CLI

Contents

Using `--help` Flag:

- Running `docker --help` displays a list of top-level commands and general usage information.
- Example: Running `docker network --help` provides details about the `docker network` command and its sub-commands.
    - Sub-commands: `connect`, `create`, `disconnect`.
- To explore a sub-command, use the `--help` flag again:
    - Example: `docker network create --help`.
    - This shows:
        - **Usage:** A one-line description of the command.
        - **Options:** All supported flags for the command.

1. Creating a Docker Container

Overview of Docker Containers:

- Containers are created from container images.
- Images are pre-packaged file systems containing the app, its environment, and a start instruction (entry point).

Steps to Create a Docker Container (Long Way):

- Use `docker container create` to create a container from an image.
    - If the image doesn't exist locally, Docker **pulls** it from Docker Hub by default.
    - Example: `docker container create hello-world` creates a container from the `hello-world` image.
    - Adding a tag: Use `:` followed by the tag (e.g., `hello-world:linux`).
- Verify that the image was pulled successfully (look for `pull complete` messages).
- Docker assigns a unique `ID` to each container.

```
$ docker container create hello-world:linux
Unable to find image 'hello-world:linux' locally
linux: Pulling from library/hello-world
c1ec31eb5944: Download complete
Digest: sha256:b7d87b72c676fe7b704572ebdfdf080f112f7a4c68fb77055d475e42ebc3686f
Status: Downloaded newer image for hello-world:linux
5ebb71abc898f315ddd51d75400bb2a3833a96545bc57dc6ce69826a1ad5f625
```

Starting the Container:

- `docker container create` does not start the container.
- Run `docker ps` to list running containers.
- To see all containers (including stopped ones), use `docker ps --all`.
- Start the container with `docker container start <container-ID>`.
- Example: Copy the container ID (or use the first few characters) and run `docker container start 5ebb`.

```
$ docker container start 5ebb
5ebb

write@Kalarava MINGW64 ~
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

write@Kalarava MINGW64 ~
$ docker ps -all
CONTAINER ID   IMAGE               COMMAND    CREATED              STATUS                      PORTS     NAMES
5ebb71abc898   hello-world:linux   "/hello"   About a minute ago   Exited (0) 13 seconds ago             relaxed_lumiere
```

Viewing Logs:

- Check logs with `docker logs <container-ID>`.
    - Example: `docker logs 5ebb` displays the friendly message from the `hello-world` container.
- Use logs for troubleshooting containers that fail.

```
write@Kalarava MINGW64 ~
$ docker logs 5ebb

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
```

Attaching to the Container's Output:

- Use `docker container start --attach <container-ID>` to start the container and immediately view its output.
- For long-running containers, use `docker container attach` after starting the container.

```
write@Kalarava MINGW64 ~
$ docker container start --attach 5ebb

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Key Points:

- Containers are not deleted automatically after creation or execution.
- The same container can be started multiple times without recreating it.

Create Docker Container: The Short Way

Run a container using the `docker run` command:

```
docker run hello-world:linux
```

- This command:
  - Creates a container from the `hello-world:linux` image.
  - Starts the container.
  - Attaches to the container to display its output.

```
write@Kalarava MINGW64 ~
$ docker run hello-world:linux

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
```

- So,
- `docker run =` docker container create
  + docker container start
  + docker container attach

2. Creating Docker Containers from Dockerfiles

Introduction

- So far, we have been creating containers from pre-existing Docker images available on Docker Hub.
- To create a container for our own application, we need to build a custom Docker image.
- Docker provides tools to:
  - Build custom images using Dockerfiles.
  - Start containers from those images.
  - Optionally, push the images to Docker Hub to share with others.

Building a Custom Docker Image

Files Required:

- **Dockerfile**: A file containing instructions to build the Docker image.
- **Example App**: A C++ program which prints `'Hello, World!'` on the screen.

```
$ ls -ltr ./DockerPractice/
total 2
-rw-r--r-- 1 write 197609 135 Jan 13 12:43 FileName.cpp
-rw-r--r-- 1 write 197609 460 Jan 13 12:45 Dockerfile
```

Dockerfile Syntax Overview

- A Dockerfile is a plain text file with a specific syntax and set of instructions to define how to build an image.

Key Dockerfile Instructions:

- `FROM`:
  - Specifies the base image for the custom image.

    - The base image can be:
        - A local image.
        - A remote image from Docker Hub (default behavior if not local).

- `LABEL`:
    - Adds metadata to the image (e.g., maintainer information).
- `COPY`:
    - Copies files from the host machine (build context) to the container image.
    - The "context" is typically the directory passed to the `docker build` command (default: current working directory).
- `RUN`:
    - Executes commands during the image build process.
    - Used to install additional software or configure the environment for the application.
    - Example: Installing `g++` for the program to build and to run.
- `ENTRYPOINT`:
    - Defines the default command or script executed by the container when it starts.
    - Alternative: `CMD`, which also defines a default command, but with different behavior.

Example Explanation:

- In the provided Dockerfile:
    - The `FROM` instruction specifies the base image.
    - The `RUN` instructions `g++` to compile `C++` program.
- The `CMD` instruction specifies the `hello` object file as the default executable when the container runs.

```
1  # Use an official base image
2  FROM ubuntu
3
4  LABEL maintainer="nishithjain <write2nishi@gmail.com>"
5
6  # Install required packages for C++ compilation
7  RUN apt-get update && apt-get install -y \
8      g++ \
9      && apt-get clean
10
11 # Set the working directory in the container
12 WORKDIR /app
13
14 # Copy the C++ source file to the container
15 COPY FileName.cpp .
16
17 # Compile the C++ source file
18 RUN g++ -o hello FileName.cpp
19
20 # Run the compiled C++ executable
21 CMD ["./hello"]
```

- Build the custom image using the `docker build` command.

```
write@Kalarava MINGW64 ~
$ docker build -t my-first-image ./DockerPractice
[+] Building 91.4s (11/11) FINISHED                                                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                        0.0s
 => => transferring dockerfile: 495B                                                                        0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest                                            2.9s
 => [auth] library/ubuntu:pull token for registry-1.docker.io                                               0.0s
 => [internal] load .dockerignore                                                                           0.0s
 => => transferring context: 2B                                                                             0.0s
 => [1/5] FROM docker.io/library/ubuntu:latest@sha256:80dd3c3b9c6cecb9f1667e9290b3bc61b78c2678c02cbdae5f0fea92cc6734ab   3.0s
 => => resolve docker.io/library/ubuntu:latest@sha256:80dd3c3b9c6cecb9f1667e9290b3bc61b78c2678c02cbdae5f0fea92cc6734ab   0.0s
 => => sha256:de44b265507ae44b212defcb50694d666f136b35c1090d9709068bc861bb2d64 29.75MB / 29.75MB            2.4s
 => => exporting attestation manifest sha256:f9befded1c64f57c99a90bda69c23e2b2defa594f5a6b7d1b04c4cba093c0a34   0.0s
 => => exporting manifest list sha256:125ae39d5a776ef89a209c6fbc2b2f3f84d72d25bf5b192f27191ffae479a668      0.0s
 => => naming to docker.io/library/my-first-image:latest                                                    0.0s
 => => unpacking to docker.io/library/my-first-image:latest                                                 1.5s
```

`docker build -t my-first-image ./DockerPractice`

Command Breakdown

`docker build:`

- This is the base command used to build a Docker image from a Dockerfile.

`-t my-first-image:`

- The `-t` flag stands for "`tag`". It assigns a name (or tag) to the resulting image.
- In this case, the image will be named `my-first-image`.

`./DockerPractice:`

- This is the path to the directory containing the Dockerfile.
- The `.` (dot) at the beginning signifies the current directory. If your Dockerfile is located in a subdirectory named `DockerPractice`, this tells Docker to use that directory as the build context.

Run the container

- Run the created image file using `docker run` command.

```
write@Kalarava MINGW64 ~
$ docker run my-first-image
Hello, World!
This is a test file.
```

3. Interacting with Running Containers

- The `docker run <image-name>` starts the container and attaches to the terminal by default.

Handling attached containers:

- If the **terminal hangs**:
  - Open a new terminal.
  - Use `docker ps` to find the container `ID`.
  - Stop the container with `docker kill <container-id>`.

Running commands inside a running container:

- `docker exec` allows additional commands to be run:
  - Example: `docker exec <container-id> date` (runs `date` command).
  - Example: `docker exec --interactive --tty <container-id> bash` (starts an interactive bash shell inside the container).

- Example:
  - We have a C++ program the prints '`Hello, World!`' on the terminal infinitely…
  - If we run the container, the program continuously prints the string on the screen.
  - To open an interactive terminal, use…

```cpp
#include <iostream>

int main(void) {

   while (true) {
      std::cout << "Hello, World!\n";
   }
   return 0;
}
```

```
docker exec --interactive --tty 025e bash
```

```
write@Kalarava MINGW64 ~
$ docker exec --interactive --tty 025e bash
root@025e1b206032:/app# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  3 13:04 ?        00:00:09 ./hello
root          19     0  0 13:09 pts/0    00:00:00 bash
root          27    19  0 13:09 pts/0    00:00:00 ps -ef
```

Stop the Container

- To stop this running container,
    - Get the container id using `docker ps -all` command...
    - Use the `docker stop <container-id>` command to stop the container.

```
write@Kalarava MINGW64 ~
$ docker ps -all
CONTAINER ID   IMAGE            COMMAND     CREATED              STATUS              PORTS     NAMES
f4f5313b6f0f   my-first-image   "./hello"   About a minute ago   Up About a minute             practical_chand
rasekhar

write@Kalarava MINGW64 ~
$ docker stop f4f5
f4f5

write@Kalarava MINGW64 ~
$ docker ps -all
CONTAINER ID   IMAGE            COMMAND     CREATED              STATUS                          PORTS     NAMES
f4f5313b6f0f   my-first-image   "./hello"   About a minute ago   Exited (137) 16 seconds ago               pract
ical_chandrasekhar
```

- The `docker stop <container-id>` command takes almost `10-15` seconds to stop the container. If we want to stop it immediately, we can give `-t` option.

```
write@Kalarava MINGW64 ~
$ docker ps
CONTAINER ID   IMAGE            COMMAND     CREATED          STATUS          PORTS     NAMES
db545c70359f   my-first-image   "./hello"   19 minutes ago   Up 19 minutes             jolly_heisenberg
cb2058a2ca0c   my-first-image   "./hello"   22 minutes ago   Up 22 minutes             nostalgic_carver

write@Kalarava MINGW64 ~
$ docker stop -t 0 db54
db54

write@Kalarava MINGW64 ~
$ docker ps
CONTAINER ID   IMAGE            COMMAND     CREATED          STATUS          PORTS     NAMES
cb2058a2ca0c   my-first-image   "./hello"   22 minutes ago   Up 22 minutes             nostalgic_carver
```

Immediately stops the container

4. Removing Containers

Remove a Stopped Container:

- Use `docker rm <container_id>` to delete a stopped container.
- docker rm will not remove running containers; they must be stopped first.

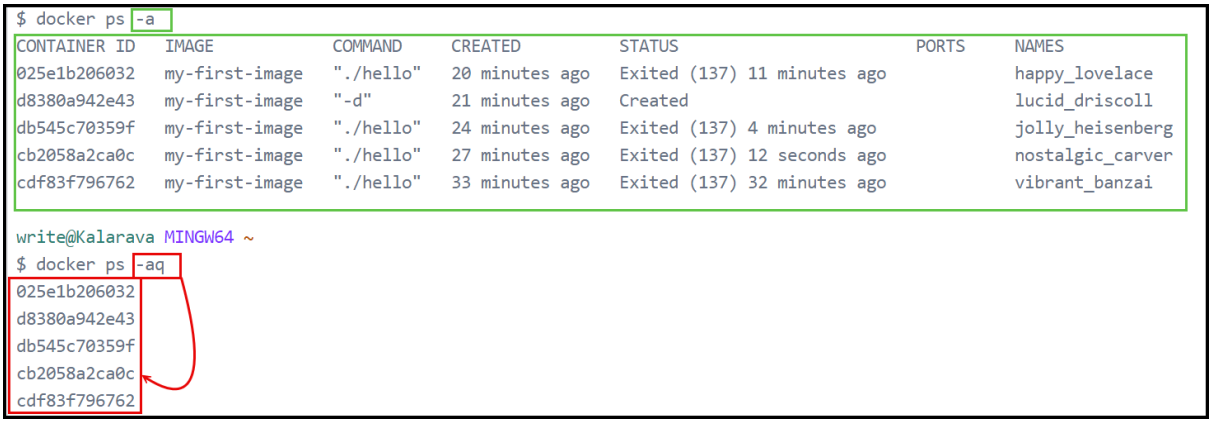
Force Removing Running Containers:

- Use `docker rm -f <container_id>` to forcefully stop and remove a running container.

Remove Multiple Containers:

- To list only the container IDs, we can use…

```
docker ps -aq
```



- Use `xargs` to remove all containers in one command...

```
docker ps -a -q | xargs docker rm
```

```
write@Kalarava MINGW64 ~
$ docker ps -aq | xargs docker rm
025e1b206032
d8380a942e43
db545c70359f
cb2058a2ca0c
cdf83f796762

write@Kalarava MINGW64 ~
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Removing images

List Docker Images:

- Use `docker images` to list all images.

```
$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED          SIZE
my-first-image   latest    fd2cb7f0cbfb   44 minutes ago   564MB
hello-world      linux     b7d87b72c676   20 months ago    24.4kB
```

Remove a Specific Image:

- Use `docker rmi <image_name>` to remove an image.

Handling Dependencies:

- If a container is using an image, stop and remove the container before removing the image.
- Use `docker rmi -f <image_name>` to force image removal, but this may cause issues.

5. Docker Hub

Introduction

Container Image Registry

- A container image registry is a platform for storing and tracking container images.
- Images are tracked using tags, which combine the image name and version (e.g., `image_name:version`).
- If no version is specified, the default tag is latest.

Docker Hub

- Docker Hub is the default registry used by the Docker client.
- Features:
  - Publicly accessible for anyone to push and pull images.
  - Automatically used by Docker when an image is referenced without a specific registry.
- Example:

```
FROM ubuntu:latest
```

  - Fetches the `latest` version of the `ubuntu` image from Docker Hub.

Pushing Images to the Docker Registry

- Create Docker Hub Account.
- Use the `docker login` command to log into Docker Hub via the Docker CLI.
- Renaming (Tagging) an Image for the Docker Registry.
  - `docker tag <local_image> <username>/<repository>:<tag>`
    - `<local_image>` is the name of your existing Docker image.
    - `<username>/<repository>` is the desired name on Docker Hub.
    - `<tag>` is an optional version identifier (e.g., `latest`, `0.0.1`).

- Example:

```
write@Kalarava MINGW64 ~
$ docker tag my-first-image nishithjain/cpp_program:0.0.1

write@Kalarava MINGW64 ~
$ docker images
REPOSITORY                TAG       IMAGE ID       CREATED          SIZE
nishithjain/cpp_program   0.0.1     39643ed06470   2 minutes ago    564MB
my-first-image            latest    39643ed06470   2 minutes ago    564MB
```

- Push the tagged image to Docker Hub using the `docker push` command:

```
docker push <username>/<repository>:<tag>
```

```
write@Kalarava MINGW64 ~
$ docker push nishithjain/cpp_program:0.0.1
The push refers to repository [docker.io/nishithjain/cpp_program]
d766e2cabb3a: Pushed
5f80e8d9c1d5: Pushed
1bd3a2ca0af7: Pushed
075410085186: Pushed
de44b265507a: Pushed
b9cf7c7638c6: Pushed
0.0.1: digest: sha256:39643ed064701865fb4be931e658fb1ed2b170258028cf14d49241a7c2c84dc8 size: 856
```

Checking Your Images in Docker Hub

- Log into Docker Hub via the browser.
- Navigate to the **Images** section.
- Locate the image you just pushed.
    - View details like tags, OS compatibility, and last pushed timestamp.