Chapter 4 Lab: Hands-on with Docker, Podman and Kubernetes

Overview

This lab provides practical experience installing Docker and Podman and setting up a Kubernetes cluster using **kind.** Once we have installed the container engine, we will understand the most commonly used container lifecycle operations and move on to build a container image using Dockerfile. We will uninstall Docker and install Podman to understand the Docker alternate tool. While there are many ways in which we can set up a Kubernetes cluster, we will use a simple method of setting up the cluster using **kind.**

Pre-Requisites

Ensure you have an Ubuntu 20.04 host. You can follow the previous lab to provision a virtual machine for yourself.

Exercise 4.1: Install Docker

In this exercise, we will install Docker and understand the most commonly used container lifecycle commands and then build an image using Dockerfile.

We can install Docker using the Docker's **apt** repository, or install it manually, or use a convenient script provided by Docker. For simplicity's sake, we will use the below method. Other installation methods can be found here.

1. Install Docker:

```
curl -fsSL https://get.docker.com/ | sh
```

2. Enable Docker to start on boot:

```
sudo systemctl enable --now docker
```

3. Check that the Docker service is running:

```
sudo systemctl status docker
```

4. Add current user to the **docker** group:

```
sudo usermod -aG docker $USER
```

5. Refresh shell session **(by exiting & logging in again)**.

6. Verify the Docker installation:

```
docker ps
```

Exercise 4.2: Manage Container Lifecycle with Docker

1. Now that we have installed and verified Docker, let's create our first container. This command pulls the **hello-world** image from Docker Hub and runs it as a container.

   ```
   student@ubuntu:~$ sudo docker container run hello-world
   ```

   Output:

   ```
   Unable to find image 'hello-world:latest' locally
   latest: Pulling from library/hello-world
   c1ec31eb5944: Pull complete
   Digest:
   sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493
   Status: Downloaded newer image for hello-world:latest

   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.
       (amd64)
    3. The Docker daemon created a new container from that image which runs the
       executable that produces the output you are currently reading.
   ```

```
 4. The Docker daemon streamed that output to the Docker client, which sent
it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

2. List the container running on our host, by executing the following command:

```
student@ubuntu:~$ sudo docker container ls
```

Output:

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
Note: No container is listed, that is because "docker container ls" command
only lists running container by default.
```

3. List the containers running on our host again, by specifying a different option:

```
student@ubuntu:~$ sudo docker container ls -a
```

Output:

```
CONTAINER ID   IMAGE         COMMAND    CREATED         STATUS
PORTS     NAMES

212315cf9ce9   hello-world   "/hello"   4 minutes ago   Exited (0) 4 minutes
ago             epic_feynman
```

4. In order to know all the options and get help from Docker, we can execute the following command:

```
student@ubuntu:~$ sudo docker container ls –help
```

Output:

```
Usage:  docker container ls [OPTIONS]

List containers

Aliases:
  docker container ls, docker container list, docker container ps, docker ps

Options:
  -a, --all             Show all containers (default shows just running)
  -f, --filter filter   Filter output based on conditions provided
      --format string   Format output using a custom template:
                        'table':            Print output in table format
with column headers (default)
                        'table TEMPLATE':   Print output in table format
using the given Go template
```

```
                          'json':              Print in JSON format
                          'TEMPLATE':          Print output using the given Go
template.
                          Refer to https://docs.docker.com/go/formatting/ for
more information about formatting output with templates
  -n, --last int          Show n last created containers (includes all states)
(default -1)
  -l, --latest            Show the latest created container (includes all
states)
      --no-trunc          Don't truncate output
  -q, --quiet             Only display container IDs
  -s, --size              Display total file sizes
```

5. Interact with a running container. This command starts up an **ubuntu** container and opens an interactive bash shell in the foreground:

```
student@ubuntu:~$ sudo docker container run -it ubuntu /bin/bash
```

Output:

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
a48641193673: Pull complete
Digest:
sha256:6042500cf4b44023ea1894effe7890666b0c5c7871ed83a97c36c76ae560bb9b
Status: Downloaded newer image for ubuntu:latest
root@8bbf327d2b32:/#
Note: Press exit to close the container shell.
```

6. Start an **ubuntu** container, but this time we will send the container to the background:

```
student@ubuntu:~$ sudo docker container run -dit ubuntu /bin/bash
```

Output:

```
c59dd04833466ee4aee856b574d918c7a3368b87b9ac7ca70b4ca9800555a33c
```

7. Now that the container is running in the background, we can interact with it by attaching to the container.

```
student@ubuntu:~$ sudo docker container attach c59dd04833466e
root@c59dd0483346:/#

Note: This time we will press ctrl+p,ctrl+q to send the container to
background.

root@c59dd0483346:/# read escape sequence

student@ubuntu:~$
```

8. Let's start an **nginx** web server and expose a port on the host to access the application. We will also make use of additional options like **--name** to name our container:

```
student@ubuntu:~$ sudo docker container run -dit --name webserver -p 8080:80
nginx
```

Output:

```
Unable to find image 'nginx:latest' locally

latest: Pulling from library/nginx

af107e978371: Pull complete

336ba1f05c3e: Pull complete

8c37d2ff6efa: Pull complete

51d6357098de: Pull complete

782f1ecce57d: Pull complete

5e99d351b073: Pull complete

7b73345df136: Pull complete

Digest:
sha256:2bdc49f2f8ae8d8dc50ed00f2ee56d00385c6f8bc8a8b320d0a294d9e3b49026

Status: Downloaded newer image for nginx:latest

51e39209944cd417424eeff133b00d0ca7dae4e44950300ca7929211fc04ca05
```

9. We have exposed the container on port 8080 of the localhost; let's access the application.

```
student@ubuntu:~$ curl localhost:8080

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
```

```
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

10. Create a custom html page and copy it to our container.

```
student@ubuntu:~$ cat > index.html << EOF

> Welcome to Docker Training!!

> EOF

student@ubuntu:~$ sudo docker container cp index.html
webserver:/usr/share/nginx/html/index.html

Successfully copied 2.05kB to webserver:/usr/share/nginx/html/index.html
```

11. Access the application; we should now see the default page changed by our copy command.

```
student@ubuntu:~$ curl localhost:8080

Welcome to Docker Training!!
```

12. Stop and remove the container.

```
student@ubuntu:~$ sudo docker container stop webserver
webserver
```

13. Remove all the stopped and completed containers.

```
student@ubuntu:~$ sudo docker container prune
 WARNING! This will remove all stopped containers.
 Are you sure you want to continue? [y/N] y
 Deleted Containers:
 51e39209944cd417424eeff133b00d0ca7dae4e44950300ca7929211fc04ca05
 8bbf327d2b32fdc73ff038ff38a7b6df4debcb59226f8e4b35d26b78a4668b99
 212315cf9ce9638786eb60ed801b48f0f43e83a0d32fd5cc222b376154ec8585

 Total reclaimed space: 1.127kB
```

14. There are additional container run options; explore the **`sudo docker container --help`** command and try additional options.

Exercise 4.3: Image Management with Docker

1. List the available images on our host. Notice when the images are not available on the host, they are by default downloaded from DockerHub.

```
student@ubuntu:~$ sudo docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
ubuntu        latest    174c8c134b2a   3 weeks ago     77.9MB
nginx         latest    d453dd892d93   2 months ago    187MB
hello-world   latest    d2c94e258dcb   8 months ago    13.3kB
```

2. If the image is already existing on the host, then creating a container is very quick. Let's see how to pull an image from the registry:

```
student@ubuntu:~$ sudo docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
661ff4d9561e: Pull complete
Digest:
sha256:51b67269f354137895d43f3b3d810bfacd3945438e94dc5ac55fdac340352f48
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

3. By default, the Daemon is configured to fetch the image from the Docker Hub registry and from the user account library, where Docker stores all the publicly available official images and fetches the image with tag as **`latest`**. We can override the default behavior by specifying what we want.

```
student@ubuntu:~$ sudo docker image pull gcr.io/google-samples/hello-app:2.0
2.0: Pulling from google-samples/hello-app
07a64a71e011: Pull complete
fe5ca62666f0: Pull complete
b02a7525f878: Pull complete
1933f300df8c: Pull complete
36cc1f0ec872: Pull complete
Digest:
sha256:7104356ed4e3476a96a23b96f8d7c04dfa7a1881aa97d66a76217f6bc8a370d0
Status: Downloaded newer image for gcr.io/google-samples/hello-app:2.0
gcr.io/google-samples/hello-app:2.0
```

4. Let us build a simple image using Dockerfile. We will use the **`index.html`** file we created in an earlier step and copy it to our new image.

```
student@ubuntu:~$ cat > Dockerfile <<EOF
> FROM nginx
> COPY . /usr/share/nginx/html
> EXPOSE 80/tcp
> CMD ["nginx", "-g daemon off;"]
> EOF


student@ubuntu:~$ sudo docker image build . -t nginx:version1
[+] Building 0.4s (7/7) FINISHED
docker:default
 => [internal] load .dockerignore
0.1s
 => => transferring context: 2B
0.0s
```

```
 => [internal] load build definition from Dockerfile
0.1s
 => => transferring dockerfile: 123B
0.0s
 => [internal] load metadata for docker.io/library/nginx:latest
0.0s
 => [internal] load build context
0.1s
 => => transferring context: 5.74kB
0.0s
 => [1/2] FROM docker.io/library/nginx
0.1s
 => [2/2] COPY . /usr/share/nginx/html
0.1s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:69be93e604b157a2dda142b3444100e5e24013f2c29db16bd25c0c8de84c28b2
0.0s
 => => naming to docker.io/library/nginx:version1
```

5. Create a container using the image we built and access the application.

```
student@ubuntu:~$ sudo docker container run -dit --name webapp -p 8081:80
nginx:version1
8fd1b47bafda05c96faba7d46486ca90f4a828c221cb5fd689e819592b839c84

student@ubuntu:~$ curl localhost:8081
Welcome to Docker Training!!
```

6. Clean up the unused images on the host by executing the following command:

```
student@ubuntu:~$ sudo docker image prune -a

WARNING! This will remove all images without at least one container
associated to them.

Are you sure you want to continue? [y/N] y

Deleted Images:

untagged: alpine:latest

untagged:
alpine@sha256:51b67269f354137895d43f3b3d810bfacd3945438e94dc5ac55fdac340352f
48

deleted:
sha256:f8c20f8bbcb684055b4fea470fdd169c86e87786940b3262335b12ec3adef418
```

```
deleted:
sha256:5af4f8f59b764c64c6def53f52ada809fe38d528441d08d01c206dfb3fc3b691

untagged: gcr.io/google-samples/hello-app:2.0
<Output truncated>
```

7. There are additional image options; explore the **sudo docker image --help** command and try additional options.

Exercise 4.4: Uninstall Docker

1. Uninstall the Docker Engine, CLI, containerd, and Docker Compose packages:

```
student@ubuntu:~$ sudo apt-get purge docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin docker-ce-rootless-extras

Reading package lists... Done

Building dependency tree

Reading state information... Done

The following packages were automatically installed and are no longer
required:

  libatasmart4 libblockdev-fs2 libblockdev-loop2 libblockdev-part-err2
libblockdev-part2 libblockdev-swap2 libblockdev-utils2 libblockdev2
libmbim-glib4 libmbim-proxy libmm-glib0 libnspr4

  libnss3 libnuma1 libparted-fs-resize0 libqmi-glib5 libqmi-proxy
libudisks2-0 libxmlb2 pigz slirp4netns usb-modeswitch usb-modeswitch-data

Use 'sudo apt autoremove' to remove them.

The following packages will be REMOVED:

  containerd.io* docker-buildx-plugin* docker-ce* docker-ce-cli*
docker-ce-rootless-extras* docker-compose-plugin*

0 upgraded, 0 newly installed, 6 to remove and 10 not upgraded.

After this operation, 410 MB disk space will be freed.

Do you want to continue? [Y/n] y

(Reading database ... 62643 files and directories currently installed.)

Removing docker-ce-rootless-extras (5:24.0.7-1~ubuntu.20.04~focal) ...
```

```
Removing docker-ce (5:24.0.7-1~ubuntu.20.04~focal) ...
<Output Trucncated>
```

2. Images, containers, volumes, or custom configuration files on your host aren't automatically removed. To delete all images, containers, and volumes, run:

```
student@ubuntu:~$ sudo rm -rf /var/lib/docker
student@ubuntu:~$ sudo rm -rf /var/lib/containerd
```

Exercise 4.5: Installing Podman

1. Let’ss download the latest Podman binaries and extract them to set up Podman on our host. Note: If you are running Ubuntu 20.10 or higher, you can simply use the package manager to install Podman.

```
student@ubuntu:~$ curl -fsSL -o podman-linux-amd64.tar.gz \
>
https://github.com/mgoltzsche/podman-static/releases/latest/download/podman-
linux-amd64.tar.gz
student@ubuntu:~$ tar -xf podman-linux-amd64.tar.gz
student@ubuntu:~$ sudo cp -r podman-linux-amd64/usr podman-linux-amd64/etc /
```

2. Verify the Podman installation:

```
student@ubuntu:~$ podman --version
podman version 4.8.2
```

3. To explore all the available options with Podman, execute:

```
student@ubuntu:~$ podman --help

Usage:
  podman [options] [command]

Available Commands:
  attach      Attach to a running container
  auto-update Auto update containers according to their auto-update policy
  build       Build an image using instructions from Containerfiles
  commit      Create new image based on the changed container
  compose     Run compose workloads via an external provider such as
docker-compose or podman-compose
  container   Manage containers
  cp          Copy files/folders between a container and the local
filesystem
  create      Create but do not start a container
  diff        Display the changes to the object's file system
  events      Show podman system events
  exec        Run a process in a running container
```

```
  export      Export container's filesystem contents as a tar archive
  generate    Generate structured data based on containers, pods or volumes
  healthcheck Manage health checks on containers
  help        Help about any command


<Output Truncated>
```

4. List the existing container, by executing the following command:

```
student@ubuntu:~$ sudo podman container ps
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS
NAMES
```

5. Create a container, by executing the following command:

```
student@ubuntu:~$ sudo podman container run hello-world

✔ docker.io/library/hello-world:latest
Trying to pull docker.io/library/hello-world:latest...
Getting image source signatures
Copying blob c1ec31eb5944 done   |
Copying config d2c94e258d done   |
Writing manifest to image destination

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.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent
it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

6. We can list the available images on our host by executing the following command:

```
student@ubuntu:~$ sudo podman image ls
REPOSITORY                      TAG         IMAGE ID      CREATED       SIZE
docker.io/library/hello-world   latest      d2c94e258dcb  8 months ago  26.3
kB
```

7. An Image can be built using Podman as well. We will use the same Dockerfile as in the earlier lab exercise.

```
student@ubuntu:~$ sudo podman image build . -t nginx:podman

STEP 1/4: FROM nginx
✔ docker.io/library/nginx:latest
Trying to pull docker.io/library/nginx:latest...
Getting image source signatures
Copying blob af107e978371 done   |
Copying blob 5e99d351b073 done   |
Copying blob 336ba1f05c3e done   |
Copying blob 8c37d2ff6efa done   |
Copying blob 51d6357098de done   |
Copying blob 782f1ecce57d done   |
Copying blob 7b73345df136 done   |
Copying config d453dd892d done   |
Writing manifest to image destination
STEP 2/4: COPY . /usr/share/nginx/html
--> 95fa83a8887f
STEP 3/4: EXPOSE 80/tcp
--> aecf247e1625
STEP 4/4: CMD ["nginx", "-g daemon off;"]
COMMIT nginx:podman
--> 7958e43aed3c
Successfully tagged localhost/nginx:podman
7958e43aed3c9591ca7ab5cae328dfdc8344a8a5420b07ac473bed169319869d
```

8. Podman provides a command line interface (CLI) familiar to anyone who has used the Docker Container Engine. Most users can simply alias Docker to Podman (**`alias docker=podman`**) without any problems. Explore further options and commands using Podman.

Exercise 4.6: Setting up a Kubernetes Cluster with kind

kind is a tool for running local Kubernetes clusters using Docker container nodes. kind was primarily designed for testing Kubernetes itself, but may be used for local development or CI. We have uninstalled Docker in exercise 4.4; kind needs Docker as a prerequisite; let's follow Exercise **4.1** steps and install Docker.

Note: Docker should be installed on the node before proceeding further.

To interact with our Kubernetes cluster, **kubectl** is the primary go to command line tool. Let us install **kubectl** on our node by executing these commands:

1. Download the binary.

```
curl -sSL -O "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

2. Modify permissions:

```
chmod +x kubectl
```

3. Move to `/usr/local/bin`**:**

```
sudo mv kubectl /usr/local/bin
```

Now that the prerequisites are completed, we can install **`kind`** by running the following command:

1. For AMD64 / x86_64:

```
[ $(uname -m) = x86_64 ] && curl -Lo ./kind
https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
```

2. For ARM64:

```
[ $(uname -m) = aarch64 ] && curl -Lo ./kind
https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-arm64
```

3. Modify permissions:

```
chmod +x ./kind
```

4. Move the kind binary `/usr/local/bin`**:**

```
sudo mv ./kind /usr/local/bin/kind
```

5. We will create a simple cluster with default configuration. A config file with additional configurations can be created and passed to the **`create cluster`** command during the run time. To keep it simple, we are going to make use of default configurations.

```
kind create cluster
```

```
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-kind
```

6. Verify the cluster by checking how many namespaces we have on our cluster:

```
kubectl get ns
```

```
NAME                 STATUS   AGE
default              Active   36s
kube-node-lease      Active   36s
kube-public          Active   36s
kube-system          Active   36s
local-path-storage   Active   30s
```

Exercise 4.7: Deploy a Sample Application

In this exercise, we will deploy a sample application in Kubernetes and access it.

1. Create `webapp.yaml` file with contents below.

This file defines a Kubernetes deployment for your Sample Application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - image: nginx
        name: nginx
```

2. Deploy the application:

```
kubectl apply -f webapp.yaml
```

3. Verify the deployment:

```
kubectl get deployments
```

```
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
webapp   2/2     2            2           89s
```

4. Set up port forwarding to access the application.

In this step, we are establishing a port forwarding rule that redirects network traffic from a specific port on your local machine to the corresponding port on the Kubernetes pod hosting the server. By doing so, you enable direct access to the server via `http://localhost:8080` from your local computer. This action bridges the network gap between your local environment and the isolated Kubernetes pod, allowing you to test and interact with the deployed application as if it were running locally. **It's important to run this in a new terminal tab to keep the port forwarding active throughout your testing session.**

```
kubectl port-forward deploy/webapp 8080:80
```

```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

5. Test the sample application by sending an HTTP request.

```
curl http://localhost:8080/
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
<Output Truncated>
```

6. Cleanup by deleting the deployment:

```
kubectl delete deploy webapp
```