the command: **`docker run <container-id>`**

Task 5: Building a web server container

A Docker container image can run any application you like. Let's build one that runs a web server. The web server will run whenever the container image starts up.

The following Python command creates and starts a simple web server:

```
python -m SimpleHTTPServer 8000
```

To build a container that runs this web server we need to change the CMD line of the Dockerfile, the line that controls the program that runs when the container starts.

1. Navigate to your starting directory of this lab and make a new directory called `webserver`

```
student@6312vm-csr00$ cd ~/Desktop/docker-lab
student@6312vm-csr00$ mkdir webserver && cd webserver
```

2. Create your Dockerfile to contain the following instructions:

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install python3
EXPOSE 8000
ENTRYPOINT ["python3", "-m", "http.server", "8000"]
```

The first three lines are familiar. The last two lines introduce the new Dockerfile options.

`EXPOSE 8000`

The EXPOSE command allows us as developers to "build in" documentation of what ports our application uses. Running this container in future with **`-P`** will automatically map any EXPOSE ports in the Dockerfile to a dynamic port on our hosts real IP, allowing real remote connections to our service if necessary. You can see what port has been mapped by using the `docker container ls` or `docker ps` **command**.

3. Build and run the new web server container image using the following command:

```
student@6312vm-csr00$ docker build -t webserver .
student@6312vm-csr00$ docker run -P <your new CONTAINER ID> &
```

Why don't we see any output?

Every time before when we started a container, we saw output from the container's application. The reason we see no output this time is that our application, a web server, doesn't print any message. So how do we know whether it's running correctly?

4. Run the following Docker command to check if the new web server image is running:

```
student@6312vm-csr00$ docker container ls
```

The command lists all running containers including the new web server image that we created:

The web server is running. Now how do we connect to it? By default Docker gives each container an internal IP address on the host. Make note of the <CONTAINER ID> of the new web server image so that you can use it in the next step.

5. Run the following command to find out the container's IP address.

```
student@6312vm-csr00$ docker inspect <CONTAINER ID>
```

Running this command displays a great deal of information about the container. Included in the output is the container's address, identified as "`IPAddress`".

```
……..
…
"MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "f13fe1e8de93520dc15b1f1b9562d78806565de0623d83d2cd29f18fbe1aeeea",
                    "EndpointID": "83be9164ce7d21d2bbc098ecc6f2da4f08d52a904071320aa70d195d173a65c4",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
……….
…
```

Now that we know the IP address and port on which the web server can be reached, we can make a connection. Make note of the IPAddress of the container, so that you can use it in the next step.

6. Run the following command to make the connection. You can use the IP address of the container that is received from the `docker inspect <CONTAINER ID>` command output.

```
student@6312vm-csr00$ curl http://<CONTAINER IPAddress>:8000
```

The built-in Python web server by default serves the contents of its working directory. Because the Dockerfile copied the Python script to the root of the container, that's the directory that the web server displays

Task 6: Containerizing an Existing Application

In this task, you will containerize an application that is already written for you. Do the following steps:

1. Clone the project's repository from Github.

```
student@6312vm-csr00$ git clone -b v1 https://github.com/inwk6312course/dockerapp.git
student@6312vm-csr00$ cd dockerapp
student@6312vm-csr00$ ls
```

2. Build the Image from the Dockerfile found in the repository. Review the contents of the Dockerfile before building. The `-t` flag is used to tag the image to a custom name of our choice. We can also tag an image, think of it as putting a version on the image. In this case we tag the image `v1`

```
student@6312vm-csr00$ docker build -t dockerapp:v1 .
```

3. Copy the Image id of the newly built image by running

```
student@6312vm-csr00$ docker image ls
```

4. Run the image as a container. The `-d` flag run the container in detached mode.

```
student@6312vm-csr00$ docker run -d -p 5000:5000 <<image-id>>
```

5. Visit the homepage of the application with the Curl program

```
student@6312vm-csr00$ curl localhost:5000
```

Clean Up!

Stop and remove the container by using the commands:

Look for the id of the container to stop:

```
docker container ls
```

To stop a container gracefully:

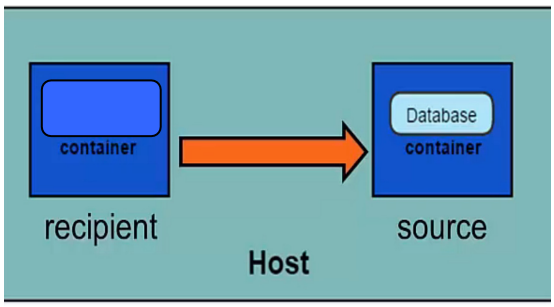
```
docker container stop <<container-id>>
```

To delete a container gracefully:

```
docker container rm <<container-id>>
```


Container Links

Container don't always have to be isolated from each other. They can communicate within themselves and to the outside world. We would be working with an application that consists of 2 different servers. A python webserver and a Redis server. Redis is memory data structure store, used as a database, cache, and message broker. We will containerize the two servers and our overall application structure looks like this:



Task 7: Linking Two Containers

1. Start up a Redis container.

```
student@6312vm-csr00$ docker run -d --name redis redis:3.2.0
```

2. Clone the Python web application.
   If you are already working from the cloned Git repository from the previous task, then you can use the `checkout` command to change to a different branch (v2) which contains the updated code and move to the next step.

```
student@6312vm-csr00$ git checkout v2
```

   Else If you are starting from a fresh older, then you should use the `clone` command to download the v2 branch and move to the next step

```
student@6312vm-csr00$ git clone -b v2 https://github.com/inwk6312course/dockerapp.git
```

3. Build the Python web application

```
student@6312vm-csr00$ docker build -t dockerapp:v2 .
```

4. Run the web application and link with the Redis container

```
student@6312vm-csr00$ docker run -d -p 5000:5000 --link redis dockerapp:v2
```

5. Connect to the web application and inspect the hosts file to see how Docker registers the hostname of the running redis container with an IP address. This is how the web application is able to connect with the redis container using an IP address or the hostname

```
student@6312vm-csr00$ docker exec -it <<dockerApp-containerId>> bash
admin@4f734e68b935:/app$ more /etc/hosts
```

6. Ping the redis container to confirm connectivity

```
admin@4f734e68b935$ ping redis
```

Clean Up!

Stop and remove the containers by using the commands:

Look for the id of the container to stop:

```
docker container ls
```

To stop a container gracefully:

```
docker container stop <<container-id>>
```

To delete a container gracefully:

```
docker container rm <<container-id>>
```

Task 8: Using Docker Compose

Manual linking containers and configuring services become impractical when the number of containers grows. This is where Docker Compose comes to the rescue. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a `YAML` file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. In summary:

1. Docker compose is a very handy tool to quickly get docker environment up and running.

2. Docker compose uses `yaml` files to store the configuration of all the containers, which removes the burden to maintain our scripts for docker orchestration.

1. Clone the Python web application.
   If you are already working from the cloned Git repository from the previous task, then you can use the `checkout` command to change to a different branch (v3) which contains the updated code and move to the next step.

   ```
   student@6312vm-csr00$ git checkout v3
   ```

   Else If you are starting from a fresh older, then you should use the `clone` command to download the v3 branch and move to the next step

   ```
   student@6312vm-csr00$ git clone -b v3 https://github.com/inwk6312course/dockerapp.git
   ```

2. Inspect the file "`docker-compose.yml`". By Default, docker-compose looks for this file to operate on. Then use `docker-compose` command to start up all the services.

   ```
   student@6312vm-csr00$ docker-compose up
   ```

3. Stop all the services by pressing "`Ctrl + c`". Then start up the services again in detached mode.

```
student@6312vm-csr00$ docker-compose up -d
```

Confirm the web application is running by using `curl localhost:5000` on the terminal

Use the following commands to interact with the running service as necessary

To checks the status of the containers managed by docker compose.

```
student@6312vm-csr00$ docker-compose ps
```

To outputs colored and aggregated logs for the compose managed containers.

```
student@6312vm-csr00$ docker-compose logs
```

Using dash f option outputs appended log when the log grows.

```
student@6312vm-csr00$ docker-compose logs -f
```

To outputs the logs of a specific container.

```
student@6312vm-csr00$ docker-compose logs <<container-name|id>>
```

To stops all the running containers without removing them.

```
student@6312vm-csr00$ docker-compose stop
```

To removes all the containers.

```
student@6312vm-csr00$ docker-compose rm
```

To stop containers and remove resources.

```
student@6312vm-csr00$ docker-compose down
```

To rebuild all the images.

```
student@6312vm-csr00$ docker-compose rm
```

Clean Up!

Stop all the service Docker Compose created by using the commands:

```
docker-compose down
```

Docker Networking

Docker includes support for networking containers through the use of network drivers. By default, Docker provides two network drivers for you, the bridge, and the overlay drivers. You can also write a network driver plugin so that you can create your own drivers but that is an advanced task. Some of the most import Docker network types are:

- Closed Network / None Network
- Bridge Network
- Host Network
- Overlay Network

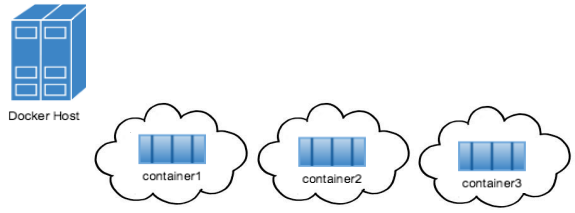
We can inspect the list of network types by using the command:

```
student@6312vm-csr00$ docker network ls
```

We can also inspect a particular network by using the command:

```
student@6312vm-csr00$ docker network inspect <<network-name>>
```

Task 8: Isolating Containers – using the None network



You may choose to move up one directory before continuing to the remaining tasks. There is no consequence.

```
student@6312vm-csr00$ cd ..
```

1. Start up a container in detached mode and specify the None network. Docker will isolate the container by not assigning any interface except a loopback. We will use an image called busybox, which is just a light weighted Linux image. The extra argument: `sleep 1000` is used to keep the container running.

```
student@6312vm-csr00$ docker run -d --net none busybox sleep 1000
```

2. Connect to the container and start the `ash` terminal service. Busybox doesn't have `bash` but `ash` terminal.

```
student@6312vm-csr00$ docker exec -it <<containerId>> /bin/ash
```

3. Ping an external network and confirm isolation.

```
/ # ping 8.8.8.8
```

4. Inspect the interfaces that was created in the container. Notice just the loopback interface

```
/ # ifconfig
```

Clean Up!

Stop and remove the containers by using the commands:

Look for the id of the container to stop:

```
docker container ls
```

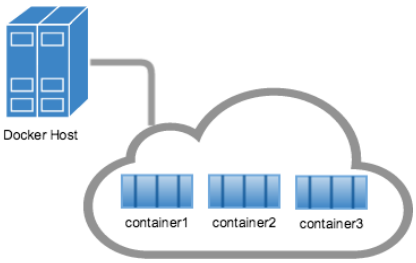
To stop a container gracefully:

```
docker container stop <<container-id>>
```

To delete a container gracefully:

```
docker container rm <<container-id>>
```

Task 9: Connecting Containers – using the Bridge network



In a Bridge network, containers have access to two (2) or more network interfaces.

- - A loopback interface
- - One or more Private interface

All containers in the same bridge network can communicate with each other. Containers from different networks can't connect with each other by default

1. Start up a container in detached mode and specify the Bridge network. Docker will place the container on the default bridge network and assign an IP address from the IP range of this bridge network.

```
student@6312vm-csr00$ docker run -d --name container1 busybox sleep 1000
```

2. Connect to container1, run `ifconfig`. Notice the appearance of the loopback and private interface. Note down the IP Address of interface `eth0`

```
student@6312vm-csr00$ docker exec -it container1 ifconfig
```

3. Start up another container in detached mode and specify the Bridge network

```
student@6312vm-csr00$ docker run -d --name container2 busybox sleep 1000
```

4. Connect to container2, run `ifconfig` and Note down the IP Address of interface `eth0`. Notice that container1 and container2 have address assigned on the same network. This means they should be able to communicate with each other

```
student@6312vm-csr00$ docker exec -it container2 ifconfig
```

5. Confirm connectivity by pinging `container2` from `container1`

```
student@6312vm-csr00$ docker exec -it container1 ping <ipAddress-eth0-container2>
```

6. Confirm external connectivity from container1 or container2

```
student@6312vm-csr00$ docker exec -it container1 ping 8.8.8.8
```

Task 10: Creating a Custom Bridge network

1. We use the argument: `create` and specifying the driver that should be used to create the new network. In this case, we are using the `bridge` driver

```
student@6312vm-csr00$ docker network create --driver bridge my_bridge_net
```

2. Confirm that the new network was created, then inspect the network to check the IP Address range that was assigned by Docker

```
student@6312vm-csr00$ docker network ls
student@6312vm-csr00$ docker network inspect my_bridge_net
```

3. Create a new container that uses the new network

```
student@6312vm-csr00$ docker run -d --name container3 --net my_bridge_net busybox sleep 1000
```

4. Connect to container3, run `ifconfig` and check the IP Address of interface `eth0`. Notice that the IP address assigned is from the IP range that you saw from step 2

```
student@6312vm-csr00$ docker exec -it container3 ifconfig
```

5. Confirm isolation of container3 from container1 that was bult in the previous task. They should not be able to communicate with each other because they reside on different networks

```
student@6312vm-csr00$ docker exec –it container3 ping <ipAddress-eth0-container1>
```

6. Delete the new network bridge.

```
student@6312vm-csr00$ docker network disconnect bridge container3
```

Task 11: Connecting a Container to an Existing Bridge/Network

Even though a container is already created, we can still attach it to an existing network. Docker will create a new private interface inside the container and assign an IP Address from the IP pool of that network.

7. Connect the default bridge to the container we created in the previous task.

```
student@6312vm-csr00$ docker network connect bridge container3
```

8. List the network interfaces of container3. We notice a new private interface has been created for us. Hence, we should be to reach containers that reside in that network

```
student@6312vm-csr00$ docker exec -it container3 ifconfig
```

9. Confirm connectivity by pinging container1 from container3

```
student@6312vm-csr00$ docker exec –it container3 ping <ipAddress-eth0-container1>
```

10. Disconnect the default bridge from container3.

```
student@6312vm-csr00$ docker network disconnect bridge container3
```

Task 12: Connecting Containers – using the Host network

The Host network is the least protected network model, it adds a container on the host's network stack.

Containers deployed on the host stack have full access to the host's interfaces and this kind of containers are usually called open containers. Containers on this network have the following characteristics:

- Minimum network security level.
- No isolation on this type of open containers, thus leave the container widely unprotected.
- Containers running in the host network stack should see a higher level of performance than those traversing the docker0 bridge and iptables port mappings.

1. Start up a new container and attach it to the host network.

```
student@6312vm-csr00$ docker run -d --name container4 --net host busybox sleep 1000
```

2. List the network interfaces of container4. Notice all the different interfaces it has. Docker gives this container access to all the different networks that reside on the host machine!

```
student@6312vm-csr00$ docker exec -it container4 ifconfig
```

Task 13: Define Container Networks with Docker Compose

In our finally task, we would show how to define networks using Docker Compose. We define the networks and allow Docker Compose to take of creating and managing these networks for us

1. Clone the Python web application.
   If you are already working from the cloned Git repository from the previous task, then you can use the `checkout` command to change to a different branch (v4) which contains the updated code and move to the next step.

```
student@6312vm-csr00$ git checkout v4
```

   Else If you are starting from a fresh older, then you should use the `clone` command to download the v3 branch and move to the next step

```
student@6312vm-csr00$ git clone -b v4 https://github.com/inwk6312course/dockerapp.git
```

2. Review the file "`docker-compose.yml`" already created for you

3. Start up the services using `docker-compose` command

```
student@6312vm-csr00$ docker-compose up
```

Clean Up!

Stop all the service Docker Compose created by using the commands:

```
docker-compose down
```

After you are finished with all tasks in this lab, delete all containers and images on your Linux machine by using the commands:

To delete all containers forcefully:

```
docker container rm -f $(docker container ls -a -q)
```

To delete all Images forcefully:

```
docker image rm -f $(docker image ls -a -q)
```

To remove all unused networks forcefully:

```
docker network prune -f
```

Appendix: Summary of Docker Commands

Commands	**Usage**
docker –version	to get the currently installed version of docker.
docker pull	to pull images from the docker repository(hub.docker.com).
docker run -it -d	to create a container from an image.
docker container ls / docker ps	to list the running containers.
docker container ls -a / docker ps -a	to show all the running and exited containers.
docker exec -it bash	to access the running container.
docker container stop <hash> / docker stop <hash>	Gracefully stops a running container
docker container kill <hash>	Forces shutdown of the specified container
docker container rm <hash>	Removes specified container from this machine
docker container rm $(docker container ls -a -q)	Removes all containers
docker container rm -f $(docker container ls -a -q)	Forcefully removes all containers
docker commit <username/imagename>	creates a new image of an edited container on the local system.
docker push <username/image name>	to push an image to the docker hub repository.
docker images / docker image ls	lists all the locally stored docker images.
docker image rm $(docker image ls -a -q)	Removes all Images
docker rm	is used to delete a stopped container.
docker rmi <image id> / docker image rm <image id>	Is used to delete an image from local storage.
docker build	is used to build an image from a specified docker file.
docker build -t friendlyname .	Is used to create image using this directory's Dockerfile
docker run -p 4000:80 friendlyname	Run "friendlyname" mapping port 4000 to 80
docker run -d -p 4000:80 friendlyname	Same thing, but in detached mode
docker login	Log in CLI session using your Docker credential
docker tag <image> username/repository:tag	Tag <image> for upload to registry
docker push username/repository:tag	Upload tagged image to registry
docker run username/repository:tag	Run image from a registry