Lab 3. Run and Operate Containers

The objective of this lab exercise is to get you started with the basic operations for Windows containers. In this lab, you are going to learn how to:

- Get started with Docker command line client utility.
- Launch and work with containers.
- Use common container launch options.
- Define port publishing options to access web applications in a container.
- Troubleshoot running containers.
- Manage container lifecycle.

Finding Information

Open a console/terminal on the host where Docker is installed and run:

```
docker
docker version
docker system info
```

Launching and Operating Your First Container

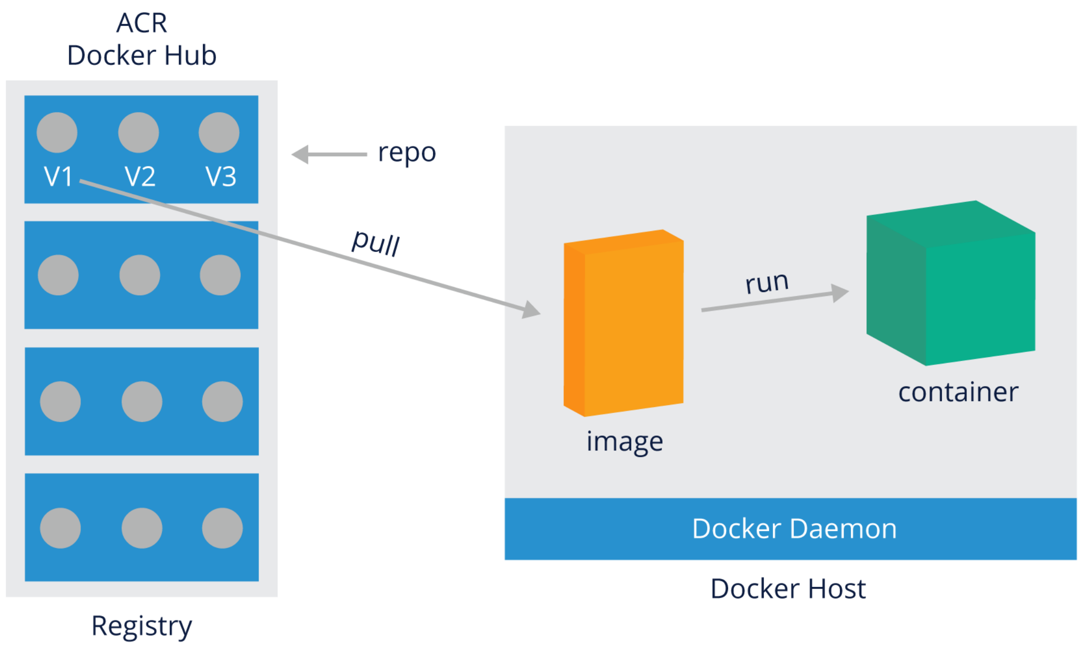
Let's launch your first container To understand what's happening on the Docker server side, you can also open another terminal and run the following command:

```
docker system events
```

When you launch the above command, you will notice no output, and it will appear as if the terminal is hung. However, it's just streaming events from the Docker server and should start showing those events as you perform different operations such as pulling an image and launching a container.

Pulling a Docker Image from Registry

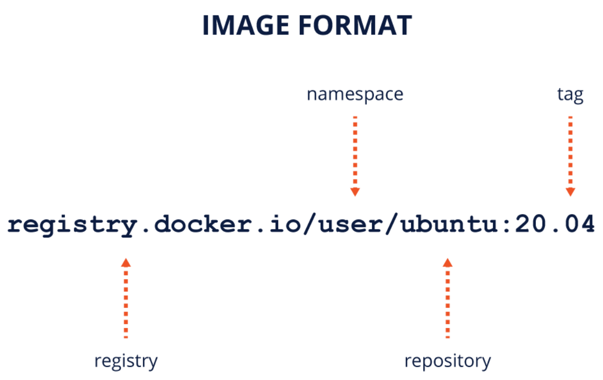
Now, what if you want to run a container? In order to do that, just like in the case of VMs, you need an image available on the host which would run this container. Where does the image come from? The answer is: *container registry* (e.g. Docker Hub/Microsoft Container Registry, etc.). You can think of a container registry as GitHub for your Docker images.



Let’s pull the official Alpine Linux image created by Docker, Inc. To access this image go to the Alpine’s image repository on Docker Hub. You can see an image example below:

`alpine:3.17`

Here, **`alpine:3.17`** is the name of the image. Each image can have up to four fields as shown in the following picture:



To confirm if this image exists locally, pull it if it is not present and validate, run the following commands:

```
docker image ls
docker image pull alpine:3.17
docker image ls
```

The image you pulled has the following fields:

- **`alpine`** = repository
- `3.17` = tag

Launching Your First Container

Now, after pulling this image, launch your container with:

```
docker container run alpine:3.17 uptime
```

or you can use the legacy way and do:

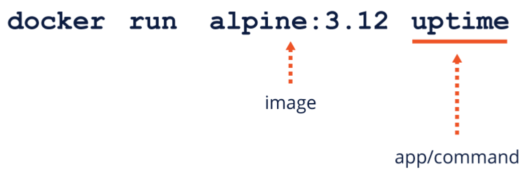
```
docker run alpine:3.17 uptime
```

You may be asking yourself, what's the difference between **`docker container run`** and **`docker run`** command? They are essentially both the same. After the 1.13 version of Docker became available, new categories of commands were introduced, and **`docker run`** became **`docker container run`**. However, the command is still backwards compatible. In fact, users are so used to the previous version of the command, that they often continue to use the old approach. Feel free to use whichever command you prefer.

In this exercise, we will go with the second option:

```
docker run alpine:3.17 uptime
```

LAUNCHING CONTAINER



```
docker run alpine:3.12 uptime
```

When you launch the container, pay attention to the following two things:

- The output on the terminal
- Events streamed from the other terminal where you launched **`docker system events`**

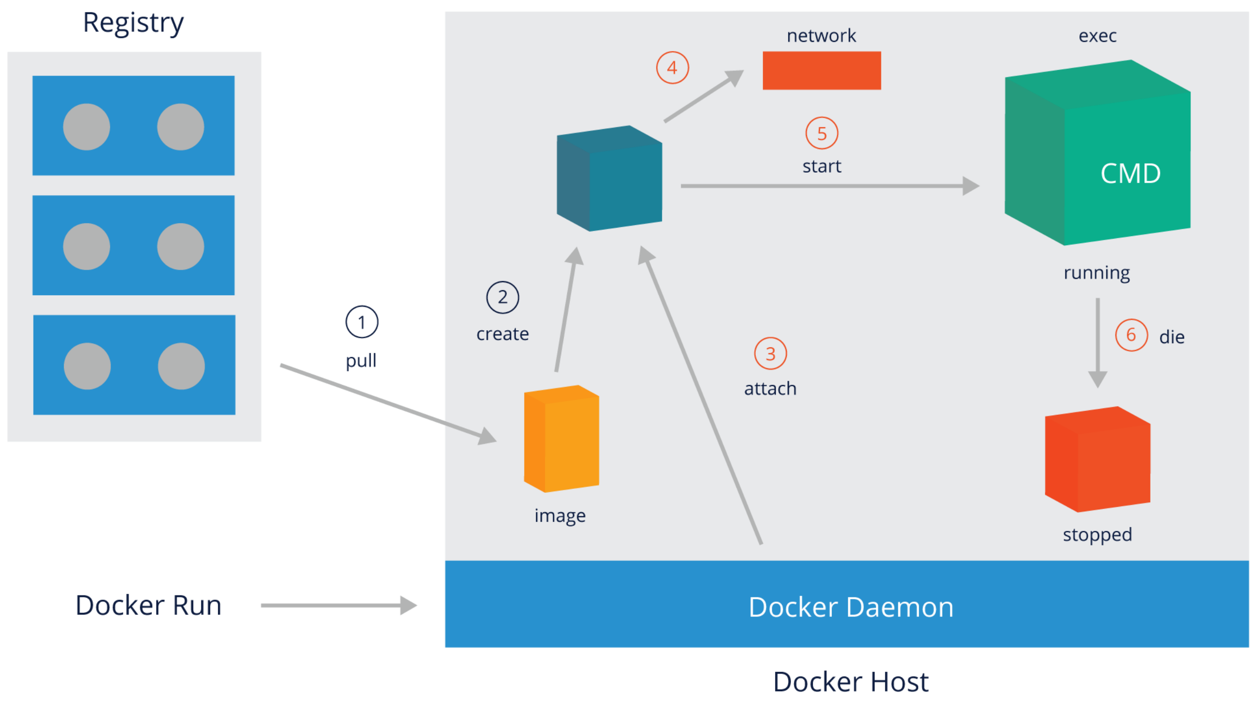
Below is the sample output of the command presented above:

```
docker run alpine:3.17 uptime
Unable to find image 'alpine:3.17' locally

3.17: Pulling from library/alpine
df20fa9351a1: Already exists
Digest:
sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Downloaded newer image for alpine:3.17
 13:33:30 up 32 days,  8:52,  load average: 0.00, 0.04, 0.00
```

The following is a list of events on the server side:

- container create
- container attach
- network connect
- container start
- container exec_create
- container exec_start: uptime
- container die
- network disconnect



Use the following command to check if the container you started is running:

```
docker ps
```

The above command is used to list the currently running containers. If your container is not showing, check its status using the `-l` flag. It will list the last run container:

```
docker ps -l
```

Below you can see the sample output:

```
 CONTAINER ID         IMAGE                 COMMAND               CREATED
STATUS                          PORTS                   NAMES
09aabefee6e2         alpine:3.17          "uptime"               About a minute ago
Exited (0) About a minute ago                           relaxed_lichterman
```

Its status says **`Exited`**.

Let's analyze what happened after you launched your container:

- Docker tried to look up the image locally. In our case, the image was present, so this step was skipped. Otherwise, Docker would try to download it from the registry.
- Docker created the container, e.g. all the components required to run a process in a contained environment.
- Docker service attached itself to this container to monitor the processes, manage logs, etc.

- Docker asked the kernel to set up network configurations for the container. This step created the virtual network interface for the container, set up the IP address, etc.
- Docker started running the container.
- Docker launched the actual command or the application of your choice (e.g. **`uptime`**). The application's start can also be configured implicitly in the image metadata, so that you do not have to provide it at the launch time.
- Since this example had a one-off command (**`uptime`** instead of a long-running process such as Apache), as soon as the process exited, Docker thought its job was done and it stopped the container.
- Once the container stopped, the network was also disconnected from the container. Please remember that the network is a separate entity than the container itself.

Listing Containers

Let's now launch a few more containers with different commands:

```
docker run alpine:3.17 hostname
docker run alpine:3.17 ps aux
docker run alpine:3.17 ifconfig
```

Now try to list containers using the following flags with the **`docker ps`** command:

- **`-l`**: to list the last container
- **`-n 3`**: to list the last three containers (you can try different integers instead of 3)
- **`-a`**: to list all containers

For example:

```
docker ps
docker ps -l
docker ps -n 3
docker ps -a
```

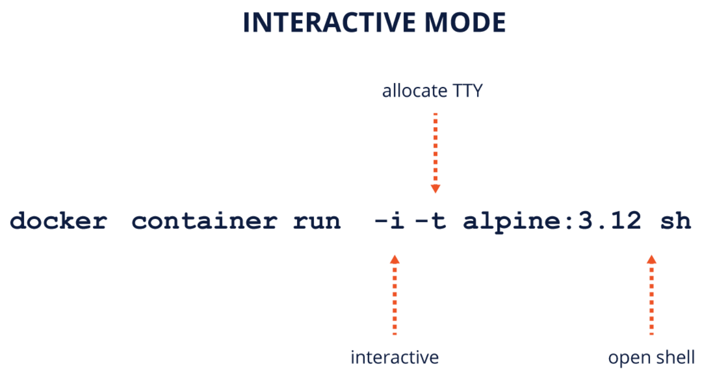
Using Common Container Runtime Options

In the previous section, we launched and ran a few containers. We didn't use any flags with a **`docker run`** command, but instead introduced the one-off command. In this section, we will discuss some common options that you can use with the **`docker run`** command. You will also see how to run containers long-term and observe how they work.

Launching an Interactive Container

The first two options that you can use are: **`-i`** and **`-t`** (you can also combine these options into **`-it`**). Also, this time, you will launch a long-running command, such as **`cmd`**:

```
docker run -it alpine:3.17 sh
```



When you run the above command, you will notice the following:

- This time not only the container is launched, but you also land inside the container with a shell prompt. You can finally get a better feel of the container!
- The container is not stopped but rather remains in a running state. This is because you launched it with **`sh`**. It will keep running as long as the shell is running and you don't exit.

While you are inside the container, you may want to run some commands and observe how containers work. Type **`exit`** and click on Enter to leave the container (alternatively you can do **`^d`**). Exiting from the shell will stop the container.

Now, try to launch a container with an Ubuntu image. Use the command below, observe Docker system events and look for differences. For example, you should see an event for the image being pulled, if you have not used this image before.

```
docker container run --rm -it ubuntu bash
```

where,

- **`ubuntu`**: is the image for the latest version of Ubuntu.
- **`bash`**: launch **`bash`** instead of **`sh`** as before, because Ubuntu comes with **`bash`**.
- **`--rm`**: tells Docker to delete the container once it has stopped (on exit from **`bash`**). This is a useful option to clean up one-off containers created just for testing.

You can also try running some commands such as those presented below inside the container:

```
ps
uptime
which apt
free
```

```
cat /proc/cpuinfo
```

Again, exiting the shell should stop and remove this container. Check if this is the case by running **`docker ps -a`** command.

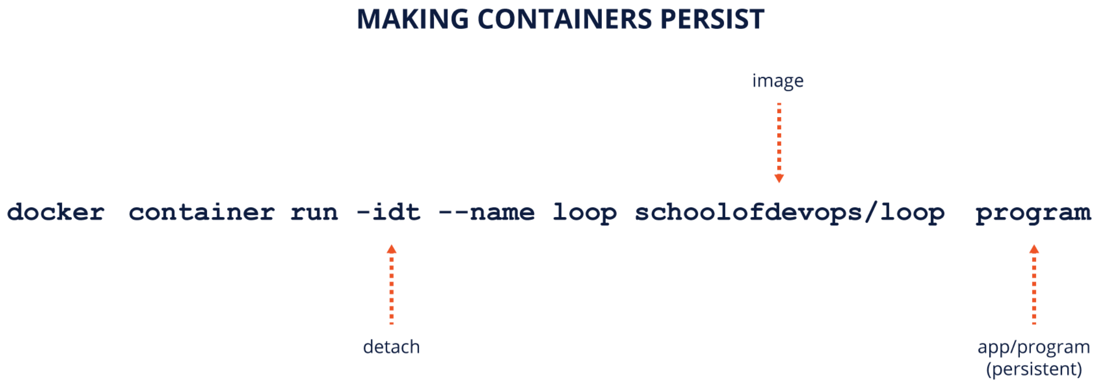
Running a Container in the Background/Detached Mode

The most desired way to run a container is to launch an application with a container, and keep it running in the background. This can be achieved by using the **`--detach`** option. This time, I will be launching a sample application:

```
docker run -idt --name loop --rm schoolofdevops/loop program
```

where,

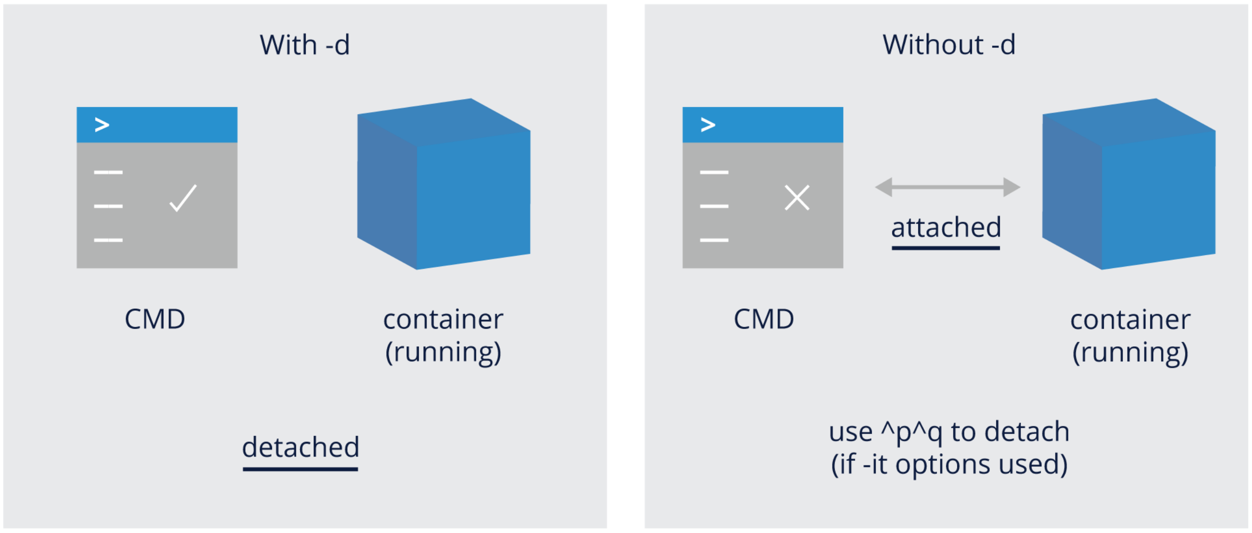
- **`--name`**: provides a name to the container instead of the automatically generated random name that Docker has assigned to it.
- **`schoolofdevops/loop`**: an image with the application **`program`** built-in.
- **`program`**: name of the command/application to be run on the launch of this container.


MAKING CONTAINERS PERSIST
image
docker container run -idt --name loop schoolofdevops/loop program
detach
app/program (persistent)

You should notice that the container is not only launched but is also running in the background, allowing you to proceed with the next commands/tasks. If you want to see the difference, try to run another container without the **`-d`** option:

```
docker run -it --rm schoolofdevops/loop
```

A container with the same image is launched without **`-d`** now. Note that you are attached to the process running inside the container now. If you use **`-d`** instead, this container will run in the background with the detached option.



If you launch a container without **`-d`**, remember to use **`^p ^q`** as an escape sequence to detach from the container's process. This will only work if **`-it`** options were used to launch the container. Use **`^c`** in other cases to kill the container.

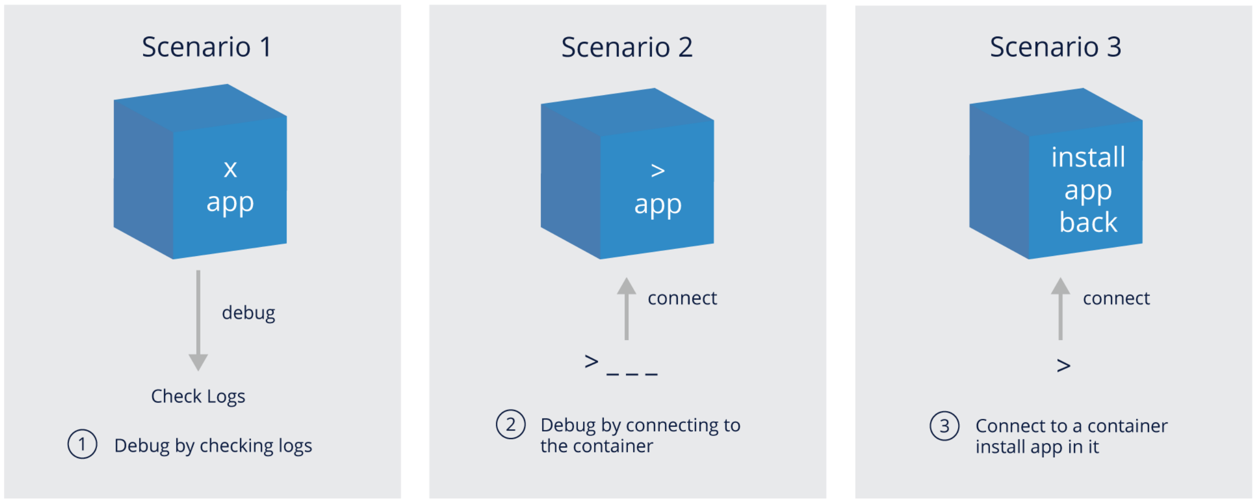
Best practice: When launching containers use **`-idt`** options. Out of these, **`-d`** is a must. **`-it`** is optional but recommended as you can detach/attach from the process/command/application running inside the container.

Troubleshooting and Modifying Containers

So far, you have launched applications in a container. What are the next steps? Most of the time, you may have to do some debugging or troubleshooting on a running container. How do you achieve that? There are two operations involved while debugging/modifying containers:

- You may want to get the logs from the application.
- You may want to connect to the container (e.g. open a shell inside it to manage and debug further).

Let's take a look at a few example scenarios.



- **Scenario 1**: You have launched an application which is misbehaving, so you want to debug it. The first thing that you may want to do is get the application logs.
- **Scenario 2**: You have an application running inside a container. Logs don't tell you much about what it's doing. You may want to connect to this container, open a shell, and debug further.
- **Scenario 3**: You have a container running a base OS. You want to modify it by installing a web server on it.

Scenario 1: Checking Application Logs

Let's consider the first scenario. If you want to check application logs, you can use either of the following commands (assuming your container's name is `loop`):

```
docker ps
```

Note container's name/ID:

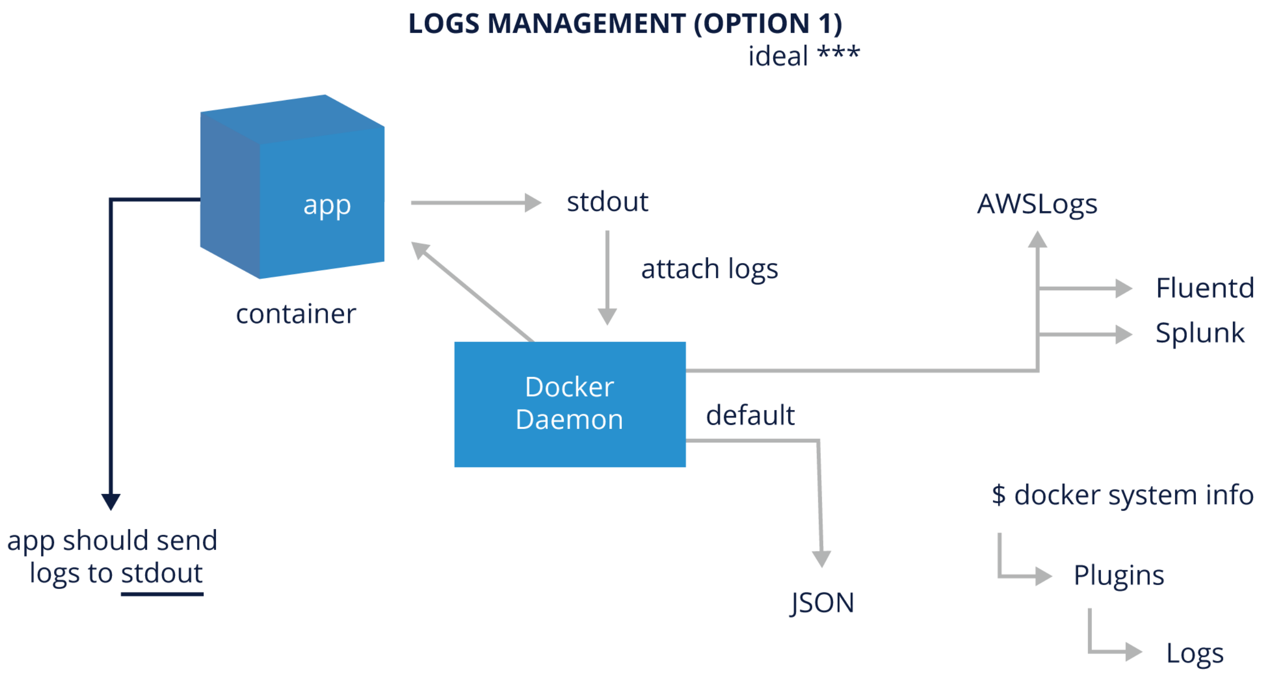
```
docker logs loop
```

This command will follow the logs and notify you as they update:

```
docker logs -f loop
```

Use `^c` to stop following.

The above command does show the logs for the `program` running inside the container. Well, the question now is, How does Docker know about these application logs? An answer to that question is depicted in the following diagram:

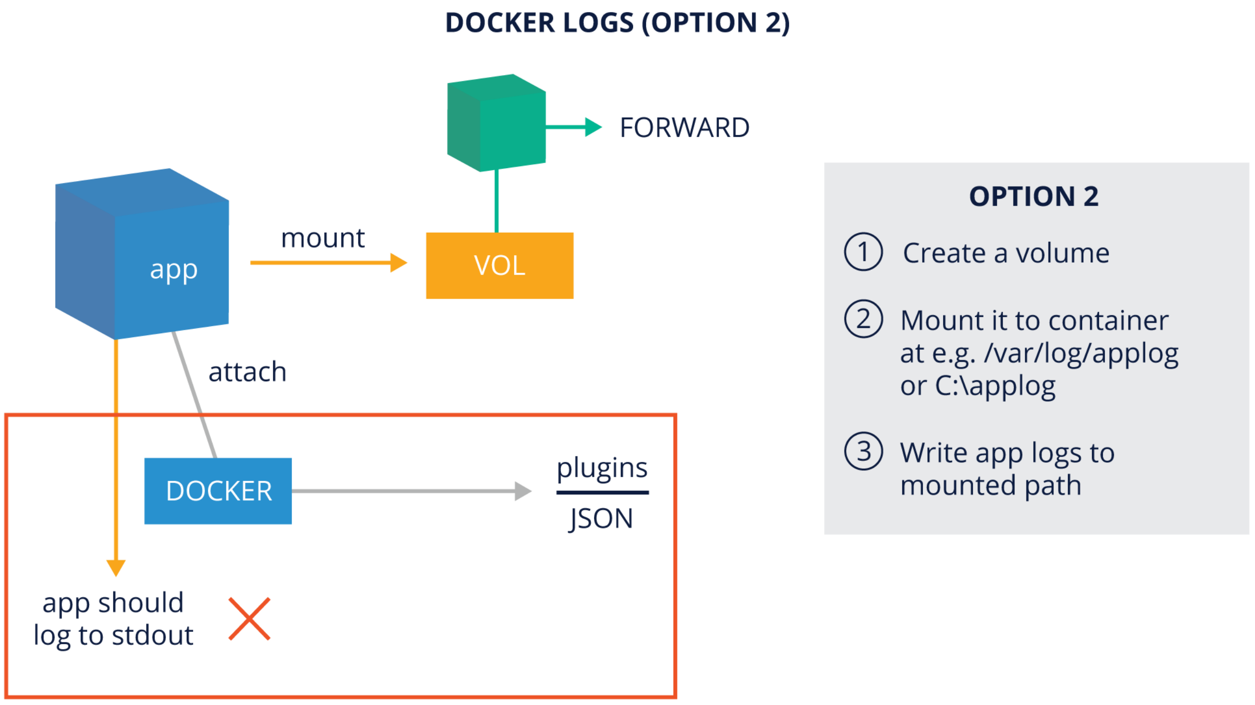


where,

- Docker attaches to the application’s standard out (stdout).
- Application writes logs to standard out (see the sample **`program`** illustrated in the diagram).
- Docker daemon then forwards the logs based on the logs plugin used (the default is json-file). However, Docker is capable of forwarding logs to remote systems using plugins such as awslogs, gcplogs, syslog, or plunk.
- This integration is available out-of-the-box. You can check existing log plugins using **`docker system info`** command plugins > *Logs* section.
- This also needs an application to log to standard out.

To learn more about Docker's logging drivers see Docker’s Documentation, *“Configure Logging Drivers”*.

Now, what happens if your application cannot be modified to log to standard output (stdio), or what if it generates multiple log streams and writes it to multiple files? In such cases, we recommend that you use the following option:



where,

- You mount a volume to the container. This needs to be done while launching the container. Volume can be local or remote.
- Mount it at the path where applications write the logs, such as **`/var/log/applog`** or **`C:\applog`**.
- Then, ensure that your application writes the logs only to that path.
- Optionally, from the volume, you can forward the logs to remote systems using forwarders.

Scenario 2: Getting Inside the Container's Shell

Unfortunately, checking the logs is not enough. You may want to get inside a shell of the container, similar to an SSH connection to a remote host. Alternatively, you can execute a one-off command against the running container. You can go either way using the **`exec`** command that Docker offers:

```
docker ps
```

Note container name/ID

To run a one-off command:
```
docker exec loop uptime
```

To get a shell access inside the container:
`docker exec -it loop sh`

The most commonly used command is the last one with the **`-it`** options, and **`sh`** or **`bash`** as the command, depending on which shell your Docker image comes with.

Once you open a shell above, you can execute any command that is available as part of that containers' image.

Scenario 3: Modifying Container by Installing an Application

Scenario 3 is an extension of Scenario 2. Launch a container with an Ubuntu OS this time:

```
docker container run -idt --rm --name redis  ubuntu bash
```

Now, log in to this container using **`exec`** and **`bash`** as in **`docker exec -it redis bash`**, and follow the steps to enable and start a web server:

```
apt-get update
apt-cache search redis
apt-get install redis
redis-server
```

The above sequence of commands should install and launch the Redis server on this host and display output such as following:

```
399:C 01 May 2023 09:44:33.756 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
399:C 01 May 2023 09:44:33.756 # Redis version=6.0.16, bits=64,
commit=00000000, modified=0, pid=399, just started
399:C 01 May 2023 09:44:33.756 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server
/path/to/redis.conf
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 6.0.16 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 399
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
```

```
          `-._           _.-'
              `-.__.-'

399:M 01 May 2023 09:44:33.758 # Server initialized
399:M 01 May 2023 09:44:33.758 * Ready to accept connections
```

From here, you can stop the application by using `^c` and then exit the container by pressing `^d` to get back to the original shell you started from.

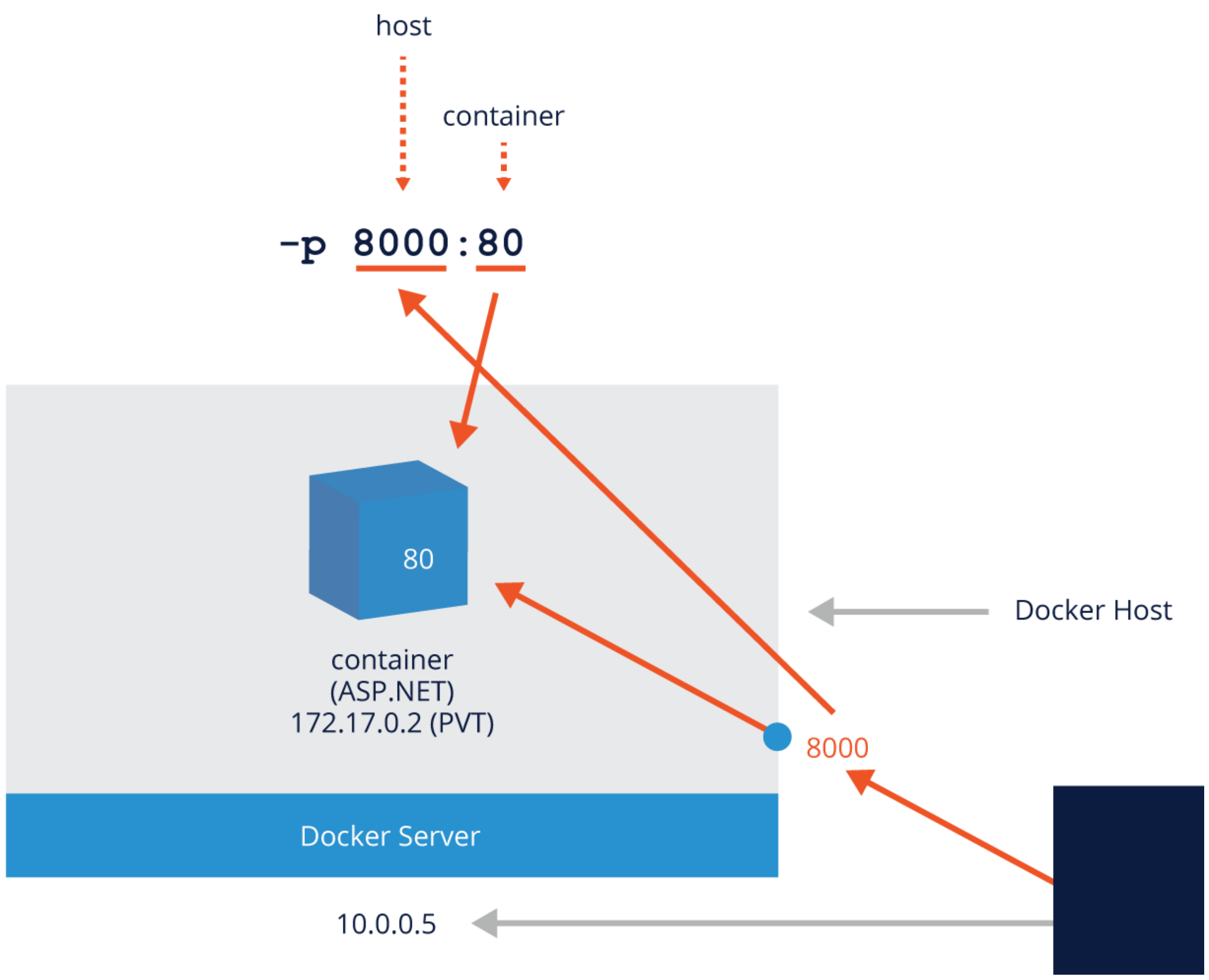
Congratulations! You have learned the basic troubleshooting of containers with logs and exec.

Accessing Web Applications with Port Mapping

While you have learned many common options to launch containers with, so far you have launched only the mock applications without any web interface. It's now time to launch a real web application. How to access this application? Well, first you need to learn about one more option: *publishing ports*/*port mapping*:

```
docker run -idt -p 8000:80 --rm schoolofdevops/vote
```

PORT MAPPING



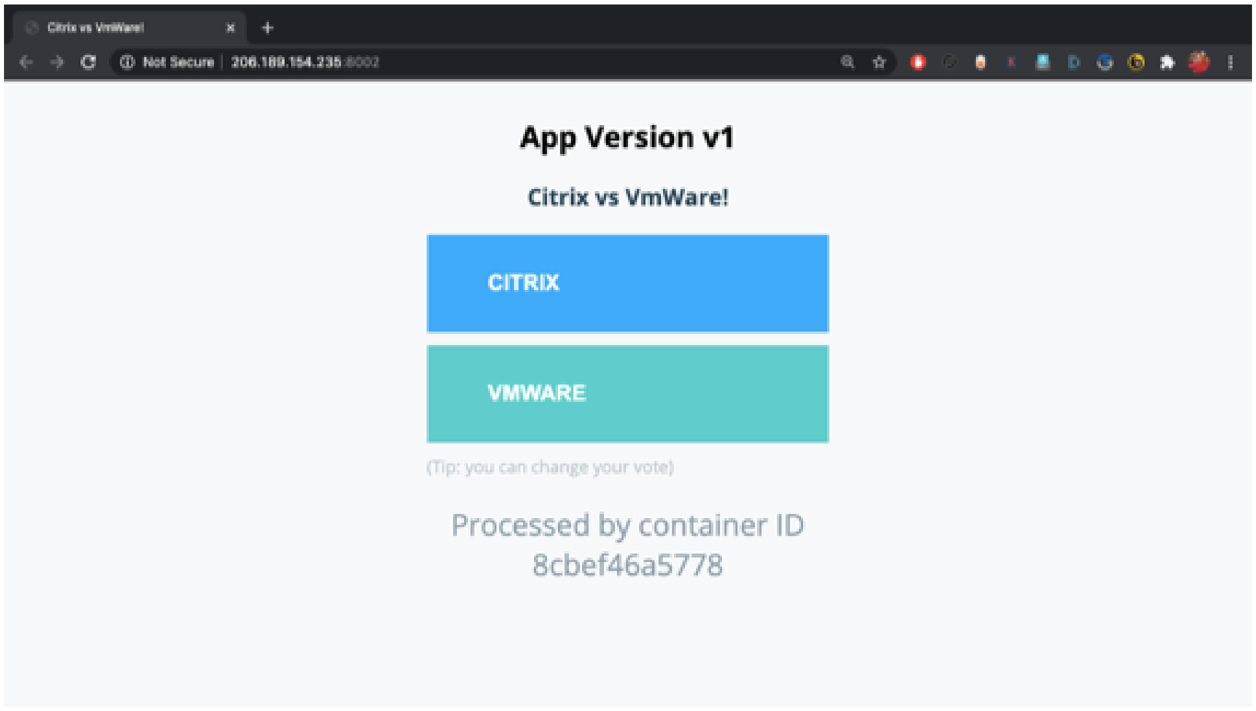
where,

- **`-p 8000:80`**: maps Docker host's **`8000`** port to the container's **`80`** port, while the actual web application is running.

This allows you to access the application using host IP and host port such as **`http://hostip:hostport`**:

- If you are using Docker Desktop replace **`hostip`** with **`localhost`** and use the following URL: **`http://localhost:8000`**.
- If you are using a remote server, use the IP address or hostname of it to access the application, e.g. for my cloud VM it's 'http://20.41.76.15:8000'.
- Ensure that port **`8000`** is open from the firewall configurations if using a remote host/ cloud server.

For example:



App Version v1

Citrix vs VmWare!

(Tip: you can change your vote)

Processed by container ID 8cbef46a5778

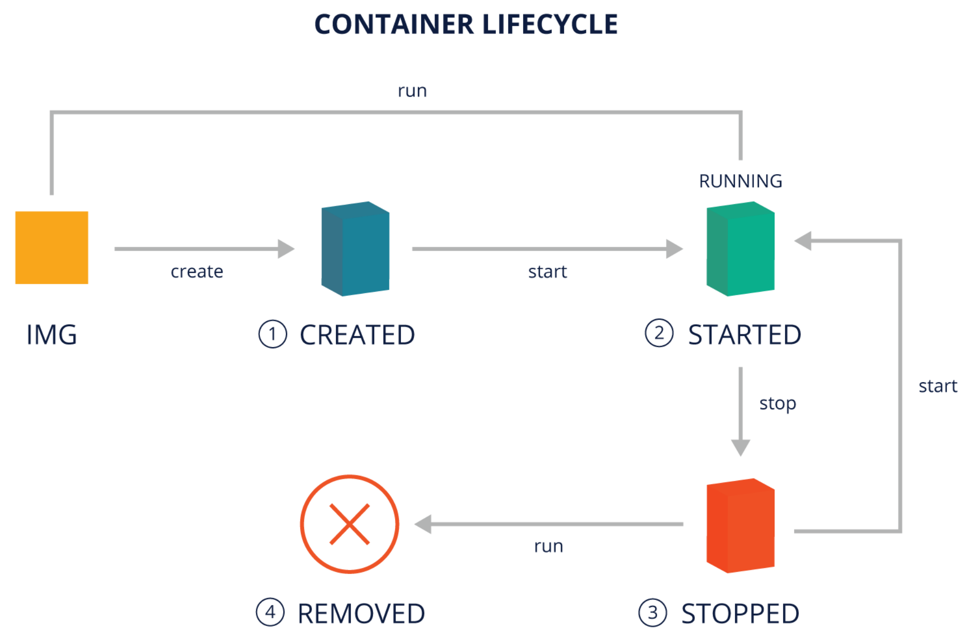
Apart from using the **`-p`** option, you can also use the following:

- Using only the **`-P`** option (capital letter) will automatically pick up the exposed port defined in the container image and map it to a host port starting with 32768. It will automatically increment this number for future port mappings.
- Using an option such as **`-p 80`** will pick up the container port 80 and map it to a host port starting with 32768. It will also automatically increment this number for a future port.

Managing Container's Lifecycle

A container goes through the following four lifecycle phases:

1. CREATED
2. STARTED/RUNNING
3. STOPPED
4. REMOVED



We have already learned how to launch a container and put it in the STARTED/RUNNING state using **`docker run`** command. You can also individually create and start containers using the following sequence:

```
docker create -it --name twostep ubuntu bash
docker start twostep
```

Please remember that you cannot use the **`-d`** option with a **`create`** command as detaching a container while creating it is invalid. In this case, when you start the container, it automatically starts with detached mode.

To stop a running container use the following command:

```
docker stop twostep
```

where,

- **`twostep`**: is the name of the container.

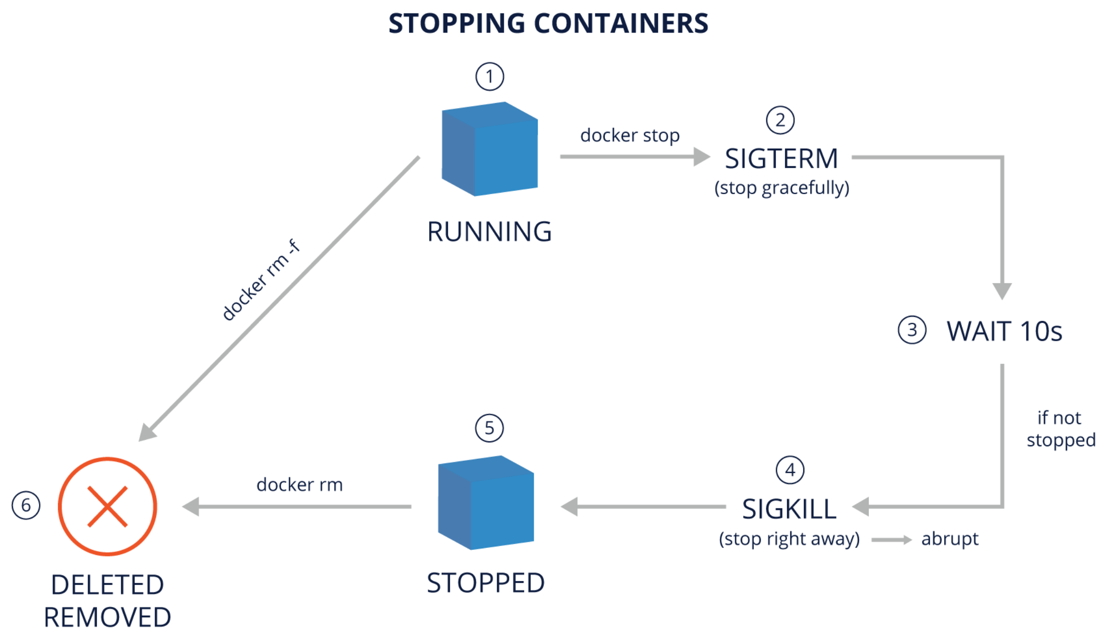
You can also stop multiple containers by providing a space separated list:

```
docker stop container_id  container_id  container_id
```

A stopped container can be started back with:

```
docker start twostep
```

NOTE: *Stopping a container does not delete the data/changes you have made to the container. When you talk about immutable deployments (e.g. Kubernetes), the containers are deleted and replaced for every update, so retaining the data is not possible in such cases. However, in local environments, it can be stopped. Which means it retains all changes until it's removed and can be used just like a VM.*



A stopped container can be removed using **`docker rm`** commands as follows:

```
docker rm twostep
```

A running container needs the **`-f`** option for it to be stopped and removed, for example:

```
docker rm -f  vote
```

You can also delete all stopped containers in one go using the following command:

```
docker container prune
```

NOTE: *The **`docker container prune`** command will delete all stopped containers. This may not be what you want if you created containers to use for your continuous development work. So use this command very carefully!*

Summary

In this lab exercise, you learned how to launch a container and use the most common container runtime options. You also observed what happens when you run a container and how to access containers by using the port mapping. In addition, you got familiar with common troubleshooting tasks such as logs and exec, and saw how to stop and remove the containers. These are essential skills for anyone who wants to get started working with containers or use a container orchestrate engine.