Docker Lab 2
Working with containers

1 REFERENCES AND CHEAT SHEETS....1
2 COMMANDS COVERED....1
3 LAB SETUP....4
4 RUNNING, STOPPING, STARTING AND LISTING CONTAINERS....4
5 REMOVING CONTAINERS....7
6 ATTACHING TO A RUNNING CONTAINER....7
7 INTERACTING WITH THE CONTAINER FILE SYSTEM....8
8 RETRIEVING CONTAINER LOGS....10
9 INSPECTING CONTAINERS....10
10 CREATING A CUSTOM IMAGE FROM A CONTAINER....11
11 RESTART POLICIES FOR CONTAINERS....13

1 References and cheat sheets

The official reference for all Docker commands:
https://docs.docker.com/reference/cli/docker/

https://dockerlabs.collabnix.com/docker/cheatsheet/
https://devopscycle.com/blog/the-ultimate-docker-cheat-sheet/
https://spacelift.io/blog/docker-commands-cheat-sheet
https://www.geeksforgeeks.org/docker-cheat-sheet/

2 Commands covered

Running, starting and stopping containers	Listing containers

`docker container run image-name`
 	
`docker container run -it image-name application-name`
`docker container run -d --name container-name image-name`
 	
https://linuxize.com/post/docker-run-command/
	`docker container ls`
	`docker container ls -a`
	`docker container ls --filter status=exited | running`
	`docker container ls --filter name=xxx`
	`docker container ls --filter id=xxx`

https://phoenixnap.com/kb/docker-run-command-with-examples
 	
`docker container stop container-name/id`
`docker container start container-name/id`
`docker container restart container-name/id`
 	
https://www.thegeekdiary.com/how-to-list-start-stop-delete-docker-containers/
 	
https://eldermoraes.com/docker-basics-how-to-start-and-stop-containers/
	https://linuxize.com/post/how-to-list-docker-containers/
	
	https://www.baeldung.com/ops/docker-list-containers

Attaching to a running container	**Interacting with the container file system**

`docker container attach container-name/id`
`docker container exec -it container-name/id shell-name`
 	
https://linuxize.com/post/how-to-connect-to-docker-container/
 	
https://www.baeldung.com/ops/docker-attach-detach-container
	`docker container cp local-file container-name:container-directory`
	`docker container cp container-name:container-file local-directory`
	
	https://www.cloudsavvyit.com/13987/how-to-use-docker-cp-to-copy-files-between-host-and-containers/
	
	https://www.baeldung.com/ops/docker-copying-files

Retrieving container logs	**Inspecting the container**

`docker container logs -f -t container-name/id`
`docker container logs --tail x container-name/id`
 	
https://geekflare.com/check-docker-logs/
https://sematext.com/guides/docker-logs/
	`docker container stats container-name/id`
	`docker container diff container-name/id`
	`docker container inspect container-name/id`
	`docker container inspect --format Go-template container-name/id`

	https://www.cloudsavvyit.com/13715/how-to-monitor-the-resource-usage-of-docker-containers/ https://semaphoreci.com/blog/container-diff-tutorial https://www.thegeekdiary.com/how-to-get-information-about-a-container-in-docker/ https://www.ctl.io/developers/blog/post/what-to-inspect-when-youre-inspecting

Creating an image from a container	**Restart policies for containers**
`docker container commit -m message -a author container-name/id new-image-name` https://adamtheautomator.com/docker-commit/ https://phoenixnap.com/kb/how-to-commit-changes-to-docker-image	`docker container run --restart always | unless-stopped | on-failure image-name` https://www.cloudsavvyit.com/10912/how-to-use-docker-restart-policies-to-keep-containers-running/

Removing containers	

`docker container rm container-name/id`
`docker container rm -f container-name/id`
`docker container rm -f $(docker container ls -a -q)`
`docker container prune`
 	
https://linuxize.com/post/how-to-remove-docker-images-containers-volumes-and-networks/
 	
https://phoenixnap.com/kb/remove-docker-images-containers-networks-volumes
 	
https://stackoverflow.com/questions/63671792/what-is-the-difference-between-docker-container-prune-vs-docker-rm-docker-c	

3 Lab setup

This is identical to the previous lab

4 Running, stopping, starting and listing containers

Type:

```
docker run hello-world
```

This attempts to start up a container from an image called `hello-world`. Notice that Docker informs us that it is not able to find the image in the local registry, and it then makes an attempt to locate and pull this image down from the default registry (Docker Hub). Since we did not specify a tag for this image, Docker automatically appends the tag `latest`.
The official page for this image is at:
https://hub.docker.com/_/hello-world

The container for this very basic image simply displays some output message from its execution and exits immediately to the command line. It exists simply to ensure that Docker is running properly on the local host machine.

You can verify that the image has been loaded to the local registry with:

```
docker images
```

You should also be able to view it in the Docker Desktop main pane.

To see the list of running containers, type:

```
docker ps
```

Nothing is listed because the container has already completed executing. To see the list of all containers, including those that have stopped, type:

```
docker ps -a
```

IMPORTANT NOTE: All command options are either one letter preceded by a -, or a complete word preceded by two dashes --. You can check this out with `docker ps --help`
Thus, the option `-a` is exactly same as `--all`
However there are some options for all commands which do not have an equivalent one letter shortcut. For example, for the case of docker ps, we must specify the options `--format` and `--no-trunc`.

We can see the container ID (which is conceptually similar to the image ID and is used to uniquely identify each running container), the image used to create it, the command it last ran when it was created, its exit status (0 - for a normal exit), ports and names. If we run a container without specifying a name, Docker automatically generates a random name for us.

Let's pull the latest `alpine` image from Docker Hub.

```
docker pull alpine
```

Next, lets run a container from this downloaded image and open an interactive terminal shell in the running container:

```
docker run -it alpine /bin/sh
```

The -i flag keeps STDIN open from the container, even if we're not attached to it. The -t flag tells Docker to assign a pseudo-tty to the container we're about to create. This provides us with an interactive shell in the new container. The `/bin/sh` is a reference to the system shell. For the `alpine` container, this will be the `ash` shell that is typically used in embedded Linux systems: it is a lighter weight version of the standard bash shell found in Ubuntu and Debian.

Check the contents of the current directory with the standard Linux command

```
/ # ls -la

total 64
drwxr-xr-x    1 root     root          4096 Jul 14 08:25 .
drwxr-xr-x    1 root     root          4096 Jul 14 08:25 ..
-rwxr-xr-x    1 root     root             0 Jul 14 08:25 .dockerenv
drwxr-xr-x    2 root     root          4096 Jun 15 14:34 bin
drwxr-xr-x    5 root     root           360 Jul 14 08:25 dev
drwxr-xr-x    1 root     root          4096 Jul 14 08:25 etc
drwxr-xr-x    2 root     root          4096 Jun 15 14:34 home
…….
```

AS you can see, it's the typical layout of the root directory of a Linux file volume. Check the account you are logged in under as well as the hostname of this container.

```
/home # whoami
root

/home # hostname
99e8416f67c6
```

If you start up another Powershell terminal, and type

```
docker ps
```

you should be able to see the running container listed with the hostname being the same as the container ID.

In the other PowerShell window, start up another container from the same image but with an explicitly assigned name (using the `--name` flag) and run it in detached mode as daemonized process (using the `-d` flag)

```
docker run -d --name superman -it alpine
```

This command exits successfully with the container ID of the newly created container.

Keep in mind that names are unique. If we try to create two containers with the same name, the command will fail. We need to delete the previous container with the same name before we can create a new one.

If you check the list of running containers again, you should see this container with our specified name.

```
docker ps
```

You can start as many containers as you want from a single image in this manner. All these containers will be based from the same image with their own individual read-write layers on the top to hold state changes related to changes in the file system of their respective containers.

In the Linux shell of the other container, exit by typing:

```
/ # exit
```

When you started that container, the system shell (`/bin/sh`) was the one and only process running inside the container. The moment you exit or kill this main process, the container will stop running as well.

Check for this with:

```
docker ps -a
```

You can start this stopped container with:

```
docker start container-name/id
```

When you start a stopped container, it automatically starts in detached mode, so you will not have an interactive terminal inside it although it remains running in the background.

You can stop the running container with:

```
docker stop container-name/id
```

You can also restart a running container (stop followed immediately by start) with:

```
docker restart container-name/id
```

We can use the `--filter` option to further filter on the list of containers returned. For e.g.

```
docker ps --filter status=exited | running
docker ps --filter name=xxx (where xxx is a substring of the full name)
docker ps --filter id=  (where xxx is a subsequence of the id)
```

Try to create a few containers from the `alpine` image to test out these commands.

5 Removing containers

Stopping containers does not remove them. If you wish to remove a container permanently from the host machine, you will first have to stop them.

Identify a running container (start one if there is none running at the moment), and attempt to remove it with:

```
docker rm container-name/id
```

Notice that you will get an error message. Identify a stopped container (stop one if all are running) and then attempt to remove it with the same command: this time you should succeed.

You can add the -f flag to force the remove of a running container. Identify another running container (start one if there is none running at the moment), and attempt to remove it with:

```
docker rm -f container-name/id
```

This time you should succeed.

Start up some containers and leave a few others stopped. To remove all stopped containers, you can type:

```
docker container prune
```

Stop some of the running containers and leave others running. To remove all containers (both running and stopped), type:

```
docker rm -f $(docker ps -a -q)
```

The first part of this command `(docker container ls -a -q)` obtains the container IDs of all the containers and then passes is to the second command (`docker container rm)` which has a -f flag to force removal regardless of whether the container is running or stopped.

6 Attaching to a running container

You can attach to an interactive shell within a detached running container in daemonized mode that supports this.

Start a new container from the Alpine image in detached mode.

```
docker run -d --name superman -it alpine
```

You can now attach to it with:

```
docker attach container-name/id
```

You can exit the container without terminating the main process (the shell) by typing Ctrl-PQ. In this case, the container still remains detached and running in the background (check with `docker ps`)

Another way to attach to it is to execute the shell process in it:

```
docker exec -it container-name/id /bin/sh
```

If you do this, and then you check for running processes within the shell, you will notice that there are two shells now running:

```
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   30 root      0:00 /bin/sh
   39 root      0:00 ps
```

This is because the exec command creates a new shell process, rather than attaching to the existing shell process in the container. This means that if you exit the shell process now (by typing `exit` at the prompt), the container still remains running because there is still a single shell process still left running in it. Again, check this with `docker ps`

7 Interacting with the container file system

Start and open an interactive shell inside the `superman` container (start a new container from the Alpine image with this name if you don't have one yet)

```
docker start superman

docker attach superman
```

Create a new directory and store a simple file in it with these commands and then exit:

```
/ # cd home
/home # mkdir mystuff
/home # cd mystuff
/home/mystuff # echo "Hello world" > message.txt
/home/mystuff # cat message.txt
Hello world
/home/mystuff # exit
```

Start the container again and attach to it in a shell terminal.

```
docker start superman

docker attach superman
```

Verify that the file and directory you created there has still persisted:

```
/ # cd home/mystuff
/home/mystuff # cat message.txt
Hello world
/home/mystuff # exit
```

This demonstrates that any state changes made to a container (i.e. changes to its file system) between successive stops and starts are preserved. However, once a container is removed and then restarted

from an image, all state changes are permanently lost. The only way to preserve them is through volumes and bind mounts (which we will be examining in an upcoming lesson).

In any empty directory on your host machine file system, create a file: `things.txt` and populate it with random content. Switch to this directory in a PowerShell terminal, and type this command to copy this file into the `/home/mystuff` directory in the `superman` container.

```
docker cp things.txt superman:/home/mystuff
```

Notice that the command above works regardless of whether the container is running or has stopped.

Start the container again and attach to it in a shell terminal.

```
docker start

docker attach superman
```

Verify that you have transferred this file from the host file system to the container file system successfully:

```
/ # cd home/mystuff
/home/mystuff # ls -l
total 8
-rw-r--r--    1 root     root            12 Jul 14 09:31 message.txt
-rwxr-xr-x    1 root     root            17 Jul 14 11:18 things.txt
/home/mystuff # cat things.txt
my things to do
/home/mystuff # exit
```

In a Powershell terminal, transfer the `message.txt` file that you created earlier within the container to any directory on your host machine file system (replace *`local-directory`* appropriately, for e.g. `D:\data`). NOTE: If your Windows file path contains spaces, ensure that you enclose it within double quotes so that Docker interprets it correctly as single argument, for e.g.: `"C:\Users\James Bond\Desktop\cool stuff"`

```
docker cp superman:/home/mystuff/message.txt local-directory
```

Verify that the file has been transferred successfully using Windows explorer.

Lastly, start another container with any random name from the same image that you started the `superman` container from:

```
docker run --name batman -it alpine
```

In the shell, navigate to the home directory and check its contents:

```
/ # cd home
/home # ls -l
total 0
/home # exit
```

This clearly demonstrates that although these 2 containers are started from the same image (`alpine`), their state of their file system is unique to each of them and is maintained in the top level

read-write layer that builds on top of the lower level read-only base image layers that are common to both containers.

8 Retrieving container logs

Here, we simulate the situation of continuous console output from a process inside a container by running a container with a script that runs in a endless loop. In a real life app (which we will see later), there may be continuous console output from a process such as web server application running within a container and we may wish to inspect this console output.

We start another container with a script that runs endlessly and outputs a console output statement every 1 second:

```
docker run --name spiderman -d ubuntu /bin/bash -c 'count=1; while true; do echo Counting $count; ((count++)); sleep 1; done'
```

To get a log of all the console output to the current point of time:

```
docker logs spiderman
```

Issue the command above after a random period of time to see how the console log output has changed.

We can also follow the log output dynamically (-f) with timestamps added in (-t)

```
docker logs -f -t spiderman
```

Press Ctrl-C to exit

To see the last 5 console output statements with timestamps added:

```
docker logs --tail 5 -t spiderman
```

9 Inspecting containers

To list all the active processes inside a running container

```
docker top spiderman
```

Here we should be able to see one bash shell process running the script that we started the container with earlier

To get a live stream of container resource usage statistics: CPU, memory usage, network I/O performance, we can run:

```
docker stats spiderman
```

Press Ctrl-C to exit.

To inspect changes to a container's file system from the time it was started, we can use the superman container as an example where earlier we had made a number of changes to the file system:

```
docker diff superman
```

The output should look similar to the following:

```
C /home
A /home/mystuff
A /home/mystuff/message.txt
A /home/mystuff/things.txt
C /root
A /root/.ash_history
```

A, C, R in the output above stands for added / changed / removed

To obtain information on configuration details, commands, networking configuration, etc, we can type:

```
docker inspect superman
```

To drill down into the detailed list of information output, we can use a Go template together with the `--format` option. For example:

```
docker inspect --format '{{ .State.Running }}' superman

docker inspect --format '{{ .NetworkSettings.IPAddress }}' spiderman
```

10 Creating a custom image from a container

One operation that is may be performed when working with containers is to create a custom image from a running container after we have made some changes to it (for e.g. installing new applications, created new files, etc). The custom image essentially provides a way for us to persist all these changes made in a container that started from a base image (for e.g. `alpine, ubuntu`, etc) so that we can start a new container with exactly the same changes in place in the future without needing to repeat all the steps to create those changes.

We can use the `superman` container from a previous session for this purpose (where we had already made some minor changes to its file system).

We can create an image (with an appropriate `repo:tag` format) from this container with a message and author info using:

```
docker commit  -m "A new custom image" -a "Clark Kent" superman new-alpine:v1
```

Check that the custom image was indeed generated with:

```
docker images
```

Check the contents of the newly generated image with:

```
docker inspect new-alpine:v1
```

To check the message and author info that you provided earlier, you can again drill down with `--format` using a Go template:

```
docker inspect  --format '{{.Author}} created :-  {{.Comment}}' new-alpine:v1
```

If you check the file system layers on the original `alpine` image:

```
docker inspect --format '{{.RootFS.Layers}}' alpine
```

and compare that with the new image we just created:

```
docker inspect  --format '{{.RootFS.Layers}}' new-alpine:v1
```

you will see the new image has an additional layer on top of the common base image shared between these two images. This additional layer represents all the file system changes we made to the `superman` container in an earlier lab session.

Lets now generate a new container from this custom image that we just created:

```
docker run -d --name ironman -it new-alpine:v1
```

Attach to this new container and check its file system contents:

```
docker attach ironman
```

```
/ # cd home
/home # ls -la
total 12
drwxr-xr-x    1 root     root          4096 Aug  2 23:56 .
drwxr-xr-x    1 root     root          4096 Aug  3 00:27 ..
drwxr-xr-x    2 root     root          4096 Aug  3 00:19 mystuff
/home # cd mystuff
/home/mystuff # ls -la
total 16
drwxr-xr-x    2 root     root          4096 Aug  3 00:19 .
drwxr-xr-x    1 root     root          4096 Aug  2 23:56 ..
-rw-r--r--    1 root     root            12 Aug  2 23:56 message.txt
-rwxr-xr-x    1 root     root            23 Aug  2 22:30 things.txt
/home/mystuff # exit
```

Notice that we have exactly all the file changes preserved from the superman container at the time when we generated the custom image. As mentioned earlier, the custom image essentially provides a way for us to persist all these changes made in a container that started from a base image (for e.g. `alpine`, `ubuntu`, etc) so that we can start a new container with exactly the same changes in place in the future without needing to repeat all the steps to create those changes.

Besides this approach, the most common way to create a custom image (from which we can later generate containers from) is to use a Dockerfile to specify the explicit steps involved in the creation of that custom image. We will be covering this in an upcoming lab.

We can now optionally tag this image to make it suitable to push to our DockerHub account to share with others (who can then pull it to their local registries) or to be used in a CI / CD workflow that runs on a different server.

```
docker tag new-alpine:v1 dockerhub-username/new-alpine:v1
```

```
docker push dockerhub-username/new-alpine:v1
```

Check your DockerHub account for this new repo and image.

Practice pulling this new image from your DockerHub account to your local registry and generating a container from it (delete the existing image from your local registry first)

11 Restart policies for containers

It's often a good idea to run containers with a restart policy. This is a form of self-healing that enables Docker to automatically restart them after certain events or failures have occurred. Restart policies are applied per-container, and can be configured imperatively on the command line as part of `docker-container run` commands, or declaratively in YAML files for use with higher-level tools such as Docker Swarm, Docker Compose, and Kubernetes.

These are the following restart policies:

- `always` policy - It always restarts a stopped container unless it has been explicitly stopped, such as via a `docker stop` command. However, if it has been stopped this way and the Docker engine (daemon) is restarted, the container will be automatically restarted.
- `unless-stopped` policy - similar to `always`, except it will not be restarted when the Docker engine restarts if they were in the stopped / exited state.
- `on-failure` policy - will restart a container if it exits with a non-zero exit code (i.e. a failure occurred in its single running process rather than a normal exit). It will also restart containers when the Docker daemon restarts, even containers that were in the stopped state.

Start a new container from the `alpine` image in the normal way with:

```
docker run --name normal -it alpine /bin/sh
```

Exit the shell, and check for all containers with:

```
docker ps -a
```

As expected, this container has stopped running as we exited the single sole process (the shell) running in that container.

Start a new container from the `alpine` image, but this time with the restart flag included:

```
docker run --name always-going -it --restart always alpine /bin/sh
```

Wait for a couple of seconds, and then exit the shell. Remember that by right existing this shell (which is the sole running process inside this container) should cause the container to stop.

However, when you check the list of running containers (with `docker ps`), we still see that the container is up and running. This is due to the restart policy that we instated for it.

The `RestartCount` variable in the container configuration keeps track of the number of times the container has been restarted. Check for it with:

```
docker inspect --format '{{ .RestartCount }}' always-going
```

Let's attach to the single shell running in the container when it was restarted, and then exit from the shell again.

```
docker attach always-going
```

```
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    7 root      0:00 ps
/ # exit
```

Once again, checking the list of running containers (with `docker ps`) shows the container is up and running.

Checking the `RestartCount` variable in the container configuration again shows it has been incremented:

```
docker inspect --format '{{ .RestartCount }}' always-going
```

If you now stop the container with:

```
docker stop always-going
```

and check again, you will now see it has permanently stopped.

Start 2 containers with a single idle process sleeping in the background with 2 different restart policies:

```
docker run -d --name always --restart always alpine sleep 1d
```

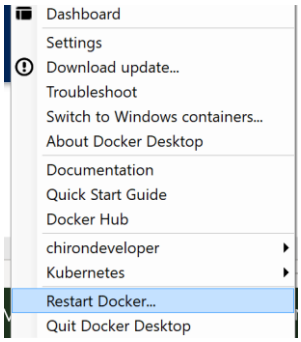
```
docker run -d --name unless-stopped --restart unless-stopped alpine
sleep 1d
```

Explicitly stop both containers:

```
docker stop always unless-stopped
```

Verify that they are stopped by listing the running containers.

Next, restart the Docker engine. If you are using Docker Desktop, you can use the option from the Docker icon at the bottom right hand corner:



When Docker restarts, check again the list of running containers. Notice that only the containers that you started earlier with the `--restart always` policy are still running (this would be the `always` and `always-going` containers)

When you are complete with this lab, you can stop these containers and remove all existing images and containers to prepare for the upcoming lab sessions:

```
docker container prune

docker image prune --all
```