Department of Information Technology

Academic Year: 2019-20 **Semester: VIII**
Class / Branch: BE IT
Subject: DevOps Lab (DL)
Subject Lab Incharge: Prof. Vishal S. Badgujar

EXPERIMENT NO. 05

Aim: To Install and Configure Docker for creating Containers of different Operating System Images.

Steps:

Lab 1: Run your first container

Install Docker or use Play-With-Docker

Install Docker

If you don't want to install Docker, skip this part and go to the next section "Use Play-With-Docker."

1. Sign up for Docker Hub: https://hub.docker.com/signup
2. Navigate to https://hub.docker.com/?overlay=onboarding
3. Download *Docker Desktop for Mac* or *Docker Desktop for Windows*.
4. Install Docker Desktop.

Use Play-With-Docker

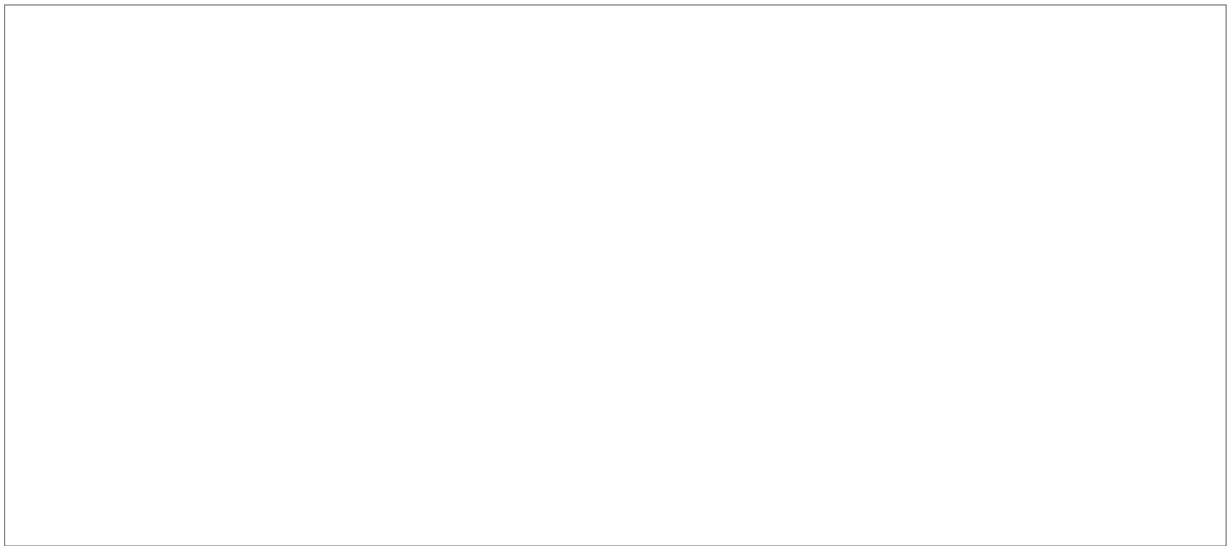
If you don't want to install Docker, an alternative is to use Play-With-Docker, which is a website where you can run terminals directly from your browser that have Docker installed.

Run a container

Use the **Run a container**
Use the Docker CLI to run your first container.
1.Open a terminal on your local computer and run this command:

```
& docker container run -t ubuntu top
```

You use the `docker container run` command to run a container with the Ubuntu image by using the `top` command. The `-t` flag allocates a pseudo-TTY, which you need for the `top` command to work correctly.



The `docker run` command first starts a `docker pull` to download the Ubuntu image onto your host. After it is downloaded, it will start the container. The output for the running container should look like this:

`top` is a Linux utility that prints the processes on a system and orders them by resource consumption. Notice that there is only a single process in this output: it is the `top` process itself. You don't see other processes from the host in this list because of the PID namespace isolation.

Containers use Linux namespaces to provide isolation of system resources from other containers or the host. The PID namespace provides isolation for process IDs. If you run `top` while inside the container, you will notice that it shows the processes within the PID namespace of the container, which is much different than what you can see if you ran `top` on the host.

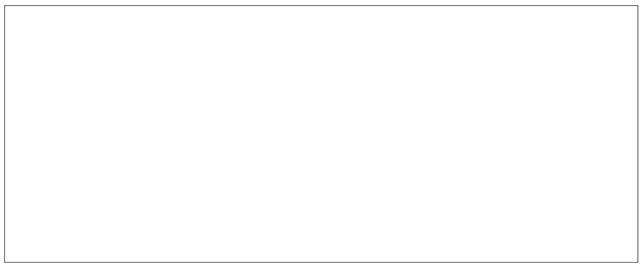
Even though we are using the Ubuntu image, it is important to note that the container does not have its own kernel. It uses the kernel of the host and the Ubuntu image is used only to provide the file system and tools available on an Ubuntu system.

2.Inspect the container:

`docker container exec`

This command allows you to enter a running container's namespaces with a new process.

3.Open a new terminal. To open a new terminal connected to node1 by using Play-With-Docker.com, click **Add New Instance** on the left and then ssh from node2 into node1 by using the IP that is listed by node1, for example:



4.In the new terminal, get the ID of the running container that you just created:
`docker container ls`

5.Use that container ID to run `bash` inside that container by using the `docker container exec` command. Because you are using bash and want to interact with this container from your terminal, use the `-it` flag to run using interactive mode while allocating a psuedo-terminal:

```
$ docker container exec -it b3ad2a23fab3 bash
root@b3ad2a23fab3:/#
```

You just used the `docker container exec` command to enter the container's namespaces with the bash process. Using `docker container exec` with `bash` is a common way to inspect a Docker container.

Notice the change in the prefix of your terminal, for example, `root@b3ad2a23fab3:/`. This is an indication that you are running bash inside the container.

Tip: This is not the same as using ssh to a separate host or a VM. You don't need an ssh server to connect with a bash process. Remember that containers use kernel-level features to achieve isolation and that containers run on top of the kernel. Your container is just a group of processes running in isolation on the same host, and you can use the command `docker container exec` to enter that isolation with the `bash` process. After you run the command `docker container exec`, the group of processes running in isolation (in other words, the container) includes `top` and `bash`.

6.From the same terminal, inspect the running processes:

```
$ ps -ef
```

You should see only the `top` process, `bash` process, and your `ps` process.

7.For comparison, exit the container and run `ps -ef` or `top` on the host. These commands will work on Linux or Mac. For Windows, you can inspect the running processes by using `tasklist`.

```
root@b3ad2a23fab3:/# exit
exit
$ ps -ef
# Lots of processes!
```

PID is just one of the Linux namespaces that provides containers with isolation to system resources. Other Linux namespaces include:

•MNT: Mount and unmount directories without affecting other namespaces.
•NET: Containers have their own network stack.
•IPC: Isolated interprocess communication mechanisms such as message queues.
•User: Isolated view of users on the system.
•UTC: Set hostname and domain name per container.

These namespaces provide the isolation for containers that allow them to run together securely and without conflict with other containers running on the same system.

In the next lab, you'll see different uses of containers and the benefit of isolation as you run multiple containers on the same host.

Tip: Namespaces are a feature of the Linux kernel. However, Docker allows you to run containers on Windows and Mac. The secret is that embedded in the Docker product is a Linux subsystem. Docker open-sourced this Linux subsystem to a new project: LinuxKit. Being able to run containers on many different platforms is one advantage of using the Docker tooling with containers.

In addition to running Linux containers on Windows by using a Linux subsystem, native Windows containers are now possible because of the creation of container primitives on the Windows operating system. Native Windows containers can be run on Windows 10 or Windows Server 2016 or later.

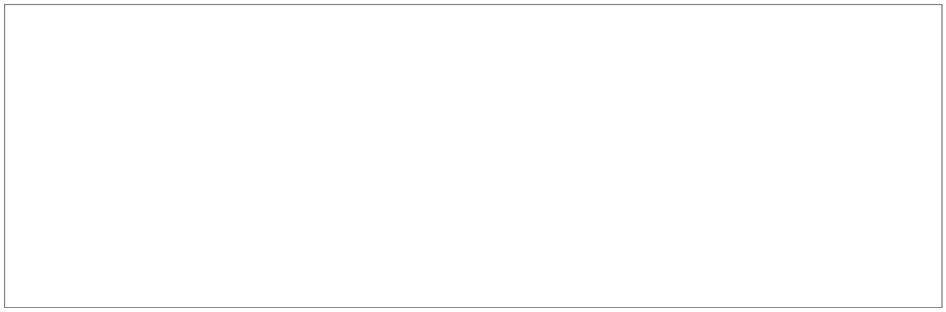
8.Clean up the container running the `top` processes:

```
<ctrl>-c
```

Tip: This is not the same as using ssh to a separate host or a VM. You don't need an ssh server to connect with a bash process. Remember that containers use kernel-level features to achieve isolation and that containers run on top of

the kernel. Your container is just a group of processes running in isolation on the same host, and you can use the command docker container exec to enter that isolation with the bash process. After you run the command docker container exec, the group of processes running in isolation (in other words, the container) includes top and bash.Docker CLI to run your first container.

1.From the same terminal, inspect the running processes:
$ ps -ef



You should see only the top process, bash process, and your ps process.

2.For comparison, exit the container and run ps -ef or top on the host. These commands will work on Linux or Mac. For Windows, you can inspect the running processes by using tasklist.
root@b3ad2a23fab3:/# exit
exit
$ ps -ef
# Lots of processes!

PID is just one of the Linux namespaces that provides containers with isolation to system resources. Other Linux namespaces include:
•MNT: Mount and unmount directories without affecting other namespaces.
•NET: Containers have their own network stack.
•IPC: Isolated interprocess communication mechanisms such as message queues.
•User: Isolated view of users on the system.
•UTC: Set hostname and domain name per container.

These namespaces provide the isolation for containers that allow them to run together securely and without conflict with other containers running on the same system.

In the next lab, you'll see different uses of containers and the benefit of isolation as you run multiple containers on the same host.

Tip: Namespaces are a feature of the Linux kernel. However, Docker allows you to run containers on Windows and Mac. The secret is that embedded in the Docker product is a Linux subsystem. Docker open-sourced this Linux subsystem to a new project: LinuxKit. Being able to run containers on many different platforms is one advantage of using the Docker tooling with containers.

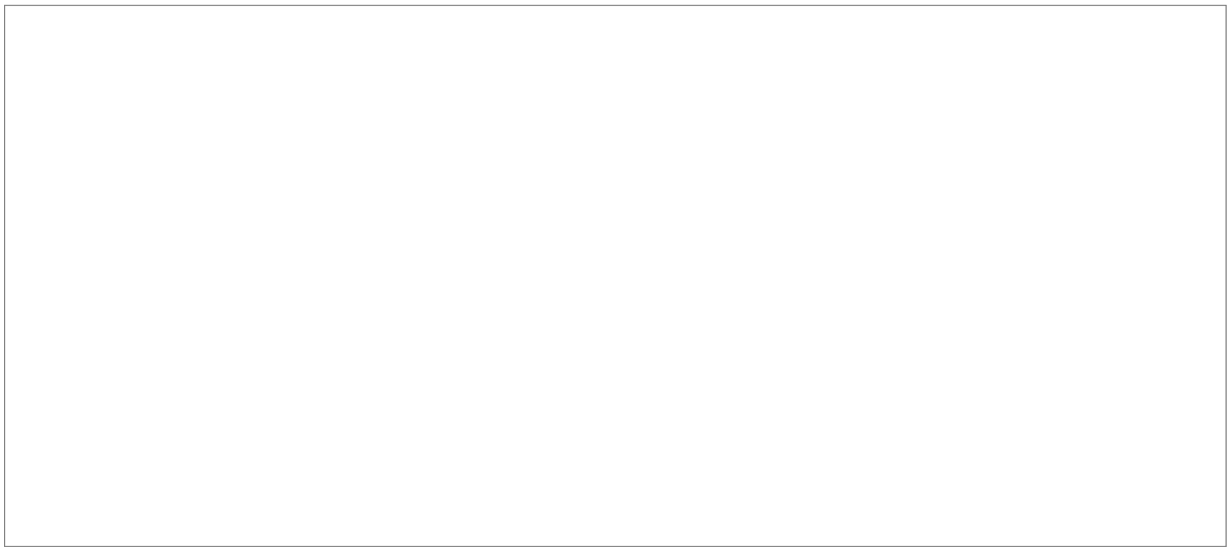
In addition to running Linux containers on Windows by using a Linux subsystem, native Windows containers are now possible because of the creation of container primitives on the Windows operating system. Native Windows containers can be run on Windows 10 or Windows Server 2016 or later.

3.Clean up the container running the `top` processes:
`<ctrl>-c`

4.Open a terminal on your local computer and run this command:

& docker container run -t ubuntu top

You use the `docker container run` command to run a container with the Ubuntu image by using the `top` command. The `-t` flag allocates a pseudo-TTY, which you need for the `top` command to work correctly.



The docker run command first starts a docker pull to download the Ubuntu image onto your host. After it is downloaded, it will start the container. The output for the running container should look like this:

top is a Linux utility that prints the processes on a system and orders them by resource consumption. Notice that there is only a single process in this output: it is the top process itself. You don't see other processes from the host in this list because of the PID namespace isolation.

Containers use Linux namespaces to provide isolation of system resources from other containers or the host. The PID namespace provides isolation for process IDs. If you run top while inside the container, you will notice that it shows the processes within the PID namespace of the container, which is much different than what you can see if you ran top on the host.

Even though we are using the Ubuntu image, it is important to note that the container does not have its own kernel. It uses the kernel of the host and the Ubuntu image is used only to provide the file system and tools available on an Ubuntu system.

5.Inspect the container:

docker container exec

This command allows you to enter a running container's namespaces with a new process.

6.Open a new terminal. To open a new terminal connected to node1 by using Play-With-Docker.com, click **Add New Instance** on the left and then ssh from node2 into node1 by using the IP that is listed by node1, for example:

7.In the new terminal, get the ID of the running container that you just created:

docker container ls

8.Use that container ID to run `bash` inside that container by using the `docker container exec` command. Because you are using bash and want to interact with this container from your terminal, use the `-it` flag to run using interactive mode while allocating a psuedo-terminal:

```
$ docker container exec -it b3ad2a23fab3 bash

root@b3ad2a23fab3:/#
```

You just used the `docker container exec` command to enter the container's namespaces with the bash process. Using `docker container exec` with `bash` is a common way to inspect a Docker container.

Notice the change in the prefix of your terminal, for example, `root@b3ad2a23fab3:/`. This is an indication that you are running bash inside the container.

Tip: This is not the same as using ssh to a separate host or a VM. You don't need an ssh server to connect with a bash process. Remember that containers use kernel-level features to achieve isolation and that containers run on top of the kernel. Your container is just a group of processes running in isolation on the same host, and you can use the command `docker container exec` to enter that isolation with the `bash` process. After you run the command `docker container exec`, the group of processes running in isolation (in other words, the container) includes `top` and `bash`.

1.From the same terminal, inspect the running processes:

```
$ ps -ef
```

```
root@b3ad2a23fab3:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 20:34 ?        00:00:00 top
root        17     0  0 21:06 ?        00:00:00 bash
root        27    17  0 21:14 ?        00:00:00 ps -ef
```

You should see only the `top` process, `bash` process, and your `ps` process.

2.For comparison, exit the container and run `ps -ef` or `top` on the host. These commands will work on Linux or Mac. For Windows, you can inspect the running processes by using `tasklist`.

```
root@b3ad2a23fab3:/# exit
exit
$ ps -ef
# Lots of processes!
```

PID is just one of the Linux namespaces that provides containers with isolation to system resources. Other Linux namespaces include:

- MNT: Mount and unmount directories without affecting other namespaces.
- NET: Containers have their own network stack.
- IPC: Isolated interprocess communication mechanisms such as message queues.
- User: Isolated view of users on the system.
- UTC: Set hostname and domain name per container.

These namespaces provide the isolation for containers that allow them to run together securely and without conflict with other containers running on the same system.

In the next lab, you'll see different uses of containers and the benefit of isolation as you run multiple containers on the same host.

Tip: Namespaces are a feature of the Linux kernel. However, Docker allows you to run containers on Windows and Mac. The secret is that embedded in the Docker product is a Linux subsystem. Docker open-sourced this Linux subsystem to a new project: LinuxKit. Being able to run containers on many different platforms is one advantage of using the Docker tooling with containers.

In addition to running Linux containers on Windows by using a Linux subsystem, native Windows containers are now possible because of the creation of container primitives on the Windows operating system. Native Windows containers can be run on Windows 10 or Windows Server 2016 or later.

3.Clean up the container running the `top` processes:
`<ctrl>-c`

2. Run multiple containers

1.Explore the Docker Store.

The Docker Store is the public central registry for Docker images. Anyone can share images here publicly. The Docker Store contains community and official images that can also be found on the Docker Hub.

When searching for images, you will find filters for Store and Community images. Store images include content that has been verified and scanned for security vulnerabilities by Docker. Go one step further and search for Certified images that are deemed enterprise-ready and are tested with Docker Enterprise Edition.

It is important to avoid using unverified content from the Docker Store when you develop your own images that are intended to be deployed into the production environment. These unverified images might contain security vulnerabilities or possibly even malicious software.

In the next step of this lab, you will start a couple of containers by using some verified images from the Docker Store: NGINX web server and Mongo database.

2.Run an NGINX server by using the official NGINX image from the Docker Store:

```
$ docker container run --detach --publish 8080:80 --name nginx nginx
```

You are using a couple of new flags here. The `--detach` flag will run this container in the background. The `publish` flag publishes port 80 in the container (the default port for NGINX) by using port 8080 on your host. Remember that the NET namespace gives processes of the container their own network stack. The `--publish` flag is a feature that can expose networking through the container onto the host.

How do you know port 80 is the default port for NGINX? Because it is listed in the documentation on the Docker Store. In general, the documentation for the verified images is very good, and you will want to refer to it when you run containers using those images.

You are also specifying the `--name` flag, which names the container. Every container has a name. If you don't specify one, Docker will randomly assign one for you. Specifying your own name makes it easier to run subsequent commands on your container because you can reference the name instead of the id of the container. For example, you can specify `docker container inspect nginx` instead of `docker container inspect 5e1`.

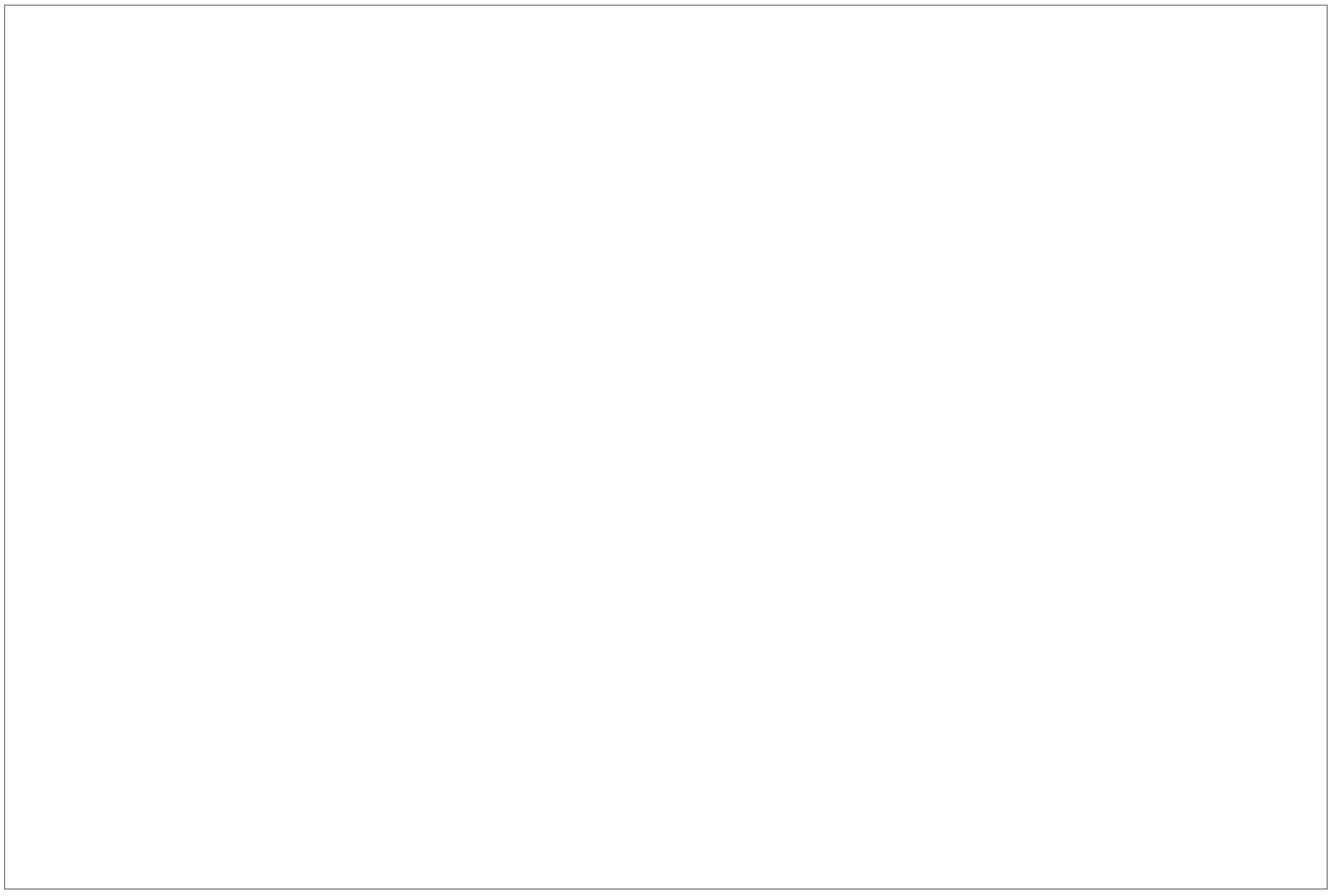
Because this is the first time you are running the NGINX container, it will pull down the NGINX image from the Docker Store. Subsequent containers created from the NGINX image will use the existing image located on your host.

NGINX is a lightweight web server. You can access it on port 8080 on your localhost.

3.Access the NGINX server on http://localhost:8080.

4.Run a MongoDB server. You will use the official MongoDB image from the Docker Store. Instead of using the `latest` tag (which is the default if no tag is specified), use a specific version of the Mongo image: 3.4.

```
$ docker container run --detach --publish 8081:27017 --name mongo mongo:3.4
```



Again, because this is the first time you are running a Mongo container, pull the Mongo image from the Docker Store. You use the `--publish` flag to expose the 27017 Mongo port on your host. You must use a port other than 8080 for the host mapping because that port is already exposed on your host. See the documentation on the Docker Store to get more information about using the Mongo image.

5.Access http://localhost:8081 to see some output from Mongo.

6.Check your running containers:

```
$ docker container ls
```

You should see that you have an NGINX web server container and a MongoDB container running on your host. Note that you have not configured these containers to talk to each other.

You can see the `nginx` and `mongo` names that you gave to the containers and the random name (in this example, `priceless_kepler`) that was generated for the Ubuntu container. You can also see that the port mappings that you specified with the `--publish` flag. For more information on these running containers, use the `docker container inspect [container id]` command.

One thing you might notice is that the Mongo container is running the `docker-entrypoint` command. This is the name of the executable that is run when the container is started. The Mongo image requires some prior configuration before kicking off the DB process. You can see exactly what the script does by looking at it on GitHub. Typically, you can find the link to the GitHub source from the image description page on the Docker Store website.

Containers are self-contained and isolated, which means you can avoid potential conflicts between containers with different system or runtime dependencies. For example, you can deploy an app that uses Java 7 and

another app that uses Java 8 on the same host. Or you can run multiple NGINX containers that all have port 80 as their default listening ports. (If you're exposing on the host by using the `--publish` flag, the ports selected for the host must be unique.) Isolation benefits are possible because of Linux namespaces.

Remember: You didn't have to install anything on your host (other than Docker) to run these processes! Each container includes the dependencies that it needs within the container, so you don't need to install anything on your host directly.

Running multiple containers on the same host gives us the ability to use the resources (CPU, memory, and so on) available on single host. This can result in huge cost savings for an enterprise.

Although running images directly from the Docker Store can be useful at times, it is more useful to create custom images and refer to official images as the starting point for these images. You'll learn to build your own custom images in the next lab.

3. Remove the containers

Completing this lab creates several running containers on your host. Now, you'll stop and remove those containers.

1.Get a list of the running containers:

```
$ docker container ls
```

2.Stop the containers by running this command for each container in the list:

```
$ docker container stop [container id]
```

You can also use the names of the containers that you specified before:

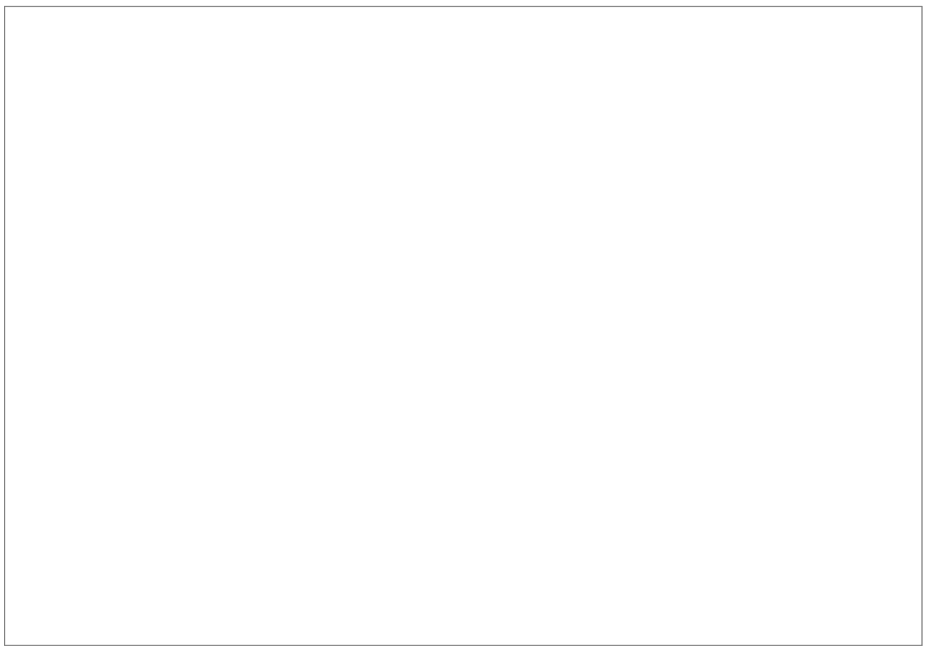
```
$ docker container stop d67 ead af5
d67
ead
```

af5

Tip: You need to enter only enough digits of the ID to be unique. Three digits is typically adequate.

3.Remove the stopped containers. The following command removes any stopped containers, unused volumes and networks, and dangling images:

```
$ docker system prune
```



Conclusion:

Write it your own Findings.