Containers

Containers Overview

Container orchestration systems like Kubernetes depend on the underlying container technology of the operating system and, in the case of Kubernetes, a pluggable container manager.

Docker is an open-source software system that automates the deployment of application processes inside software containers. Containers abstract an application's operating environment from the underlying operating system. Containers use the resource constraints and isolation features of the Linux kernel, such as cgroups and namespaces, allowing many diverse containers to run within a single Linux instance without resource contention. Containers eliminate the overhead and startup latency of a full virtual machine while preserving most of the isolation benefits. In recent years Windows has replicated most of the features found in Linux containers, making Windows a viable platform for container technology and a supported Kubernetes worker node OS.

Docker can package an application and its dependencies into a Container Image, which can then be used to launch containerized processes on any compatible system. This enables applications to run reliably across a number of OS versions and distributions with various configurations in a range of cloud settings.

In this lab we will install and test Docker Community Edition on the RX-M cloud supplied virtual machine. The lab VM is an Ubuntu server available via an SSH connection. If you need help setting up an ssh client, some instructions can be found here: https://github.com/RX-M/classfiles/blob/master/ssh-setup.md

The Lab system user account is "ubuntu"; this user has full sudo permissions.

1. Connect to the lab VM

You will be assigned an AWS workstation IP address and SSH key to complete the labs in this course.

To access your instance:

Open an SSH client in a Terminal on MacOS or Putty/MobaXterm/CMD/PowerShell/GitBash on Windows. Locate your private key file (e.g. key.pem). Your key must not be publicly viewable for SSH to work; use the command `chmod 400 key.pem` if needed:

```
@laptop:~$ chmod 400 \path-to-ssh-key\key.pem
```

Then, connect to your instance using its IP Address, username `ubuntu` and your SSH key. The `-i` option tells ssh to use the specified identify file (private key). e.g. `ssh -i \path-to-ssh-key\key.pem ubuntu@<IP>`.

```
@laptop:~$ ssh -i \path-to-ssh-key\key.pem ubuntu@55.55.55.55

The authenticity of host '55.55.55.55 (55.55.55.55)' can't be established.
ECDSA key fingerprint is
SHA256:PKRtJDb4CvjAq4Hh7qpISyZQdPXw3oc5QcJFRzhRkuQ.
Are you sure you want to continue connecting (yes/no)? yes

...

~$
```

When your prompt changes to something like `ubuntu@ip-172-31-4-159:~$` you are connected to the lab VM.

2. Install Docker

Docker provides a convenience script to install the latest version of Docker. It is not recommended to use in a production environment, however, it is simple and fast for testing and experimentation. Install docker on your lab system using the get.docker.com install script:

> DO NOT follow the instructions provided by the script's output!

```
~$ wget -qO - https://get.docker.com/ | sh

# Executing docker install script, commit:
e5543d473431b782227f8908005543bb4389b8de
+ sudo -E sh -c apt-get update -qq >/dev/null
+ sudo -E sh -c DEBIAN_FRONTEND=noninteractive apt-get install -y -qq apt-
transport-https ca-certificates curl >/dev/null
+ sudo -E sh -c install -m 0755 -d /etc/apt/keyrings
+ sudo -E sh -c curl -fsSL "https://download.docker.com/linux/ubuntu/gpg"
| gpg --dearmor --yes -o /etc/apt/keyrings/docker.gpg
+ sudo -E sh -c chmod a+r /etc/apt/keyrings/docker.gpg
+ sudo -E sh -c echo "deb [arch=amd64 signed-
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu
jammy stable" > /etc/apt/sources.list.d/docker.list
+ sudo -E sh -c apt-get update -qq >/dev/null
+ sudo -E sh -c DEBIAN_FRONTEND=noninteractive apt-get install -y -qq
docker-ce docker-ce-cli containerd.io docker-compose-plugin docker-ce-
rootless-extras docker-buildx-plugin >/dev/null
+ sudo -E sh -c docker version
Client: Docker Engine - Community
 Version:           24.0.7
 API version:       1.43
 Go version:        go1.20.10
 Git commit:        afdd53b
 Built:             Thu Oct 26 09:07:41 2023
 OS/Arch:           linux/amd64
 Context:           default
```

```
Server: Docker Engine - Community
 Engine:
  Version:          24.0.7
  API version:      1.43 (minimum version 1.12)
  Go version:       go1.20.10
  Git commit:       311b9ff
  Built:            Thu Oct 26 09:07:41 2023
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.27
  GitCommit:        a1496014c916f9e62104b33d1bb5bd03b0858e59
 runc:
  Version:          1.1.11
  GitCommit:        v1.1.11-0-g4bccb38
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0

==========================================================================
======

To run Docker as a non-privileged user, consider setting up the
Docker daemon in rootless mode for your user:

    dockerd-rootless-setuptool.sh install

Visit https://docs.docker.com/go/rootless/ to learn about rootless mode.


To run the Docker daemon as a fully privileged service, but granting non-
root
users access, refer to https://docs.docker.com/go/daemon-access/

WARNING: Access to the remote API on a privileged Docker daemon is
equivalent
         to root access on the host. Refer to the 'Docker daemon attack
surface'
         documentation for details: https://docs.docker.com/go/attack-
surface/

==========================================================================
======

~$
```

For our in-class purposes, eliminating the need for `sudo` execution of the `docker` command will simplify our practice sessions--but setting up docker for rootless access (as the script suggests) would be overly complicated. The easiest way to make it possible to connect to the local Docker daemon without `sudo` is to add our user id to the *docker* group. To add your user to the *docker* group execute the following command:

```
~$ sudo usermod -aG docker $(whoami)

~$
```

Even though the *docker* group was added to your user's group list, your login shell maintains the old groups. After updating your user groups you will need to restart your login shell to ensure the changes take effect.

In the lab system the easiest approach is to logout at the command line:

```
~$ exit

logout
Connection to 55.55.55.55 closed.

@laptop:~$
```

Log back in as *ubuntu* with the SSH key:

```
@laptop:~$ ssh -i \path-to-ssh-key\key.pem ubuntu@55.55.55.55

...

~$
```

After logging back in, check to see that your user shell session is now a part of the *docker* group:

```
~$ id

uid=1000(ubuntu) gid=1000(ubuntu)
groups=1000(ubuntu),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(au
dio),30(dip),44(video),46(plugdev),119(netdev),120(lxd),999(docker)

~$
```

Great. Your current shell, and any new shell sessions, can now use the `docker` command without sudo.

3. Verify the installation

Verify that the Docker server (aka. daemon, aka. engine) is running by checking the version of all parts of the Docker platform with the `docker version` subcommand:

```
~$ docker version
```

```
Client: Docker Engine - Community
 Version:           24.0.7
 API version:       1.43
 Go version:        go1.20.10
 Git commit:        afdd53b
 Built:             Thu Oct 26 09:07:41 2023
 OS/Arch:           linux/amd64
 Context:           default

Server: Docker Engine - Community
 Engine:
  Version:          24.0.7
  API version:      1.43 (minimum version 1.12)
  Go version:       go1.20.10
  Git commit:       311b9ff
  Built:            Thu Oct 26 09:07:41 2023
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.27
  GitCommit:        a1496014c916f9e62104b33d1bb5bd03b0858e59
 runc:
  Version:          1.1.11
  GitCommit:        v1.1.11-0-g4bccb38
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0

~$
```

The client version information is listed first followed by the server version information.

You can also use the Docker client to retrieve basic platform information from the Docker daemon:

```
~$ docker system info

Client: Docker Engine - Community
 Version:    24.0.7
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.11.2
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx
  compose: Docker Compose (Docker Inc.)
    Version:  v2.21.0
    Path:     /usr/libexec/docker/cli-plugins/docker-compose

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 24.0.7
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: systemd
 Cgroup Version: 2
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries
splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: a1496014c916f9e62104b33d1bb5bd03b0858e59
 runc version: v1.1.11-0-g4bccb38
 init version: de40ad0
 Security Options:
  apparmor
  seccomp
   Profile: builtin
  cgroupns
 Kernel Version: 6.2.0-1017-aws
 Operating System: Ubuntu 22.04.3 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 2
 Total Memory: 3.769GiB
 Name: ip-172-31-4-161
 ID: a0f317cc-1e20-465c-bcad-bdb88ac30860
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

~$
```

Food for thought:

- What version of Docker is your `docker` command line client?
- How many containers are running on the server?
- What version of Docker is your Docker Engine?
- What is the Logging Driver in use by your Docker Engine?

- What is the Storage Driver in use by your Docker Engine?
  - Does the word "Driver" make you think that these components can be substituted?
- Is the server in debug mode?
- What is the Docker Engine root directory?

4. Run a container

As a final lab step we will run our first container. Use the `docker container run` command to run the hello image:

```
~$ docker container run rxmllc/hello

Unable to find image 'rxmllc/hello:latest' locally
latest: Pulling from rxmllc/hello
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
5c9b4d01f863: Pull complete
Digest:
sha256:0067dc15bd7573070d5590a0324c3b4e56c5238032023e962e38675443e6a7cb
Status: Downloaded newer image for rxmllc/hello:latest
 _________________________________
/ RX-M - Cloud Native Consulting! \
\ rx-m.com                        /
 ---------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

~$
```

You now have a working Docker installation!

Food for thought:

- What kind of program created the container output?
  - Does it matter?
- What operating environment existed inside the container when it ran?
  - Does it matter?
- Did the Docker engine run the container from a hello image it found locally?
- In the Docker output the "hello" image is listed with a suffix, what is the suffix?
  - What do you think this suffix represents?
- In the Docker output the "hello" image is listed with a prefix followed by a slash, what is the prefix?
  - What do you think this prefix represents?

5. Run a container in the background

When processes are run in the background they are often referred to as daemons. You can run containers in the background as daemons with the `docker container run -d` command. The *-d* switch stands for "detached" and runs the container detached from your shell's input and output.

Imagine we have a service and that its job is to log data to STDOUT every 10 seconds. We can run this service in the background with the `-d` switch. Execute the following Docker command:

```
~$ docker container run -d --name c1 ubuntu:20.04 \
/bin/sh -c "while true; do echo 'hello Docker'; sleep 10; done"

Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
527f5363b98e: Pull complete
Digest:
sha256:f2034e7195f61334e6caff6ecf2e965f92d11e888309065da85ff50c617732b8
Status: Downloaded newer image for ubuntu:20.04
ac698b8c2d395e5a1e2871a9eca50fb23bc6a8d05f0f2ecd2efe91a83342e06a

~$
```

Let's break down the command above:

- `docker` - invokes the docker CLI tool which makes API calls to the docker daemon
- `container run` - the docker subcommand that creates and starts a containerized process
- `-d` - run the containerized process "detached" in the background
- `--name c1` - name our container
- `ubuntu:20.04` - the image to use when launching the containerized process (contains the root filesystem for the process)
- `/bin/sh -c "while true; do echo 'hello Docker'; sleep 10; done"` - runs a shell to process a short script

Thus our containerized shell will simply echo 'hello Docker' every 10 seconds.

Display the running containers:

```
~$ docker container ls

CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS          PORTS     NAMES
ac698b8c2d39   ubuntu:20.04   "/bin/sh -c 'while t…"   12 seconds ago   Up
10 seconds             c1

~$
```

You should see the new container running in the background.

The `docker container logs` subcommand can display the output of a background container. Display the STDOUT log for the container you started above by passing the name (`c1`) of the container:

```
~$ docker container logs c1

hello Docker
hello Docker
hello Docker
hello Docker


~$
```

5.1. CHALLENGE: logs

- Get help on the `docker container logs` subcommand
- Display the container log data with timestamps added
- Use the "follow" option to continuously display the log output of the container

6. NGINX container as a service

Next let’s run an actual service within a container. Nginx is a popular web server with an available image on Docker Hub. Run an Nginx container instance:

```
~$ docker container run -d --name c2 nginx:1.20

Unable to find image 'nginx:1.20' locally
1.20: Pulling from library/nginx
f7ec5a41d630: Pull complete
d064bcebeb06: Pull complete
975c0a9d7b94: Pull complete
1384c783a2a3: Pull complete
a0f1673d45e1: Pull complete
fc05b7dad2e3: Pull complete
Digest:
sha256:c1e6c074ac4f7a543d5a55379ba7bb0d331961b9deca1bc5c5902ef0a354a63d
Status: Downloaded newer image for nginx:1.20
8fbca366157fd7503205760bb80342797e3ff3bb7e2a0d44fa81b18211743fa2

~$
```

By default, containers run processes using the same kernel as the host. This means that any suitably privileged environment (like the host) can still see the processes in the container.

Use the `docker container top` subcommand to display the processes running within the container:

```
~$ docker container top c2

UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
```

```
root                9149                9125                0
00:17               ?                   00:00:00            nginx: master
process nginx -g daemon off;
systemd+            9199                9149                0
00:17               ?                   00:00:00            nginx: worker
process
systemd+            9200                9149                0
00:17               ?                   00:00:00            nginx: worker
process

~$
```

Using `top` allows you to see how the host sees the containerized process.

Imagine you are a BSD Unix fan and prefer an "aux" style `ps` output to the default `top` output. You can have `top` display any `ps` style you like by simply including the *ps* options after the container ID/Name. For example:

```
~$ docker container top c2 aux

USER                PID                 %CPU                %MEM
VSZ                 RSS                 TTY                 STAT
START               TIME                COMMAND
root                9149                0.0                 0.1
8680                6016                ?                   Ss
00:17               0:00                nginx: master process nginx -g
daemon off;
systemd+            9199                0.0                 0.0
9084                2640                ?                   S
00:17               0:00                nginx: worker process
systemd+            9200                0.0                 0.0
9084                2640                ?                   S
00:17               0:00                nginx: worker process

~$
```

6.1. CHALLENGE: top

- Run the top command with the `-t` ps switch
- Run the top command with the `-w` ps switch
- Try your own favorite ps switches with `docker container top`
- Learn more here https://docs.docker.com/engine/reference/commandline/top/
- And even more here `man ps` (especially when to use '-' or not)

7. Container exec

The "container" concept is implemented by the Linux kernel using namespaces and cgroups. Many processes can run within the isolation/constraints of the same "container". Notice that the previous Nginx container has no shell running within it. This is typical of service/microservice containers. You can still launch a shell within the container to perform diagnostics using the `docker container exec` subcommand.

Run a new interactive shell within the Nginx container:

```
~$ docker container exec -it c2 /bin/sh

#
```

This launches a new process inside the container. The primary process will continue to run uninterrupted. This is in contrast to `docker container attach` which gives you control over (and the potential to interrupt) the primary process in the container.

Now list all of the processes running within the container:

```
# ps -ef

/bin/sh: 1: ps: not found

# apt update && apt install -y procps

...

# ps -ef

UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 00:17 ?        00:00:00 nginx: master process
nginx -g daemon off;
nginx         31       1  0 00:17 ?        00:00:00 nginx: worker process
nginx         32       1  0 00:17 ?        00:00:00 nginx: worker process
root          33       0  0 00:22 pts/0    00:00:00 /bin/sh
root         388      33  0 00:24 pts/0    00:00:00 ps -ef

#
```

In this listing we can see the two processes launched with the container and the shell we have launched within the container along with the `ps` command running under the shell.

Also note that the `top` command run from the host shows the host relative process ids (the main nginx process id was 3887 in the above example). Within the container the main nginx process has process id 1. This is the manifestation of the container's process namespace. Processes within a container have one id in the container and another id on the host.

The first process launched within each container is always process id 1 within the container.

Display the IP addresses within the container:

```
# ip a show

/bin/sh: 9: ip: not found

# apt install -y iproute2

...

# ip a show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever

#
```

Now exit the container:

```
# exit

~$
```

See if you can request the nginx root web page with `wget`:

```
~$ wget -qO- 172.17.0.3 | head

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }

~$
```

Since the process created by `exec` is not the primary process, it can safely be exited without disrupting the container.

8. CHALLENGE: container runtime options

Run `docker container run --help` and explore available options.

You have probably noticed that Docker gives random names to containers in the pattern of `adjective_noun`. Use the appropriate options with `docker container run` to run a container with the following requirements:

- use the nginx:1.19.0 image
- name the container `webserver`
- run the container in the background
- add an environment variable `CHALLENGE=containers`
    - exec into the container and list the environment variable to confirm it worked

9. Removing containers

Stopped containers will stay on your system perpetually unless deleted. You can delete unneeded containers with the `docker container rm` subcommand. You can also run interactive containers with the `--rm` switch, which will cause the container to self-delete when you exit the container.

Execute Docker commands to perform the following:

- Display all (`-a`) of the containers on your system
- Stop the container via `docker container stop <cid | cname>`
- Remove the container via `docker container rm <cid | cname>`
- What is the `--force` switch used for on `rm`, what happens?
- One way to remove all running/stopped containers in one go looks like this:

```
~$ docker container rm $(docker container stop $(docker container ls -qa))

2f9bb12d95a7
a56fd796940d
ac698b8c2d39
899f194f62db

~$ docker container ls -a

CONTAINER ID   IMAGE      COMMAND    CREATED    STATUS     PORTS      NAMES

~$
```

Congratulations you have completed the lab!